WQuestions

UNA GRAMÁTICA UNIVERSAL DE LA ARQUITECTURA DE DATOS

JOSÉ ABANTO MARÍN

LA TORRE DE BABEL

Cada industria, su propio lenguaje. Datos fragmentados, sistemas incompatibles, conocimiento aislado.

UN LENGUAJE COMÚN

Un estándar universal basado en la lingüística y la ciencia formal para conectar datos, sistemas y personas.

WQuestions propone un vocabulario universal para estructurar, conectar y comprender cualquier tipo de información.

DE LA TEORÍA A LA PRÁCTICA

Casos reales en Python aplicados a sectores comerciales y científicos.

UN MODELO ESCALABLE

Diseñado para funcionar en cualquier dominio, en cualquier industria, en cualquier idioma.

EL MOMENTO ES AHORA

Los Modelos de Lenguaje (LLMs) ofrecen la tecnología ideal para llevar este lenguaje común a escala global.

SI PODEMOS PREGUNTARLO, PODEMOS ESTRUCTURARLO.
SI PODEMOS ESTRUCTURARLO, PODEMOS COMPRENDERLO Y TRANSFORMARLO.

ARQUITECTURA UNIVERSAL DE LA INFORMACIÓN

Las preguntas como *coordenadas*

Quién, qué, dónde, cuándo, cuánto, cuál y cómo. Siete preguntas bastan para organizar la información de cualquier dominio del mundo —y para que la inteligencia artificial hable nuestro idioma sin perder un gramo de rigor.

Q quién · **O** qué · **L** dónde · **T** cuándo · **N** cuánto · **K** cuál · **M** cómo

José Abanto Marín · edición en hipertexto

CÓMO LEER ESTE LIBRO

Cada capítulo está construido en capas. El primer tercio cuenta una historia y da la intuición; las secciones siguientes bajan al modelado, al código y a los precedentes académicos. Si solo buscas la visión estratégica, lee en superficie; si buscas implementar, sigue hasta las cajas de **decisión de diseño** y los bloques de código.

APERTURA

00 Introducción

Hay una intuición que cualquier niño domina antes de saber qué es un sustantivo, y que la industria del software lleva décadas olvidando. Este libro trata de recuperarla y volverla arquitectura.

Existen seis palabras que un niño de cinco años usa con la soltura de un experto: **quién, qué, dónde, cuándo, cómo y por qué**. Las pronuncia mucho antes de aprender en la escuela qué cosa es un verbo o un adjetivo. Las usa cuando todavía confunde el ayer con el mañana, cuando ignora que existen océanos al otro lado del mundo y cuando los números mayores que diez le suenan a magia. Las usa, en suma, porque esas preguntas son las piezas con las que el cerebro humano empieza a ensamblar la realidad para que tenga sentido.

> **ADELANTO**
> Esas mismas preguntas aparecen en Aristóteles[1], en los manuales de periodismo de 1900 y en el orden en que los niños adquieren el lenguaje. El capítulo 6 rastrea esa convergencia.

Lo notable no es que el niño las use, sino que nunca deja de usarlas. Cambian los temas (pasamos de preguntar quién rompió el florero a preguntar quién firmó el contrato), pero la forma de la pregunta permanece idéntica a lo largo de toda una vida y a lo largo de toda la historia. Son, podríamos decir, invariantes: el armazón estable bajo la infinita variedad de lo que queremos saber.

Una intuición que la industria olvidó

Ahora trasladamos la escena a una oficina, un martes por la tarde. Una desarrolladora intenta que el sistema de facturación de una clínica converse con la plataforma estatal de salud. Tiene delante toda la información necesaria: sabe qué se le diagnosticó al paciente, quién lo atendió y cuándo. El problema no son los datos, sino que cada sistema los nombra a su manera. Uno exige una estructura rígida; el otro habla un dialecto distinto. Y se le va la tarde escribiendo traductores a mano (puentes, *scripts*, exportaciones a medida) solo para que dos bases de datos se pongan de acuerdo sobre un hecho que, para aquel niño de cinco años, sería elemental.

Ese desgaste no es un gaje del oficio. Es el síntoma de una falla arquitectónica profunda en cómo la industria modeló la información durante décadas. Observa el mismo hecho clínico (un diagnóstico) tal como lo guardan tres sistemas que deberían entenderse:

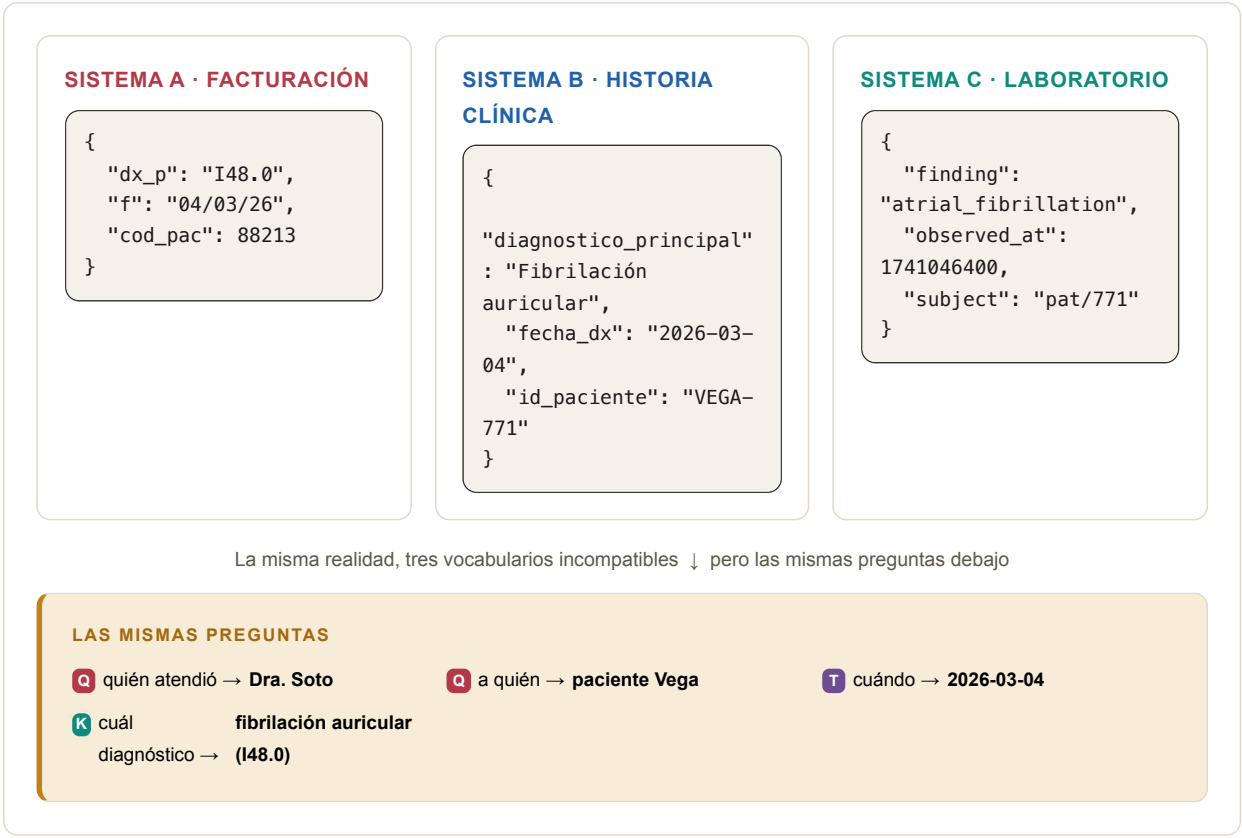


Figura 0.1. Tres sistemas describen el mismo diagnóstico con claves distintas (`dx_p`, `diagnostico_principal`, `finding`). Pero todos responden, sin saberlo, a las mismas preguntas. Esa capa común es la que este libro hace explícita.

Si los tres sistemas declararan de antemano *qué pregunta responde cada campo*, la traducción dejaría de ser un proyecto y pasaría a ser una obviedad. No haría falta que `dx_p` y `finding` se conocieran entre sí: bastaría con que ambos admitieran que están respondiendo a la pregunta «¿cuál es el diagnóstico?». Esa es, enunciada en una línea, la apuesta del libro.

Que quede dicho desde ya: lo que está en juego no es ordenar el diagnóstico de un paciente como Vega, sino poder preguntar por toda la clínica a la vez —cuántos casos como el suyo, qué condición encabeza el mes— sin rearmar nada. Ese salto de uno a todos es el que recorre el libro.

La tesis

IDEA CENTRAL

Las preguntas básicas de toda la vida (**quién, qué, dónde, cuándo, cuánto, cuál y cómo**) son suficientes para organizar, con precisión técnica, la información de cualquier dominio del mundo.

Quizá notes un pequeño desplazamiento respecto de las seis preguntas con las que abrimos. Es deliberado. Al exigirles precisión, dos preguntas que parecían una sola se separan, y una que parecía pregunta resulta ser otra cosa:

LA INTUICIÓN DE LA INFANCIA

quién · qué · dónde · cuándo · cómo · por qué

El genérico *qué* esconde tres preguntas distintas; el *por qué* no apunta a una cosa, sino que enlaza un hecho con otro.

LAS SIETE COORDENADAS

- **Q** quién · agente
- **O** qué · objeto, evento, situación
- **L** dónde · lugar
- **T** cuándo · tiempo
- **N** cuánto · número, magnitud
- **K** cuál · clase, categoría
- **M** cómo · predicado, conexión

Y el *por qué* queda como **relación entre hechos**, no como coordenada.

Figura 0.2. Al llevar la intuición a la precisión, *cuánto* y *cuál* se revelan como preguntas que el genérico *qué* tenía escondidas, y *por qué* se descubre como una relación. La intuición no se reemplaza: se afina hasta volverse siete coordenadas exactas.

De esas siete, seis fijan la *posición* de un hecho (quién, qué, dónde, cuándo, cuánto, cuál) y la séptima, *cómo*, aporta los enlaces que amarran todo: los predicados. Por eso conviene pensar cada hecho no como una fila de una tabla, sino como un **punto en un espacio de coordenadas**.

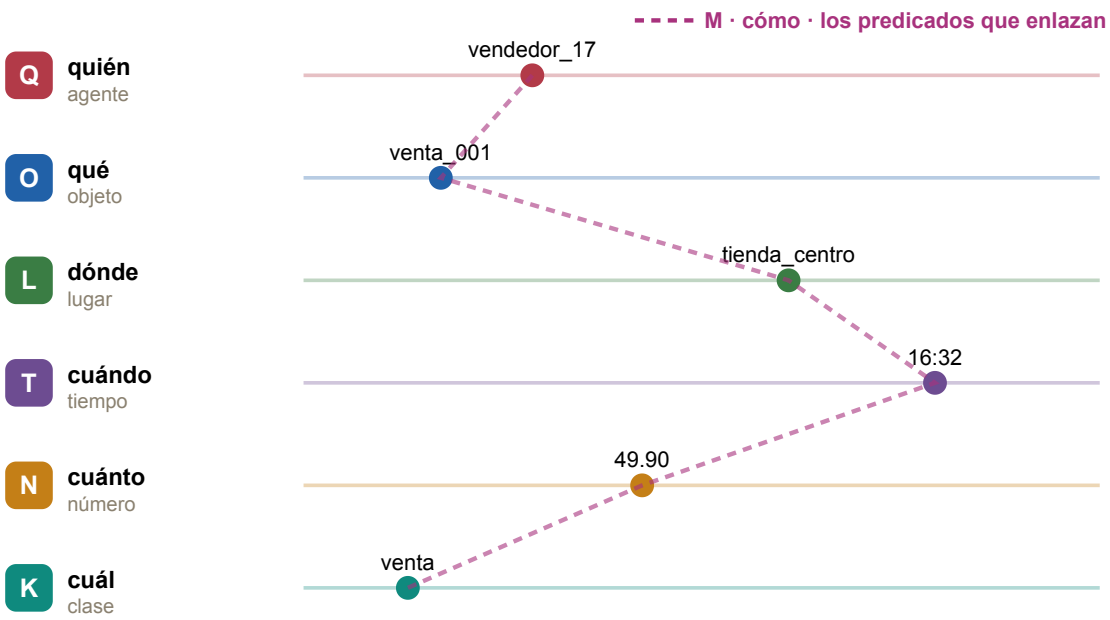


Figura 0.3. El hecho `venta_001` (una camiseta que el vendedor 17 vendió a las 16:32 por 49,90 dólares) leído como un punto: una lectura por cada eje de valor. La línea magenta que une las lecturas son los predicados (*cómo*), el séptimo eje, que conecta el hecho con cada uno de sus valores.

No hay trucos ocultos ni capas de complejidad innecesaria. Pero, como ocurre con los verdaderos cambios de perspectiva, cuando tomas esta premisa y la llevas hasta sus últimas consecuencias, el panorama entero cambia: cambia cómo diseñas una base de datos, cómo construyes un agente de inteligencia artificial, cómo conectas sistemas que hoy no saben hablarse, e incluso cómo le explicas un negocio nuevo a un programador que recién empieza. Lo hemos puesto a prueba. Funciona. Y el propósito de este texto es mostrarte la ingeniería exacta detrás de ese funcionamiento.

Por qué hace falta un libro entero

Si la idea central cabe en un párrafo, la pregunta es legítima: ¿hace falta un libro? Sí, por una razón concreta. En ingeniería de software, entre **enunciar** una propuesta y **poder usarla** en producción hay un abismo. Para cruzarlo, una arquitectura como esta necesita cinco cosas, y cada una ocupa una parte del libro:

1 UNA INTUICIÓN CLARA

Demostrar que la idea no es un capricho de diseño, sino que se apoya en cómo la mente, la lingüística y la ciencia formal organizan la información.

2 UN MODELO FORMAL

Reglas escritas con precisión (notación de conjuntos cuando hace falta, prosa clara cuando basta) que no dejen ambigüedad a quien implemente.

3 PRUEBAS DE ESTRÉS

Aplicar el modelo a dominios reales y exigentes (nada de ejemplos de juguete) para que el lector escéptico vea que no colapsa.

4 DIÁLOGO CON EL ECOSISTEMA

Conversar de frente con las ontologías existentes, los estándares de la industria y las herramientas modernas de IA.

5 UNA HOJA DE RUTA CLARA

Qué está resuelto y qué falta para que esto sea infraestructura de uso diario.

Qué es, exactamente, WQuestions

A lo largo del libro construirás conmigo una arquitectura llamada **WQuestions**. Conviene pensarla no como otra base de datos rígida, sino como un *protocolo de normalización semántica*: un motor que toma las preguntas cotidianas y las convierte en los ejes formales de un espacio geométrico. Es un mapa de coordenadas para el conocimiento. Al plantearlo como un estándar de interoperabilidad (y no como una taxonomía que se impone desde arriba) deja de competir con las ontologías existentes y empieza a hacerlas dialogar.

> *Si todas las aplicaciones del mundo estructuraran su información sobre los ejes de las preguntas, un agente de IA tendría que aprender a hablar una sola vez.*
>
> LA APUESTA DEL LIBRO

Por qué este libro aparece ahora

Si hubiéramos propuesto esta arquitectura hace cinco años, su adopción habría sido casi imposible. Faltaba una pieza: una capa capaz de traducir con fluidez entre el modelo de datos estructurado y el lenguaje natural de las personas. Hoy esa pieza existe: son los grandes modelos de lenguaje. Su capacidad para invocar herramientas de forma autónoma (*function calling*) y sostener flujos de trabajo dinámicos reescribió las reglas de la integración.

Cuando un agente moderno necesita registrar una venta o leer un dato hospitalario, los «cables» ya están tendidos: domina JSON, valida esquemas, entiende tipos. Lo que le sigue faltando a la industria es un **vocabulario común** que viaje por esos cables. Hoy cada API expone su dialecto privado, y el agente gasta tiempo y *tokens* aprendiendo un idioma nuevo por cada herramienta. El costo de no tener ese vocabulario crece más rápido de lo que parece.

Figura 0.4. Conectar *n* sistemas de a pares cuesta `n(n−1)/2` puentes (rojo): a diez sistemas, cuarenta y cinco. Con un vocabulario común al centro, cada sistema se conecta una sola vez (verde): diez. Pasa el cursor sobre los puntos para ver las cifras.

La tesis de este libro es que las preguntas son, por su propia naturaleza, ese idioma común. Y el momento tecnológico para implementarlo es ahora.

Para quién está escrito

Diseñé el texto apuntando a tres lectores simultáneos. Si el equilibrio es el correcto, cada uno encontrará valor sin estorbarse con el de los demás.

🧭

CURIOSIDAD ESTRUCTURAL

Si te fascina cómo la IA intenta mapear el conocimiento humano, este texto te hará mirar con rigor algo que damos por sentado: la naturaleza de las preguntas.

⚙️

PROFESIONAL TÉCNICO

Si eres arquitecto de software o ingeniero de datos, aquí tienes un marco concreto, probado y desmenuzado: ocho industrias modeladas, decisiones justificadas y un lexicon para arrancar.

🎓

INVESTIGACIÓN

Si trabajas en semántica, bases de datos o lingüística computacional, cada propuesta está anclada en la literatura: Barwise y Perry[11], Davidson[12], Gärdenfors[13], CIDOC CRM[4], FrameNet[14].

Para que las tres lecturas convivan, cada capítulo está **estratificado**: abre con una escena accesible y aumenta su densidad hacia el final. Si solo buscas la visión estratégica, el primer tercio basta; si buscas el sustento formal, sigue hasta las cajas de decisión de diseño; si quieres ver código aplicado, tu lugar es la Parte V.

El recorrido

El desarrollo del modelo se reparte en seis bloques:

Parte I · El problema. La torre de Babel de los datos: la información existe, pero está atrapada en esquemas que no se hablan. Lo ilustramos con el elefante en la habitación oscura (cada sistema palpa una parte y la nombra distinto) y apostamos a que la solución es anterior y más simple que cualquier base de datos.

Parte II · Las siete coordenadas. Diseccionamos quién, qué, dónde, cuándo, cuánto, cuál y cómo (su carga semántica, sus trampas) y cerramos mostrando por qué son *estas* preguntas y no otras.

Parte III · Cómo funcionan juntas. El ensamblaje: cómo un evento se vuelve la unidad atómica, cómo la intersección de los ejes crea una geometría consultable, cómo se enlaza la causalidad, cómo se reconoce a una misma entidad (el Juan de la tienda y el de la clínica) aunque cada sistema le ponga un identificador distinto, y cómo todo esto se apoya en la tecnología que ya conoces: objetos, bits, grafos y cadenas de bloques.

Parte IV · Del lenguaje a los hechos. El puente entre la ambigüedad del lenguaje y la rigidez de los datos: el papel del verbo, el diseño del lexicon y los problemas lingüísticos difíciles.

Parte V · En la práctica. La sala de máquinas: ocho dominios industriales completos (del banco a la clínica con su historia clínica, hospitalización y farmacia; de la municipalidad y sus trámites a la minera con su comisionamiento, mantenimiento y punchlists) y cuatro escenarios de alto estrés técnico (música, química, fútbol y contratos).

Parte VI · IA, futuro y cierre. Cómo WQuestions potencia a los LLMs, qué aplicaciones desbloquea, la prueba más exigente (describirse a sí mismo), la seguridad del grafo compartido y los retos pendientes.

Una nota sobre el método

CÓMO SE CONSTRUYÓ ESTE MODELO

Esta arquitectura no se diseñó en abstracto sobre una pizarra para después forzar a los datos a encajar. Al contrario: emergió de forma empírica, dominio tras dominio, chocando con las fricciones reales que cada negocio exigía resolver. Ninguna regla está aquí por amor a la elegancia; cada decisión técnica existe porque un caso de uso real la reclamó.

Esa génesis empírica es su mayor garantía. El modelo tomó su forma actual el día en que comprendimos que ocho industrias sin relación entre sí estaban pidiendo la *misma* solución estructural, disfrazada con vocabularios distintos. Cuando ocho problemas independientes convergen en la misma respuesta, esa respuesta deja de ser una hipótesis.

La arquitectura se expone con la convicción de quien la ha puesto a prueba, y se defiende con argumentos, no con adjetivos.

Empecemos.

PARTE I · EL PROBLEMA

01

La torre de Babel de los datos

La información que salva una vida puede existir, estar guardada y no estar oculta para nadie, y aun así ser imposible de consultar. No es un fallo del azar: es un fallo de diseño que hemos repetido durante décadas.

Son las dos de la madrugada y entra en urgencias una mujer de cuarenta y dos años. La llamaremos **Vega**. Está consciente, pero suda frío y le falta el aire; refiere un dolor en el pecho que no cede. El médico de guardia tiene minutos, no horas, para decidir, y para decidir bien necesita responder tres preguntas concretas: qué medicación toma a diario, si tiene alergias documentadas y si ya vivió antes un episodio parecido. Teclea su nombre en el sistema del hospital. La pantalla devuelve tres líneas: una bronquitis tratada hace cuatro años. Nada más.

> **HILO CONDUCTOR**
> La paciente `vega` reaparece en el cierre del libro. Cuando lleguemos a la conclusión, esta misma escena se resolverá: la consulta que aquí es imposible será una sola línea.

Lo desesperante es que la información existe. Vega tiene una endocrinóloga que gestiona su historia desde hace siete años. Hace unos meses la evaluó un cardiólogo en otra ciudad. Y ya pasó por una urgencia muy parecida en un tercer centro, donde le hicieron un electrocardiograma completo. Cada dato está escrito, guardado en algún disco, y ninguno de esos profesionales tiene el menor interés en ocultarlo. El obstáculo no es el secreto ni la pérdida: es estructural. Los datos de Vega están repartidos en esquemas que no se hablan entre sí.

La endocrinóloga usa un software que guarda el diagnóstico en un campo llamado `diagnostico_principal`. El sistema del cardiólogo, hecho por otra empresa, lo registra como `dx_p`. Para la medicación, la primera clínica emplea una estructura plana (`medicacion_actual` con `nombre`, `dosis` y `frecuencia`); la segunda diseñó una base relacional con una tabla de `prescripciones` que pide un identificador numérico (`farmaco_id`), los miligramos por toma (`mg_por_toma`) y el número de tomas diarias (`tomas_dia`). Dos formas legítimas y razonables de anotar exactamente lo mismo. Y, sin embargo, incompatibles.

Si trabajas en software, ya sabes que conectar esas dos bases no es ciencia ficción. Es posible. Pero es un trabajo artesanal, caro y frágil: hay que programar un puente de código (un *middleware*), mapear a mano qué columna equivale a qué columna y escribir una excepción por cada formato de fecha o de número. Y ese puente caduca. El día en que una de las clínicas actualice su sistema y renombre un solo campo, el puente se rompe en silencio. Multiplica ese esfuerzo por miles de hospitales, decenas de proveedores y décadas de historias clínicas, y la integración deja de ser un reto de ingeniería para revelarse como un problema de diseño a escala planetaria. La información existe; lo que falta es la capacidad de razonar con ella.

> *La información de Vega no está perdida ni oculta. Está atrapada en idiomas que nunca acordaron hablar entre sí.*
>
> EL PROBLEMA DE BABEL

El mismo hecho, cuatro idiomas distintos

El drama de urgencias es la versión extrema; pero el mecanismo que lo causa late en el sistema más banal. Para verlo en cámara lenta, bajemos a un escenario sin víctimas: una tienda de ropa. A las 16:32, en la tienda del centro, el vendedor 17 le vende una camiseta a un cliente. La llamaremos `venta_001`. Es *un* hecho del mundo, uno solo. Y, sin embargo, si nos asomamos a los servidores del negocio, descubriremos que sus distintos sistemas anotan ese mismo instante en cuatro lenguajes radicalmente distintos.

Para el **punto de venta** (la caja) lo que importa es la transacción y quiénes la protagonizan. El hecho se guarda como una fila en una base relacional:

SQL · PUNTO DE VENTA

```
INSERT INTO ventas
  (id, cliente_id, producto_id, monto, fecha, vendedor_id)
VALUES (1, 1042, 88, 49.90, '2026-05-14', 17);
```

Para **contabilidad**, en cambio, la identidad del vendedor o del cliente es irrelevante. Su única obsesión es que las cuentas cuadren por partida doble. El mismo evento se descompone en un asiento:

ASIENTO · CONTABILIDAD

```
asiento: 2026-05-14
  debe:  cuentas_por_cobrar       49.90
  haber: ventas_brutas           42.29
  haber: impuesto_al_consumo      7.61
```

El equipo de **analítica de producto** ve otra cosa todavía. Le importa el comportamiento y el canal, para medir campañas. Estructura el hecho como un evento de telemetría:

JSON · TELEMETRÍA

```
{
  "evento": "compra_completada",
  "id_usuario": "u_1042",
  "sku": "sku-88",
  "ingreso": 49.90,
  "canal": "tienda_fisica",
  "id_sesion": "s_abcdef"
}
```

Y para **inventario**, por fin, solo ocurrió una cosa: una camiseta salió del almacén. El evento se reduce a un movimiento de existencias:

JSON · INVENTARIO

```
{
  "movimiento": "salida",
  "producto": 88,
  "cantidad": 1,
  "almacen": "tienda_central",
  "ref": "vta-1"
}
```

Observa la desconexión. Cuatro equipos, cuatro estructuras. Cada esquema es perfecto y coherente para quien lo diseñó. La fractura aparece cuando la dirección pide una visión global. Si le entregas estas cuatro piezas a una computadora, no tiene ninguna forma *obvia* de saber que las cuatro hablan del mismo suceso. Contabilidad no sabe quién vendió la camiseta; analítica no sabe de qué almacén salió; la caja no sabe separar el impuesto. Cada software capturó solo la *sombra* del evento que le interesaba y la archivó en su dialecto privado.

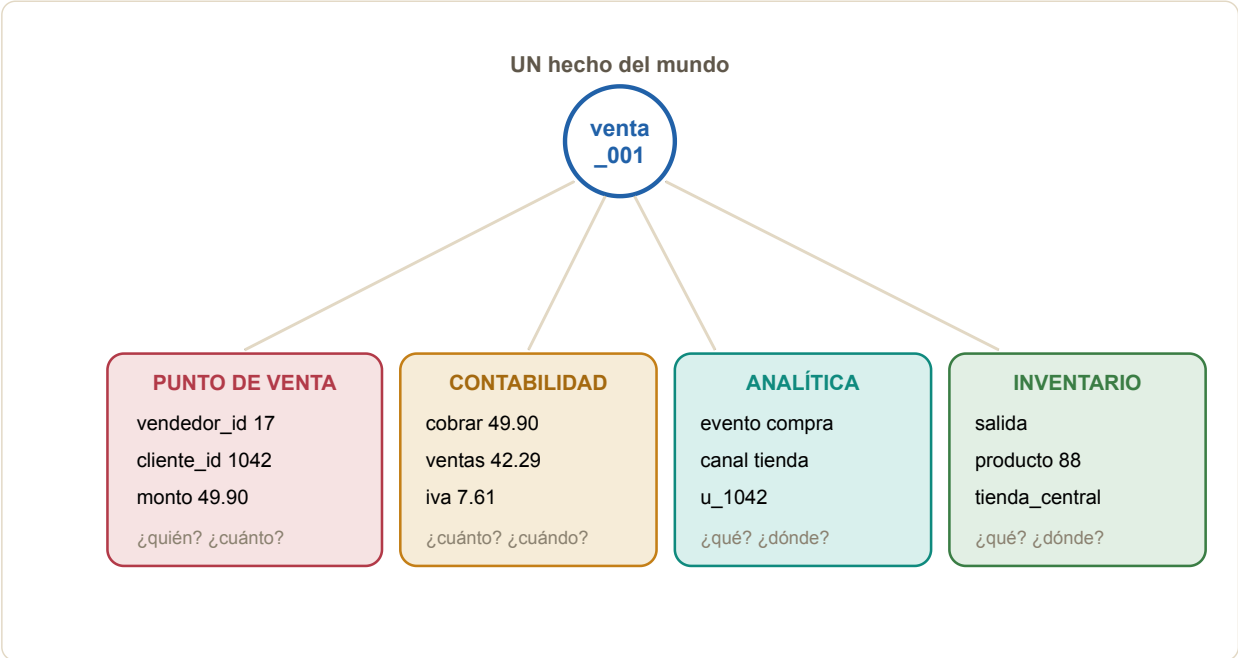


Figura 1.1. Un único hecho (la camiseta que el vendedor 17 vendió a las 16:32) proyecta cuatro *sombras* parciales. Cada sistema captura la cara que le importa (`vendedor_id`, `cobrar`, `evento`, `movimiento`) y la escribe en su dialecto. Ninguna sombra sabe de las otras, aunque debajo todas respondan a las mismas preguntas.

A esta fragmentación masiva la llamaremos, a lo largo del libro, **la torre de Babel de las ontologías**. Conviene desactivar una alarma: en datos, la palabra *ontología* no es un concepto filosófico inalcanzable.

> **ONTOLOGÍA (EN INGENIERÍA DE DATOS)**
>
> El catálogo oficial de conceptos que un sistema reconoce («cliente», «factura», «producto», «prescripción») junto con las reglas lógicas que dictan cómo se relacionan entre sí. Nada más esotérico que un diccionario con gramática. El problema no es que existan ontologías: es que cada departamento, empresa y disciplina construye la suya *cerrada*, aislándose del resto.

El elefante en la habitación oscura

Hay una vieja parábola que describe esta torre de Babel mejor que cualquier diagrama de arquitectura. En una habitación a oscuras encierran a un elefante, y entran a tientas varios hombres que jamás han tocado uno. El primero palpa una pata y, convencido, anuncia: «esto es una columna». El segundo abraza la trompa y lo corrige: «no, es una gruesa manguera». Un tercero acaricia la oreja y jura que es un abanico; otro topa con el colmillo y asegura que es una lanza pulida; el último se aferra a la cola y sentencia que es una cuerda. Ninguno miente. Cada uno informa con honestidad lo que sus manos encontraron. Y, sin embargo, todos se equivocan, porque confunden *la parte que tocaron* con *el animal entero*.

Esa es, exactamente, la situación de nuestros sistemas. El punto de venta toca la pata y la nombra `vendedor_id`; contabilidad abraza la trompa y la llama `cuentas_por_cobrar`; analítica acaricia la oreja y la registra como `evento`; inventario topa con el colmillo y anota `movimiento`. Cada ontología de dominio es uno de esos ciegos: palpa una región de la misma realidad, la describe con rigor impecable dentro de su alcance y la bautiza con un nombre que no se parece a ningún otro. La torre de Babel no nace de la mentira ni de la incompetencia; nace de que **cada quien toma su vista parcial por la totalidad**.

Pero el elefante existe. Hay un animal completo debajo de las cinco descripciones incompatibles, y la pregunta decisiva es qué lo reconstruye. La respuesta es el corazón de este libro: lo que devuelve el animal entero no es una sexta ontología más fina, sino un repertorio de preguntas anterior a todas ellas. «¿Quién?», «¿qué?», «¿dónde?», «¿cuándo?», «¿cuánto?» y «¿cuál?» no tocan una parte del elefante: son las coordenadas que sitúan cualquier parte dentro del mismo cuerpo. La pata y la trompa dejan de ser objetos rivales y pasan a ser dos respuestas (al «qué» y al «dónde») de un único hecho.

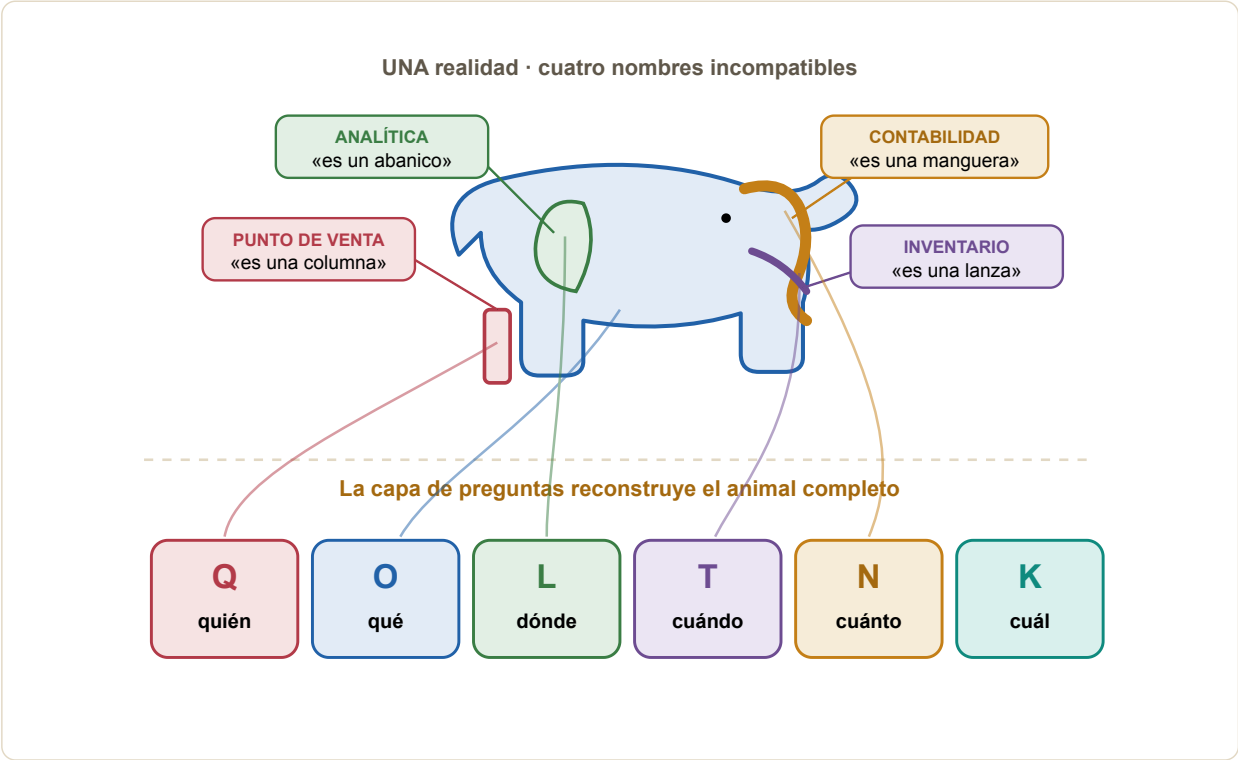


Figura 1.2. El elefante en la habitación oscura. Cada sistema toca una parte del mismo animal y la nombra distinto («columna», «abanico», «manguera», «lanza»): así nace la torre de Babel. Las ontologías de dominio capturan vistas parciales; *las preguntas son el animal completo*. La capa de coordenadas (`quién`, `qué`, `dónde`, `cuándo`, `cuánto`, `cuál`) no añade otra parte: reconstruye el todo situando cada parte dentro del mismo cuerpo.

La causa de todo el problema

La raíz de esta torre de Babel (y el punto de partida de este libro) se enuncia en una frase: hemos permitido que **cada sector invente, desde cero, su propia forma de entender la realidad**. La ingeniería médica no hereda nada de la contabilidad. El comercio no comparte cimientos con el derecho. La educación diseña bases que jamás conversarán con las del urbanismo. Cada industria levanta su pirámide con planos distintos y luego nos sorprende lo carísimo que resulta contratar gente para tender puentes colgantes entre las cimas.

Llevemos esa frase a un caso donde el aislamiento no cuesta dinero, sino una nave espacial.

EL PRECIO DE NO ACORDAR LAS UNIDADES

En 1999, la *Mars Climate Orbiter* de la NASA se desintegró al entrar en la atmósfera de Marte. La causa no fue un error de cálculo sino de *vocabulario*: un equipo de software entregaba el empuje en unidades inglesas (libra-fuerza por segundo) y el otro lo leía como si fueran unidades del sistema internacional (newton por segundo). Cada número, por separado, era correcto. La nave navegó hacia una órbita equivocada y se perdió.

Doscientos millones de dólares y años de trabajo evaporados porque dos sistemas no declararon, de forma explícita, *qué pregunta respondía cada número*. La distancia entre `dx_p` y `diagnostico_principal` es la misma distancia, solo que sin el cráter.

El patrón es siempre idéntico. Donde cada parte cultiva su jardín semántico amurallado, el coste de conectar no crece poco a poco: crece de forma combinatoria. Si tienes *n* sistemas y los quieres comunicar de a pares, el número de puentes que debes construir y mantener es `n(n−1)/2`. Diez sistemas no piden diez traductores: piden cuarenta y cinco. Y cada uno de ellos es un punto que puede romperse.

Figura 1.3. El coste de la fragmentación es combinatorio: conectar diez sistemas de a pares exige `n(n−1)/2` = 45 puentes (rojo). Un vocabulario común al centro reduce el problema a un solo punto de conexión por sistema (verde). Pasa el cursor sobre las barras para ver las cifras.

Por eso el integrador de datos (ese rol que casi ninguna escuela enseña y que casi toda empresa necesita) existe. Su oficio es escribir y vigilar esos *n(n−1)/2* puentes: los ETL que vacían una base en otra, los mapeos columna a columna, las excepciones por cada formato de fecha. Es trabajo valioso y, a la vez, trabajo que no debería existir. Es el impuesto que paga la industria por un pecado original de diseño.

Lo que ya se intentó (y por qué no bastó)

Sería injusto sugerir que nadie vio el problema. Llevamos décadas intentando derribar la torre, y los intentos se agrupan en dos familias, opuestas y simétricas en su fracaso. Vale la pena entenderlas, porque marcan exactamente el hueco que este libro quiere ocupar.

VÍA 1 · LAS GRANDES ONTOLOGÍAS

Construir esquemas formales, ricos y minuciosos para cada dominio. Funcionan con una precisión admirable… dentro de su parcela. Pero son **profundas y estrechas**: modelar un campo nuevo cuesta años de comité, y dos ontologías profundas vuelven a necesitar un puente entre ellas. Resuelven Babel *dentro* de cada torre, no *entre* torres.

VÍA 2 · LA EXTRACCIÓN ABIERTA

Renunciar al esquema y dejar que una máquina extraiga relaciones del texto libre (la llamada *OpenIE*[23]). Es **ancha pero informe**: cubre cualquier tema, sí, pero produce tripletas sin tipos estables, sin unidades, sin distinguir un agente de un lugar. Mucha cobertura y poca estructura: imposible de consultar con rigor.

EL CALLEJÓN DE LAS DOS VÍAS [4]

Las ontologías clásicas (de las que CIDOC CRM o las redes de *frames* son ejemplos refinados) ganan en estructura lo que pierden en alcance. La extracción abierta gana en alcance lo que pierde en estructura. La industria osciló entre ambos extremos sin encontrar el punto medio: un esquema *a la vez ancho y estructurado*. Ese punto medio es lo que perseguimos, y la pista de dónde buscarlo no está en la informática, sino en algo mucho más antiguo.

Y aquí cabe una pregunta incómoda, la que abre la puerta a todo lo demás: ¿y si la solución a este caos fuera *anterior* (y mucho más simple) que cualquier diseño de base de datos? ¿Y si, mucho antes de que existieran los hospitales, los bancos o el comercio internacional, la humanidad ya hubiera fijado un esquema cognitivo universal que nadie, en ningún idioma, puede evitar usar?

La apuesta de este libro

Volvamos a la venta una última vez. Si detuviéramos a cualquier persona en la calle (sin nociones de bases de datos ni de código) y le pidiéramos describir lo que pasó en esa tienda, su respuesta natural sería algo muy parecido a esto:

> *«El vendedor (**quién**) vendió (**qué**) una camiseta (**qué**) en la tienda del centro (**dónde**) esta tarde (**cuándo**) por casi cincuenta dólares (**cuánto**).»*

Sin el menor esfuerzo, cualquier ser humano produce una descripción **estructurada, completa y combinable**. La mente no percibe el mundo como un bloque de texto ni como una hoja de cálculo: descompone cualquier evento con un repertorio brevísimo de preguntas fundamentales. Y aquí está el giro: esas preguntas son exactamente las coordenadas que las cuatro sombras de la venta estaban capturando por separado, sin saberlo.

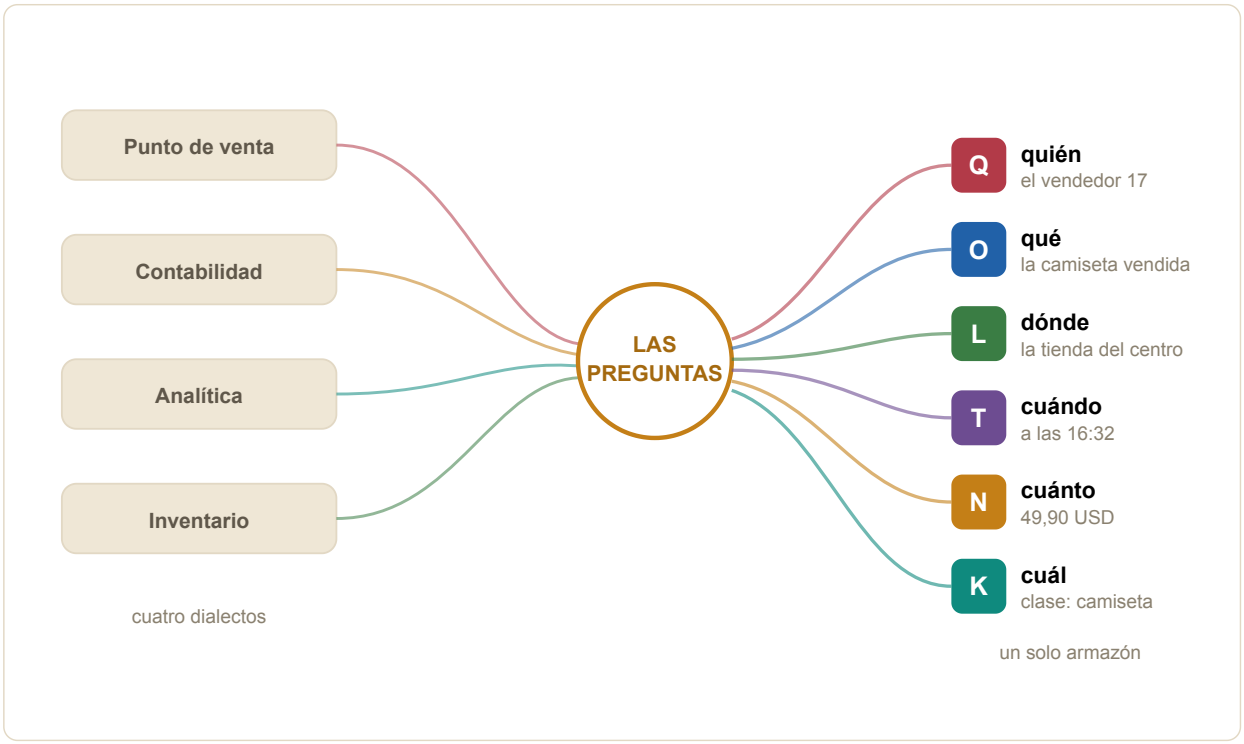


Figura 1.4. Los cuatro dialectos de la venta no son cuatro realidades: son cuatro recortes de la misma. Debajo de todos late un único armazón de preguntas. Si los sistemas declararan *qué pregunta responde cada campo*, convergerían sin necesidad de conocerse entre sí.

Esa es, enunciada con todas sus letras, la apuesta del libro:

> **LA APUESTA**
>
> Existe un conjunto reducido y estable de preguntas primarias (**siete**) con poder descriptivo suficiente para modelar cualquier hecho factual, en cualquier dominio industrial y en cualquier idioma: **quién, qué, dónde, cuándo, cuánto, cuál y cómo**. Al diseñar las bases de datos alrededor de estas coordenadas cognitivas (y no de la terminología de turno de cada industria) se neutraliza de raíz la fragmentación semántica. La torre de Babel se desmorona desde los cimientos.

Y viene un beneficio que en 1999, o en 2010, habría sonado a ciencia ficción y hoy es decisivo: **cualquier inteligencia artificial entrenada con lenguaje humano entiende esta estructura de fábrica**, porque es la misma con la que aprendió a leer. Un modelo de lenguaje no necesita que le expliques qué es un «quién» o un «cuándo». Esas categorías ya viven en él. Si los datos del mundo se expusieran sobre los ejes de las preguntas, un agente tendría que aprender a hablar una sola vez —y no un dialecto nuevo por cada API que toca.

Conviene separar aquí las siete coordenadas en dos grupos, porque así las recorreremos. Seis fijan la *posición* de un hecho (dónde cae en cada dimensión de la realidad); la séptima, *cómo*, aporta los enlaces que amarran las otras seis entre sí:

Q **quién** y O **qué** y L **dónde** y T **cuándo** son las cuatro coordenadas más concretas: el agente, el objeto o evento, el lugar y el tiempo. Son las que un niño domina primero y las que ningún sistema, por raro que sea su dialecto, deja de registrar.

N **cuánto** y K **cuál** son las dos que el genérico «qué» suele esconder: la magnitud (con sus unidades, que la *Mars Climate Orbiter* aprendió a la mala) y la clase a la que algo pertenece. Y M **cómo** es la séptima: los predicados que conectan todo lo demás.

Anticipando un poco la maquinaria que viene, así se ve un fragmento del hecho `venta_001` expresado ya en el formato del modelo: cada línea es una tripleta que conecta el hecho con uno de sus valores, y la firma final declara a qué par de ejes pertenece:

TRIPLETAS · WQUESTIONS

```
(venta_001, agente,   vendedor_17)   ∈ M(O, Q)
(venta_001, lugar,    tienda_centro) ∈ M(O, L)
(venta_001, instante, 16:32)         ∈ M(O, T)
(venta_001, monto,    49.90)         ∈ M(O, N)
(venta_001, clase,    camiseta)      ∈ M(O, K)
```

No te detengas todavía en la notación; el capítulo 7 la desarma pieza por pieza. Lo único que importa ver aquí es que las cuatro sombras de la Figura 1.1 acaban de fundirse en un solo registro, y que ese registro es legible (de izquierda a derecha, como una frase) por cualquiera de los cuatro sistemas y por una máquina por igual.

Una promesa que se puede comprobar

No te pido que tomes esto como un acto de fe. La fuerza de la apuesta no está en que suene elegante, sino en que estas siete preguntas no son un invento moderno. Filósofos griegos, juristas romanos, escuelas de periodismo del 1900 y los propios niños que aprenden a hablar convergen (sin coordinarse, separados por siglos) en la misma lista. Cuando tantas tradiciones independientes llegan a la misma respuesta, esa respuesta deja de ser una hipótesis y empieza a parecer un descubrimiento.

MÁS ADELANTE

La demostración de *por qué estas siete y no otras* (la convergencia entre Aristóteles[1], la retórica clásica y la adquisición del lenguaje) tiene capítulo propio: el capítulo 6.

A esa convergencia (la prueba de por qué estas preguntas y no otras) le dedicamos un capítulo entero. Pero no hace falta esperar a la demostración para empezar a construir. De hecho, conviene construir primero: ver la maquinaria funcionando convence más que cualquier argumento.

CÓMO NACIÓ ESTE MODELO

Esta arquitectura no se diseñó en abstracto para luego forzar a los datos a encajar. Emergió al revés: dominio tras dominio, chocando con las fricciones reales que cada negocio exigía resolver. El día decisivo fue aquel en que comprendimos que ocho industrias sin relación entre sí (de un spa a un banco, de una clínica a una mina) estaban pidiendo la *misma* solución estructural, disfrazada con vocabularios distintos. Cuando ocho problemas independientes convergen en la misma respuesta, esa respuesta deja de ser una opinión.

Lo que sigue es, sencillamente, la ingeniería. Vamos a diseccionar las siete coordenadas una por una: su carga de significado, sus trampas, los casos límite donde parecen romperse. Empezaremos por las cuatro más concretas (quién, qué, dónde y cuándo), porque son el suelo firme sobre el que se levanta todo lo demás. La paciente Vega nos espera al final del camino; cuando volvamos a su sala de urgencias, la consulta imposible de esta noche será una sola línea.

02

Quién, qué, dónde, cuándo

Cuatro preguntas bastan para levantar el esqueleto de cualquier hecho. Y cada una, vista de cerca, esconde una trampa que arruina los sistemas que la ignoran.

El vendedor entregó la camiseta sobre el mostrador a las cuatro y media de la tarde. La frase tiene once palabras y la entiendes sin pensar: alguien hizo algo, con una cosa, en un sitio, en un instante. No hubo esfuerzo consciente, pero tu mente acaba de ejecutar una hazaña silenciosa. En menos de lo que dura un parpadeo separó la escena en cuatro piezas de naturaleza completamente distinta y le asignó a cada una su casillero exacto: **una persona que actúa** (el vendedor), **una cosa que cambia de manos** (la camiseta), **un lugar físico** (el mostrador) y **un momento del reloj** (las 16:32).

Y no solo las separó: las encajó sin equivocarse. A nadie le pasaría por la cabeza que «el mostrador» sea la persona, ni que «las cuatro y media» sea el objeto que cambió de manos. Esa operación de descomponer un hecho y colocar cada fragmento en su lugar correcto es, literalmente, la materia prima de todo este libro. Las cuatro preguntas que tu cerebro acaba de responder a ciegas (*quién*, *qué*, *dónde*, *cuándo*) son los **cuatro pilares**: las dimensiones que comparecen en cualquier descripción de algo que sucede, sin importar el idioma ni el grado de tecnicismo. Si falta uno, la historia se siente incompleta. Si están los cuatro, tienes al menos el armazón de un hecho.

> **POR QUÉ CUATRO Y NO SEIS**
>
> La introducción hablaba de seis preguntas infantiles. Aquí aislamos las cuatro que fijan el escenario básico de un suceso. *Cuánto*, *cuál* y *cómo* llegan en los capítulos siguientes, cuando estos cuatro se queden cortos.

En este capítulo recorreremos los cuatro pilares uno a uno. Todavía no como ejes matemáticos formales (eso vendrá cuando construyamos la arquitectura en la Parte III), sino como **preguntas con personalidad propia**. Cada una esconde rarezas, decisiones de diseño y convenciones que solemos pasar por alto. Comprenderlas a fondo ahora nos ahorrará incontables dolores de cabeza cuando empecemos a modelar bases de datos de verdad.

Para que el recorrido sea concreto, cruzaremos cada pilar con cuatro escenas que nos acompañarán toda la sección: la **venta de una camiseta** en una tienda, un **gol de fútbol**, una **película** y una **ordenanza municipal**. Cuatro mundos sin relación aparente; verás que responden a las mismas cuatro preguntas con la misma facilidad.

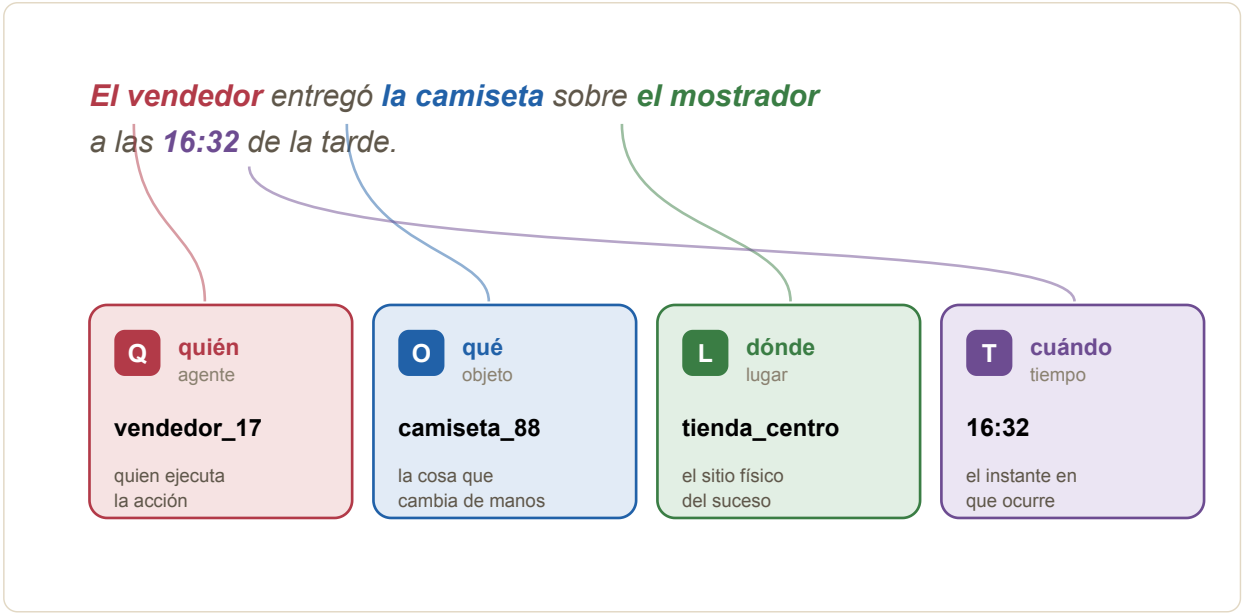


Figura 2.1. La misma operación que tu mente hace sola: cada fragmento de la oración aterriza en uno de los cuatro pilares. *El vendedor* al eje Q, *la camiseta* al O, *el mostrador* al L y *las 16:32* al T. Nadie los confunde, y ese acuerdo tácito es lo que vamos a volver arquitectura.

Q — Quién: la pregunta por el agente

A primera vista, *quién* parece el más inofensivo de los cuatro. Quieres saber quién hizo algo, quién recibió la acción o sobre quién recae la responsabilidad. La respuesta canónica que esperas es **un agente capaz de ejecutar una acción**: una persona, un grupo o una organización formal.

Cruzado con nuestras cuatro escenas, el panorama es plácido:

- Q 👕 **Venta**: ¿Quién despachó la camiseta? El vendedor, `vendedor_17`.
- Q ⚽ **Fútbol**: ¿Quién marcó el gol? El delantero, `messi` (y, con más detalle, quién dio la asistencia: `di_maria`).
- Q 🎬 **Cine**: ¿Quién dirigió la película? La directora, `serra`.
- Q 🏛️ **Ordenanza**: ¿Quién la promulgó? La `municipalidad_centro`, por mano del `alcalde_reyes`.

Hasta aquí, ninguna fricción. Pero la pregunta se vuelve fascinante apenas la sacamos de su zona de confort y la confrontamos con la realidad de los datos.

Un agente, varios agentes, agentes en disputa

Piensa en aquel gol. Lo hemos atribuido a `messi`, que remató, y a `di_maria`, que asistió. Pero «¿quién hizo la jugada?» admite respuestas en capas: quien remató, quien dio el pase, quien arrastró a los defensas para abrir el hueco. Y se complica más cuando el balón rebota en un defensor rival y entra al arco: las estadísticas oficiales hablan de «gol en contra» y, para la tabla de goleadores, le acreditan el tanto al delantero que pateó, no al defensor que lo desvió. El reglamento zanja la disputa; pero traducir esa decisión a una estructura de datos («este evento tiene *un* agente principal y *varios* agentes secundarios con roles distintos») está lejos de ser trivial.

El cine lo lleva al extremo. ¿Quién «hizo» `pelicula_marea`? La directora `serra` y la guionista `haddad` responden ambas a «¿quién?», pero con roles legalmente distintos: dirección y guion no son lo mismo, ni siquiera ocurren en el mismo momento. El problema empieza cuando un sistema mete a todos en un único campo genérico llamado `autor`. Cuando esa base de datos intenta compartir la información con otra, si nadie sabe qué convención interna usó cada una, los datos chocan: exactamente el conflicto de la torre de Babel.

AUTOR VS. EJECUTOR

> La política de la venta (precio de lista, descuentos, impuesto aplicable) pudo fijarla la gerencia hace meses; quien la *ejecuta* esta tarde es `vendedor_17`. Ambos responden a «¿quién?», pero uno aporta la regla y el otro el acto. Confundirlos es un clásico de los sistemas heredados.

Y aún hay una distinción más fina, de naturaleza temporal. ¿Quién «hace» la venta que tienes delante? `vendedor_17` la cerró esta tarde, pero las condiciones exactas (precio de 49,90 dólares, 18 % de impuesto) las fijó alguien más, tiempo atrás. Hay una diferencia de fondo entre el *autor* (quien diseñó la regla) y el *ejecutor* (quien la materializa hoy). Las dos personas son respuestas legítimas a «¿quién?», pero pertenecen a momentos y a naturalezas de acción distintas.

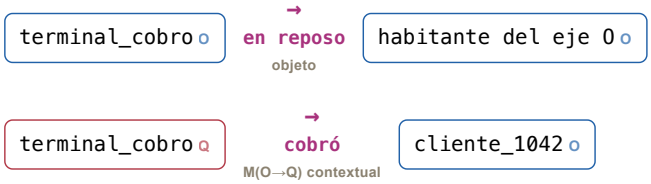
El caso límite: cuando una cosa actúa

Y aquí aparece un caso que merece tratarse con cuidado. ¿Qué ocurre con el *quién* cuando el que ejecuta la acción no es una persona ni una empresa, sino un objeto inanimado? La terminal de cobro que aplica sola el descuento al pasar la camiseta; el sistema VAR que interrumpe el partido para anular un gol; el motor de recomendación que decide sugerirte cierta película. Al hablar, les concedemos agencia sin titubear: *la terminal cobró el impuesto*, *el VAR anuló la jugada*, *el algoritmo eligió*.

Sin embargo, a nivel de arquitectura de la información, estos objetos no pertenecen por naturaleza al eje *quién*; su domicilio de origen es el eje *qué*, porque son cosas o sistemas. Lo que sucede en estos casos es que, **en situaciones muy específicas, un objeto asume temporalmente la capacidad de agencia**.

> ADELANTO · AGENCIA CONTEXTUAL
>
> La **agencia** (la capacidad de hacer algo) no es una propiedad estática que un objeto posea siempre, sino una **propiedad puramente contextual**. Según la situación, un objeto pasivo puede «vestirse» de agente y ocupar transitoriamente el eje Q, sin dejar de ser, en su origen, un habitante del eje O.
>
> A este principio lo llamaremos **agencia contextual**, y volverá como una de nuestras decisiones de diseño más fuertes (la formalizaremos como la regla **D5** en el capítulo 9). Por ahora basta con dejar el nombre puesto: una cosa puede actuar, pero solo dentro del marco que se lo permite.

Esta observación, en apariencia menor, tiene un peso enorme en el diseño de software. Decide si tu modelo puede registrar honestamente frases tan cotidianas como «el sistema rechazó el pago» sin mentir sobre la naturaleza del sistema. Mira la misma terminal de cobro bajo sus dos lecturas: en reposo es un objeto del eje O; pero en el instante en que aplica el impuesto sola, ese mismo objeto ocupa, transitoriamente, el eje Q:



El identificador no cambió (sigue siendo la misma terminal); lo que cambió fue el eje que ocupa, y solo mientras dura la acción. Volveremos sobre ello; por ahora, sigamos hacia la pregunta más resbaladiza de todas.

O — Qué: la pregunta por la cosa

El *qué* es profundamente engañoso porque la gramática parece exigirle una sola respuesta concreta, pero en la práctica abre la puerta a entidades de tipos irreconciliables. Si preguntas «¿Qué pasó?», esperas un evento. Si preguntas «¿Qué vendió el vendedor?», buscas un objeto material. Si preguntas «¿Qué situación atraviesa el equipo?», pides un contexto complejo. Las tres son gramaticalmente correctas, las tres aterrizan en el eje *qué*, y las tres apuntan a estructuras de datos que no se parecen en nada.

Para no perder la cordura adoptaremos un criterio funcional y deliberadamente amplio: **el eje *qué* aloja absolutamente todo lo que no sea un agente, un lugar o un momento**. Y, al hacerlo, hay que reconocer que dentro de ese gran contenedor conviven al

menos tres familias muy distintas.

① OBJETOS

Cosas (tangibles o no) que existen por sí mismas, persisten en el tiempo y pueden mencionarse en muchas situaciones. La `camiseta_88` que despachó el vendedor; el balón del partido; el guion de `pelicula_marea`; el texto publicado de la `ordenanza_142`.

② EVENTOS

Hechos que ocurren, con un inicio y un fin claros; la materia prima de las noticias. La `venta_001`; el gol `gol_001`; el rodaje de `escena_42`; el acto de promulgar la ordenanza.

③ SITUACIONES

Marcos amplios y duraderos dentro de los cuales ocurren muchos eventos pequeños interconectados. La jornada entera de la tienda; el `partido_arg_per_2026` completo; el rodaje entero de la película; el período de vigencia de la ordenanza.

Aquí asoma una intuición de diseño sutil pero crítica: estas tres familias no están separadas por muros impenetrables. Un evento se transforma en objeto cuando otro dato lo menciona («**ese gol** fue revisado por el VAR»); ahí, la jugada deja de *ocurrir* y pasa a ser una cosa a la que apuntamos. Y, en sentido inverso, una situación gigantesca puede verse como un evento simple desde un nivel aún mayor («**el partido** se suspendió por un corte de luz»). El eje *qué* tiene una flexibilidad inherente respecto de la escala: lo que en un nivel del modelo es un evento activo, en un nivel superior es un objeto pasivo al que se hace referencia.

Esa flexibilidad es una ventaja arquitectónica enorme. Permite tratar de forma uniforme cosas que la programación tradicional se ve obligada a separar en tablas distintas y difíciles de cruzar. Pero —cuidado— también es una trampa mortal.

TRAMPA: APLANAR UN EVENTO HASTA VOLVERLO UN OBJETO MUDO

La maleabilidad entre objeto, evento y situación es poderosa, y por eso es peligrosa. Reducir un evento rico (con sus agentes, su instante, sus magnitudes) a una simple cadena de texto destruye información que luego nadie podrá recuperar. El proceso disciplinado de «cosificar» un evento sin perder su contenido tiene nombre técnico: **reificación**, y lo desmenuzaremos en la Parte III. Por ahora, basta con desconfiar de todo campo que guarde «lo que pasó» como una frase suelta.

Asimilemos la intuición con nuestras cuatro escenas. Verás que el eje *qué* aparece tres veces en cada una, operando en registros distintos sin romperse:

El mismo eje **O** en tres registros, sobre cuatro dominios.

ESCENA	OBJETO	EVENTO	SITUACIÓN
👕 **Venta**	la camiseta	la venta cerrada	la jornada de la tienda
⚽ **Fútbol**	el balón	el gol marcado	los noventa minutos de partido
🎬 **Cine**	el guion	el rodaje de una escena	la producción entera
🏛️ **Ordenanza**	el texto publicado	el acto de promulgación	el período en que rige

En cada fila usamos el mismo eje *qué* tres veces, en tres niveles. El modelo no se confunde ni se quiebra: simplemente aloja cada cosa en el registro que le corresponde.

L — Dónde: la pregunta por el lugar

El *dónde* suele ser el más físico y concreto de los cuatro, y precisamente por esa aparente simpleza es el que más a menudo los programadores dan por sentado. Pero detrás de él se esconde una decisión semántica que, si no se toma de forma explícita, arruina cualquier cruce de información.

En el habla cotidiana manejamos dos «dónde» radicalmente distintos. El primero es el **lugar físico estricto**: una coordenada, una dirección, una ciudad, un recinto. El mostrador de la tienda, el césped del estadio, el plató donde se rodó `escena_42`, el despacho del alcalde. El segundo es el **lugar organizacional o administrativo**: un departamento, un ministerio, una franquicia, un club. La `gerencia_transito`, la `municipalidad_centro`, la selección a la que pertenece `messi`.

Ambos responden con total naturalidad a «¿dónde?», pero técnicamente apuntan a realidades de estructura incompatible. Una ciudad es un polígono geográfico; una gerencia es un organigrama de personas e intenciones. Si los mezclamos en el diseño (si guardamos `plaza_central` y `gerencia_transito` en una misma columna genérica llamada `ubicacion`) perdemos para siempre la capacidad de separar *dónde ocurre algo físicamente* de *a qué estructura administrativa pertenece*.

> **LA REGLA DEL LUGAR**
>
> Explicita siempre la **naturaleza** del lugar. El eje L se reserva de forma estricta para los lugares físicos. Una organización, cuando *ejecuta* una acción, se traslada al eje Q y actúa como agente (ecos de la agencia contextual); cuando funciona como mero contenedor administrativo, puede aparecer en L, pero etiquetada de forma obligatoria con una marca de «organización». La distinción se volverá nítida cuando veamos el código; por ahora, basta con tenerla mapeada.

Veamos cómo conviven los lugares en nuestras escenas —y nota que, dentro de un mismo hecho, varios «dónde» de escalas distintas coexisten sin estorbarse:

- L 👕 **Venta**: ¿Dónde se cobró? En el mostrador. ¿Dónde está la tienda? En la `tienda_centro`. ¿En qué ciudad? En el casco urbano entero. Tres escalas físicas conviviendo.
- L ⚽ **Fútbol**: ¿Dónde se remató el gol? Desde el área chica. ¿Dónde se jugó el partido? En un estadio, dentro de una ciudad. Escalas anidadas.
- L 🎬 **Cine**: ¿Dónde se rodó la escena? En un plató de la ciudad. ¿Dónde transcurre *dentro* de la ficción? En un puerto imaginario. El lugar real y el lugar narrado no coinciden.
- L 🏛️ **Ordenanza**: ¿Dónde se firmó? En el despacho municipal (lugar físico). ¿Dónde tiene jurisdicción? En todo el distrito (lugar político).

Conviene dejar asentada una observación de fondo: el pilar *dónde* está diseñado por naturaleza para soportar **jerarquías y anidamientos**. El mostrador está dentro de la tienda, la tienda pertenece a un barrio, el barrio se inscribe en la ciudad y la ciudad en el país. A nivel lógico, las cuatro respuestas son correctas para «¿dónde ocurrió?» aplicada al *mismo* hecho; todo depende del nivel de zoom que pida la consulta. El modelo que construimos tendrá que navegar esas escalas hacia arriba y hacia abajo sin generar ambigüedad.

T — Cuándo: la pregunta por el tiempo

Llegamos al último de los cuatro pilares base, y al que solemos tratar con más ligereza, dando por hecho que se resuelve con un formato de fecha (`AAAA-MM-DD`). La realidad es que el *cuándo* es el eje más rico y lleno de sutilezas lógicas de todo el modelo. El lenguaje natural tiene muchísimas maneras de expresar el tiempo, y las fechas de calendario son apenas una de ellas.

Volvamos a la tarde de la venta. Si alguien la cuenta diciendo «la camiseta la vendió *hace un rato*», ¿cómo lo guardamos? Si el sistema almacena solo el dato frío `2026-05-14T16:32`, destruye información: se pierde que la acción se vivió como reciente respecto del momento en que se habló. Pero si guarda literalmente la cadena «hace un rato», el motor se vuelve inútil, porque no puede calcular antigüedades ni ordenar eventos. La salida razonable es guardar la marca absoluta para el cómputo y, en una capa complementaria, conservar la expresión relativa como contexto asociado.

El problema se ahonda apenas salimos del calendario. En la música, una entrada no se sitúa «a las 20:03», sino «en el **compás 17**»: el tiempo no lo dicta un reloj de cuarzo, sino un compás interno. En el análisis de una película, una revelación ocurre «al **final del segundo acto**»: tiempo narrativo, medido por la posición dentro del relato. En una historia clínica, una crisis aparece «**tres horas después** de la última dosis»: tiempo relativo, anclado a un evento previo. Y en lo jurídico, la `ordenanza_142` entra en vigor «**a los treinta días** de su publicación»: tiempo derivado por una regla.

Si atendemos al uso real de la información, descubrimos que *cuándo* debe soportar, como mínimo, cinco tipos de tiempo:

1 · TIEMPO ABSOLUTO

Fechas y horas exactas atadas al calendario gregoriano. El día en que se rodó la película; las 16:32 de la venta.

2 · TIEMPO RELATIVO

Expresiones relacionales («antes de», «durante», «inmediatamente después de») atadas a otro hecho comprobable.

3 · TIEMPO DE RELOJ CORTO

Marcadores internos de un evento cerrado: el minuto 87 del partido, el tiempo de descuento.

4 · TIEMPO CÍCLICO

Patrones de repetición: cada lunes, el primer día de cada mes, la apertura diaria de la tienda.

5 · TIEMPO NO-RELOJ

Escalas abstractas sin relación con el cronómetro: el compás de una partitura, la página de un libro, el «final del segundo acto», el paso número cuatro de un manual.

Un eje *cuándo* robusto tiene que aceptar los cinco formatos y, más importante aún, reconocer cuál se está usando en cada transacción. La estrategia que adoptaremos formalmente más adelante es la **pluralidad de tiempos**: no forzaremos toda la información a pasar por un único reloj universal, sino que el eje funcionará como un espacio donde conviven múltiples escalas temporales, todas válidas, cada una bajo su propio sistema de coordenadas.

Cómo ordenar sin aplastar: reificar el instante

Esa pluralidad plantea de inmediato una pregunta operativa: si conviven varios relojes, ¿cómo ordena el sistema los eventos sin reducirlos a uno solo? La regla de normalización es deliberadamente humilde: **no traducir**. El compás 17 no se convierte a un horario de cuarzo (eso destruiría su naturaleza); en su lugar, cada instante no-reloj se reifica con tres datos estándar.

TRIPLETAS

```
# un instante musical, reificado en tres datos (no se "traduce" a reloj)
(entrada_coro, escala, "musical")        ∈ M(0, K)   # a qué sistema temporal pertenece
(entrada_coro, posicion, 17)             ∈ M(0, N)   # ordinal: ordenable DENTRO de su escala
(entrada_coro, valor_nativo, "compás 17") ∈ M(0, 0)   # la expresión original, intacta

# y cuando el mundo provee un puente real, se declara como hecho explícito:
(entrada_coro, sonó_en, 2026-05-14T20:03:14) ∈ M(0, T)  # no una conversión inventada
```

- `escala`: a qué sistema temporal pertenece (`musical`, `narrativo`, `paginado`, `reloj_absoluto`). Impide comparar peras con manzanas: el motor nunca ordenará un compás contra una fecha por accidente.
- `posicion`: un ordinal numérico que lo vuelve **ordenable dentro de su propia escala** (compás 17 → `17`; página 240 → `240`; «final del segundo acto» → `acto 2, 1.0`). Es la clave de orden.
- `valor_nativo`: la expresión original, preservada tal cual («compás 17», «p. 240», «final del segundo acto»), para no perder fidelidad.

Con esto, ordenar es trivial *dentro* de una escala (se ordena por `posicion` filtrando por `escala`) y honesto *entre* escalas: no se mezclan, salvo que exista un puente real. Y los puentes existen cuando el mundo los provee: la grabación de un concierto ata el `compás 17` de la escala musical a las `20:03:14` de la escala absoluta mediante un hecho explícito, no una conversión inventada. El modelo conserva la naturaleza del dato y, a la vez, gana la capacidad de secuenciar que el negocio necesita.

> *El tiempo no es un único río que todo lo arrastra, sino un delta de cauces que rara vez se tocan; el modelo no los funde, los deja correr en paralelo.*
>
> LA PLURALIDAD DE TIEMPOS

Lo que era cierto y dejó de serlo

> **DOS NOMBRES, UN FENÓMENO**
> La teoría de bases de datos lo llama **bitemporalidad**; la lingüística estructural, **vigencia**. Ambos describen lo mismo: un dato verdadero dentro de una ventana de tiempo y falso fuera de ella.

A todo lo anterior se suma una complicación fascinante que solo dejaremos planteada. Un dato puede ser plenamente **válido** durante un período y dejar de serlo en el siguiente. La tienda abrió en la `plaza_central` entre 2019 y 2025; si preguntas hoy dónde está, la respuesta es otra dirección. «¿Dónde queda la tienda?» no es un bloque de piedra: es un flujo de valores, cada uno con su propia fecha de caducidad. Ese fenómeno (un dato verdadero solo dentro de una ventana temporal) será un componente explícito de nuestro diseño final.

Cerremos el pilar aplicándolo a las cuatro escenas:

- T👕 **Venta**: Tiempo absoluto (se cobró a las 16:32), cíclico (la tienda abre cada día) y relativo interno («aplicar el impuesto una vez confirmado el pago»).
- T⚽ **Fútbol**: El gol cae en el minuto 87: tiempo de reloj corto, que solo cobra sentido respecto del inicio del partido.
- T🎬 **Cine**: La revelación ocurre «al final del segundo acto» (tiempo narrativo); el rodaje de esa escena tiene una fecha de calendario (tiempo absoluto). Dos líneas conviviendo en paralelo.
- T🏛️ **Ordenanza**: Se firmó ayer (tiempo absoluto); entra en vigor a los treinta días (tiempo derivado por regla). Dos relojes operando sobre el mismo objeto.

Cada uno de estos cuatro pilares es, además, un eje por el que cortar el conjunto. Fijar el «cuándo» en las 16:32 y dejar libre el «quién» ya es pedir *todos los que vendieron a esa hora*; fijar el «dónde» en la `tienda_centro` y soltar el «qué» es pedir *todo lo que se despachó allí*. Las cuatro preguntas no describen solo una venta: rebanan miles a la vez.

Cuando un pilar se confunde: una IA en libertad

Antes de cerrar el capítulo conceptual, vale la pena diseccionar un caso real. Es la clase de anécdota que circula por los foros de tecnología y que, por lo común del fallo, se ha vuelto casi un ejemplo de manual sobre los límites de la inteligencia artificial sin estructura.

Imagina a alguien que sale del trabajo con la bicicleta pinchada. Saca el teléfono y le dicta a su asistente:

> *Tengo la bici con una rueda reventada y la quiero llevar al taller de reparación, que me queda a tres cuadras de aquí. ¿Me conviene ir caminando o en bici?*

La respuesta del modelo fue desconcertante:

> *Si te queda tan cerca, ve caminando sin problema. Son apenas tres cuadras, es más saludable y de paso no gastas energía pedaleando.*

El consejo es un disparate físico: no se lleva una bicicleta al taller yendo a pie (la bicicleta *es* lo que hay que transportar). Pero el error no nace de una «estupidez» del algoritmo, sino de un problema muy concreto: una **mala asignación de roles a los cuatro pilares**. La estadística llenó las cajas, pero las llenó mal.

Una lectura humana y estructuralmente correcta de la escena reparte así las variables:

LECTURA CORRECTA

```
Q (quién)                 : la persona           (agente)
O (qué, lo que se mueve)   : la bicicleta         (paciente)   ← el centro de la escena
L (dónde, destino)         : el taller, a 3 cuadras
T (cuándo)                 : ahora
M (con_finalidad)          : reparar la bicicleta            ← le da sentido a todo
```

El verbo núcleo de la situación es *llevar*. La persona es, sin duda, el agente; pero el **paciente** (la cosa física que sufre el transporte) es la bicicleta. Y la finalidad ineludible de la maniobra es someter esa bicicleta a una reparación. Si quitas la finalidad de la ecuación, la pregunta «¿caminando o en bici?» pierde todo su sentido.

¿Qué entendió, en cambio, el modelo basado en pura correlación estadística? Su reparto de roles fue, con toda probabilidad, este:

LECTURA DEL MODELO (ERRÓNEA)

```
Q (quién)                 : la persona           (agente)
O (qué, lo que se mueve)   : la persona           (¡la misma!) ← la bicicleta desapareció
L (dónde, destino)         : el taller, a 3 cuadras
T (cuándo)                 : ahora
M (con_finalidad)          : trasladarse al taller           ← finalidad real, perdida
```

Bajo esa lectura, la bicicleta se cae de su rol legítimo de paciente y queda empujada a un rol de **instrumento opcional** («puedes ir en bici o no»). La finalidad real (repararla) se diluye y la sustituye el mero acto de llegar a unas coordenadas. Como el modelo evalúa que la distancia es mínima, sus pesos probabilísticos concluyen que caminar es lo más eficiente y saludable. Impecable… salvo que el objeto central de la frase se evaporó.

EL FALLO NO ES DE CÁLCULO: ES DE MODELADO DEL MUNDO

El sistema perdió el rastro de una asignación estructural básica: en este contexto, la bicicleta **es** el objeto central del eje O, no un accesorio. Apenas esa pieza se cae del tablero, toda la cadena de razonamiento posterior se derrumba —y el modelo ni siquiera percibe el error.

Es aquí donde la promesa central de este libro cobra fuerza industrial. Si pudiéramos entregarles los hechos del mundo a los asistentes de IA con una estructura donde los cuatro pilares fueran **explícitos** (el agente siempre en Q, el paciente en O, el lugar en L y el momento en T), esta clase de absurdos desaparecería. Y no porque hubiéramos fabricado un algoritmo más listo, sino porque la propia **arquitectura impediría la asignación incorrecta**.

Un sistema con bases firmes «sabe», por diseño, que un verbo transitivo como *llevar* exige obligatoriamente un paciente, y que ese paciente debe registrarse en el eje O. Si el paciente no está, el sistema detecta que la orden está incompleta y, en lugar de alucinar una respuesta, se detiene a pedir una aclaración.

La estadística deja de gobernar a ciegas; la estructura toma el control y le pone límites seguros. Las cajas vacías se vuelven preguntas, no invenciones. Esto (dar al modelo un esqueleto que no se puede llenar mal) es justo lo que la industria de la IA necesita con más urgencia hoy. Y volveremos a ello en la Parte VI.

Los cuatro pilares y lo que falta

Si repasamos las cuatro preguntas (*quién*, *qué*, *dónde*, *cuándo*) y volvemos a la tarde de la venta, las cuatro piezas están ahí, haciendo un trabajo de ordenamiento preciso. El vendedor (quién) entregó la camiseta (qué) sobre el mostrador (dónde) a las 16:32 (cuándo). La escena queda sólidamente descrita.

¿Bastan para modelar la realidad entera? Para una frase así, sí. Pero no transcurrirá mucho tiempo en un entorno de producción antes de toparnos con registros donde estas cuatro cajas se quedan cortas. Hagamos crecer un poco la escena:

> *El vendedor despachó, en plena liquidación de temporada, una camiseta deportiva que costó cuarenta y nueve dólares con noventa.*

El hecho base es el mismo, pero el caudal de información acaba de escalar. De pronto hay que procesar un **precio** (49,90). Entró en juego un **motivo** claro de la acción (la liquidación de temporada). Aparece una **clasificación más fina** del objeto: no es una prenda cualquiera, es una *camiseta deportiva*. Y, de paso, la frase deja entrever que la camiseta pertenece a la categoría *prenda de vestir*: un detalle que nuestros cuatro pilares todavía no permiten declarar, porque el eje Q guarda individuos concretos, no define categorías abstractas.

Conceptos como *«prenda de vestir»*, *«camiseta deportiva»* o *«dólar»* no son cosas palpables. Son **clases**, arquetipos. Y, por definición, no encajan limpiamente en ninguno de los cuatro pilares que acabamos de estudiar.

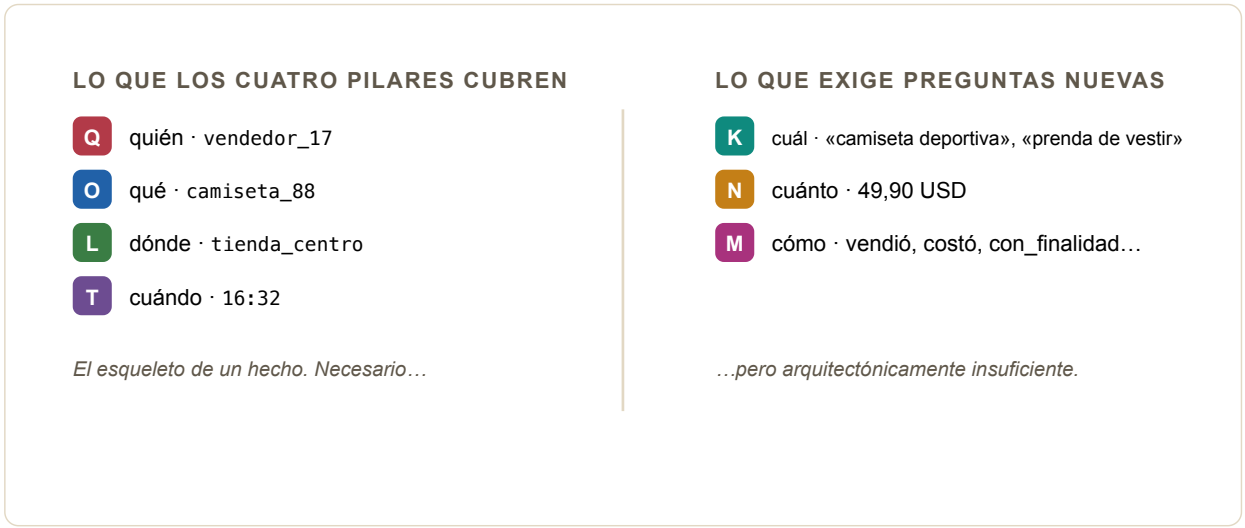


Figura 2.2. Los cuatro pilares levantan el esqueleto de cualquier hecho (son *necesarios*), pero la frase enriquecida revela tres huecos que ninguno de ellos cubre: la **clase** del objeto (K), su **magnitud** (N) y las **conexiones** que lo amarran todo (M). Necesarios, pero insuficientes.

IDEA CLAVE

Los cuatro pilares son **estrictamente necesarios, pero arquitectónicamente insuficientes**. Describen *dónde* y *cuándo* actúa *quién* sobre *qué*; no saben decir de qué *tipo* es cada cosa, *cuánto* mide, ni *cómo* se enlaza con el resto.

De ahí emerge el resto del modelo. Necesitamos, como mínimo, una quinta pregunta (*cuál*, es decir, qué tipo de cosa es esto) con eje propio: el zócalo categórico donde habitarán los tipos, las familias y los conceptos abstractos. A ella hay que sumarle una sexta (*cuánto*) para las métricas, con sus propias trampas de conversión de unidades; y una séptima (*cómo*) encargada de articular las conexiones y atributos que amarran todo el modelo.

De esas tres, la primera y más delicada de todo el sistema es el **eje categórico**. Es, paradójicamente, el que soporta la mayor carga de trabajo en los sistemas del mundo real, y es allí donde la arquitectura de WQuestions se integra pacíficamente con los gigantes de la industria (Schema.org[30], QUDT[18], SNOMED, CIDOC CRM[4]), usándolos como aliados en lugar de competir con

ellos: el vocabulario común que nos asegura no volver a levantar, nunca más, la torre de Babel de los datos. A ese zócalo (el eje K, *cuál*) dedicamos el capítulo siguiente.

03

Cuál: el zócalo categórico (K)

Los cuatro pilares saben dónde guardar a un vendedor y a la camiseta que vende, pero no saben decir qué son. Falta un eje para los tipos, las unidades y los estados: el segundo gran zócalo del modelo.

El vendedor pone delante de ti, sobre el mostrador, una prenda doblada de tela suave, con el escudo bordado al pecho. Sabes muchas cosas concretas de *esa* prenda: que te la mostró el vendedor, que está sobre el mostrador, que cuesta 49,90 dólares. Pero hay otra pregunta que tu mente responde antes incluso de tocarla, y que ninguno de los cuatro pilares del capítulo anterior puede contestar: **¿qué es esto?** Es una camiseta. No *esa* camiseta en particular (la que tienes enfrente), sino un ejemplar de una categoría que existía mucho antes de que llegara a la tienda y seguirá existiendo cuando la prenda esté guardada en tu armario.

Esa pregunta (*¿de qué tipo es?*) parece menor, casi una formalidad. No lo es. Es la grieta por la que se cuela un eje entero, y con él la mitad del poder de razonamiento de cualquier sistema serio. Este capítulo trata de ese eje: **K**, el zócalo categórico, la respuesta a la pregunta *cuál*.

> **DÓNDE ENCAJA**
> El capítulo anterior levantó el esqueleto con cuatro pilares concretos. Este añade el primer eje *abstracto*. El siguiente mostrará por qué los números no pueden existir sin él.

Lo que los pilares dejan sin decir

Conviene volver un momento a la escena con la que cerró el capítulo dos. Teníamos un hecho (la venta que el vendedor cerró en el mostrador de la tienda) repartido limpiamente en cuatro casilleros: el vendedor en el Q, la camiseta en el O, la tienda en el L y el instante en el T. A primera vista, el hecho quedó descrito por completo. Pero si lo miras con ojos de ingeniero de datos, descubres un punto ciego enorme: hay información crítica que esos cuatro pilares, *por diseño*, no te dejan expresar.

Sabemos que el vendedor vive en el eje Q. Y como en la escena aparece también el cliente que compró la camiseta, sabemos que él vive igualmente en Q. Hasta ahí, sin problemas. Pero un sistema que aspire a ser útil necesita algo más profundo que un lugar donde guardar dos nombres: necesita saber *que ambos son personas*. Necesita la capacidad lógica de reconocer que `vendedor_17` y `cliente_1042` comparten una naturaleza (pertenecen a la misma **categoría**) para poder, por ejemplo, contar cuánta gente pasó por la tienda sin confundirla con las prendas.

Lo mismo ocurre con la camiseta. Ese objeto físico concreto, con su escudo bordado y su etiqueta de talla, vive correctamente en el eje O. Pero la palabra «camiseta» nombra algo mucho más amplio que ese ejemplar específico que descansa sobre el mostrador. «Camiseta» es **el tipo** de cosa al que pertenece esa prenda. Cualquier sistema con aspiraciones de inteligencia tiene que distinguir sin esfuerzo entre el ejemplar concreto (esta camiseta puntual, con su talla y su color) y la categoría universal a la que pertenece: la camiseta como concepto general, la prenda, el tipo de producto que se cataloga en cualquier tienda del mundo.

La limitación es estructural y vale la pena nombrarla con claridad: los pilares que hemos visto funcionan como un **inventario de individuos**. Catalogan a esta persona, a este objeto, a este lugar exacto, a este instante del reloj. Lo que falta es un inventario distinto, uno que defina **lo que esos individuos son**. Y esa definición de identidad no cabe en Q ni en O. Reclama su propio eje.

Qué es K, exactamente

La letra **K** se eligió por convergencia lingüística: *Kind* en inglés, *Klasse* en alemán, la raíz griega *katēgoría* que comparten las lenguas latinas. Es el eje que responde a la pregunta **¿cuál?**: cuál, de entre un conjunto acotado de tipos, estados o categorías, le corresponde a algo. Está diseñado en exclusiva para alojar **tipos, categorías y conceptos abstractos**. Es el territorio lógico donde viven ideas genéricas como `camiseta`, `prenda`, `largometraje`, `genero_drama`, `gol_jugada_abierta`, `ordenanza_municipal`, `kilogramo`, `infarto_agudo_de_miocardio` o `arquitectura_transformer`.

La imagen que mejor lo captura es la de un edificio sostenido por dos grandes fundaciones. Los pilares Q, O, L y T forman el **primer zócalo**, el de lo concreto. Allí viven las cosas que existen materialmente, entidades con identidad propia y, casi siempre, una ubicación en el espacio y en el tiempo. K es el **segundo zócalo**, el de lo categórico. Allí no hay nada que puedas tocar: hay nombres genéricos, patrones y moldes bajo los cuales agrupamos a los ejemplares del primer zócalo.

> **IDEA CLAVE**
>
> El modelo descansa sobre **dos zócalos**, no uno. El primero (Q, O, L, T) cataloga los individuos del mundo. El segundo (K) define las categorías a las que esos individuos pertenecen. Sin el segundo, un sistema puede guardar datos, pero no puede *razonar* sobre ellos.

Esta distinción entre el individuo y su categoría es sutil, pero es la herramienta más potente del modelo. Una *directora de cine* de carne y hueso vive en Q: es una persona, `serra`, con fecha de nacimiento, nacionalidad y una filmografía. Pero el concepto de *directora* (el rol, el oficio) vive en K. Un *partido de fútbol* concreto vive en O: `partido_arg_per_2026`, ocurrió un sábado, terminó 2 a 1. Pero el concepto de *partido de fútbol* vive en K. La ciudad donde se jugó vive en L; la categoría abstracta de *ciudad sede* vive en K.

Separar las instancias de sus categorías es justamente lo que le permite a un sistema **razonar con generalidad**. Es lo que hace posible que el buscador de una plataforma de streaming entienda la orden «muéstrame todos los largometrajes de género drama», en lugar de obligar al usuario a teclear, uno por uno, el título de cada película que cumple esa condición.

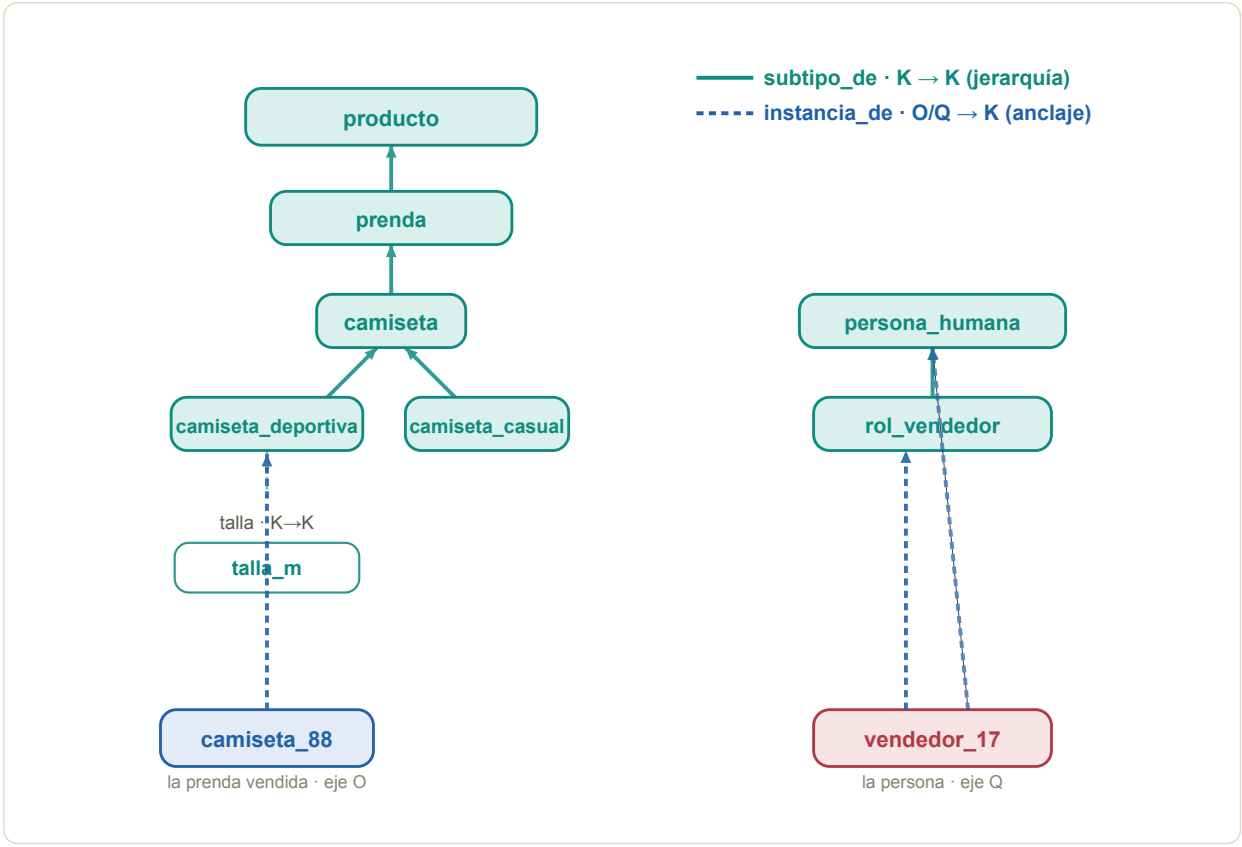


Figura 3.1. El eje K como red de conceptos. Las flechas **verdosas** son `subtipo_de` y arman la jerarquía categórica: `camiseta_deportiva` es subtipo de `camiseta`, que es subtipo de `prenda`. Las flechas **punteadas** son `instancia_de` y atan cada individuo concreto a sus tipos: `camiseta_88` (una prenda en O) y `vendedor_17` (una persona en Q). Las categorías también se conectan entre sí (`camiseta_deportiva` tiene una `talla_m`): K no es una lista, es un grafo.

Cuatro familias dentro de K

El eje K no es una bolsa plana donde arrojamos etiquetas al azar. Es un entorno muy estructurado que aloja, al menos, cuatro familias distintas de entidades categóricas. Vale la pena recorrerlas una a una para entender su alcance.

1 · Tipos de objetos y eventos

Conceptos como `camiseta`, `largometraje`, `gol_jugada_abierta`, `ordenanza_municipal` o `sesion_ia` son, en sentido estricto, tipos. La camiseta física está en O; el molde teórico `camiseta` está en K. Para coser ambos ejes usamos una relación fundamental que llamaremos `instancia_de`. Esa única tripleta es el cordón umbilical entre la prenda concreta y su concepto:

Esa misma forma se repite para cualquier individuo de cualquier pilar; lo que cambia es el eje de origen:

TRIPLETAS

```
(camiseta_88,    instancia_de, camiseta)            ∈ M(O, K)
(vendedor_17,    instancia_de, rol_vendedor)        ∈ M(Q, K)
(pelicula_marea, instancia_de, largometraje)        ∈ M(O, K)
(modelo_lumen_2026, instancia_de, modelo_de_lenguaje) ∈ M(O, K)
```

Cualquier individuo registrado en Q, O o L (y a veces en T) debe poder responder a la pregunta «¿de qué concepto eres instancia?». Ese hilo es lo que ata el grafo de hechos concretos con el diccionario abstracto del sistema. Sin él, los pilares serían un montón de nombres flotando sin red.

2 · Unidades de medida

Términos como `kilogramo`, `segundo`, `token`, `dolar`, `milisegundo` o `grado_celsius` son categorías puras; no son entidades materiales. Nunca te cruzarás por la calle con «un kilogramo físico» flotando en el aire como una botella; lo que experimentas es el patrón de «ser un kilogramo» aplicado a la masa de una botella. Como son conceptos teóricos para medir, todas las unidades habitan en K.

> **ADELANTO**
> No hay que inventar las unidades: existe un catálogo canónico, **QUDT**[18], que clasifica miles de ellas, define sus dimensiones físicas y permite conversiones. Lo estudiaremos a fondo en el capítulo del eje N.

Que las unidades vivan en K no es un capricho de catalogación: es una necesidad arquitectónica. Como veremos en el próximo capítulo, un número desnudo no significa nada («18» no es nada hasta que sabes que son *18 gramos de café molido*) y el único lugar donde puede habitar esa unidad es el zócalo categórico. Por eso K tenía que presentarse antes que N.

3 · Estados y valores enumerativos

En el diseño de software es muy común encontrar atributos que solo admiten un valor de una lista estricta y cerrada: *casado / soltero / viudo*, *aprobado / pendiente / rechazado*, *titular / suplente*, *encendido / apagado*. Esos valores predefinidos no son números ni deberían tratarse como texto libre: son **categorías de estado**. Y como son conceptos genéricos aplicables a muchos casos, viven en K.

TRIPLETAS

```
(gol_001,       pierna_ejecutora, pierna_zurda)      ∈ M(0, K)
(ordenanza_142, estado_tramite,    promulgada)        ∈ M(0, K)
(sesion_ia_5521, modo_entrega,     streaming)         ∈ M(0, K)
```

4 · Conceptos abstractos y nomenclaturas

Todo dominio profesional serio cuenta con diccionarios controlados. Aquí entran los diagnósticos médicos oficiales (CIE-10, SNOMED), las categorías comerciales y códigos de producto (SKU), los géneros cinematográficos y musicales, las tipologías de actos administrativos o las arquitecturas de inteligencia artificial. Cualquier nomenclatura controlada, sin excepción, aterriza en el eje K.

TRIPLETAS

```
(pelicula_marea, genero,       genero_drama)     ∈ M(0, K)
(camiseta_88,    sku,          "cam_dep_m_azul")  ∈ M(0, K)
(ordenanza_142,  tipo_norma,   ordenanza_municipal) ∈ M(0, K)
(modelo_lumen_2026, arquitectura, transformer)     ∈ M(0, K)
```

Si una línea atraviesa a las cuatro familias es esta: **todo lo que vive en K no es un ejemplar único del mundo, sino un patrón conceptual que puede aplicarse a decenas, miles o millones de casos**.

Una sola plantilla, todas sus instancias

Que la plantilla viva separada de sus ejemplares no es solo orden conceptual: es lo que hace barata una pregunta que en otros modelos sale cara. Si `camiseta` es una categoría en K, entonces «todas las prendas de ese tipo» ya está dicho —son los individuos que cuelgan de ella por `instancia_de`, y `camiseta_88` es solo uno entre ellos—. Preguntar por una unidad o por las diez mil cuesta lo mismo: en ambos casos recorres el mismo cable hacia la misma plantilla.

PYTHON

```
# Cuántos ejemplares instancian la misma plantilla de K
count(u, Pattern(type_constraint=u.ind("camiseta")))
```

La frontera resbaladiza: ¿instancia (O) o clase (K)?

El lector técnico choca tarde o temprano con un caso incómodo: ciertos objetos híbridos (un **VAR** en el fútbol, un **algoritmo de recomendación**, un **modelo de lenguaje**) parecen vivir en varios ejes a la vez. La confusión se disuelve apenas se entiende que **no es la cosa la que tiene un eje, sino cada uno de sus papeles**. Una misma palabra nombra cosas distintas según qué le preguntemos.

La misma palabra, distintos papeles, distintos ejes

SI TE REFIERES A…	EJE	VAR	ALGORITMO DE RECOMENDACIÓN
El **tipo o concepto** (la tecnología, la categoría)	K	`var` como clase de sistema arbitral	`filtrado_colaborativo` como técnica
El **ejemplar concreto** (un despliegue, una unidad)	O	el VAR instalado en el estadio Monumental	el motor de recomendación de una plataforma
Una **acción suya, reificada** (un evento que ejecutó)	O	*«revisó la jugada del minuto 87»*	*«recomendó la serie a Paredes»*
Cuando **actúa** (ocupa el rol `agente`)	Q	el VAR como agente de *anular_gol*	el motor como agente de *recomendar*

Figura 3.2. Un mismo objeto híbrido se reparte por varios ejes según el papel que cumpla en cada hecho. No se duplica: es un solo individuo, referido desde situaciones distintas.

Aquí se ve la **reificación** en movimiento, y su parentesco con la agencia contextual (la regla D5 que formalizaremos en el capítulo 9). El mismo VAR es un objeto pasivo en O cuando describimos su instalación; es el `agente` en Q cuando *anula* un gol; y ese «anular el gol» es, a su vez, una situación reificada que vive en O. La entidad **no se duplica** al cambiar de eje: es un solo individuo, referido desde situaciones distintas. Y un evento activo en un nivel (la revisión del VAR) puede volverse el objeto pasivo de otro («la revisión del minuto 87 fue impugnada por el club»): eso es, exactamente, reificar.

LA REGLA DE LOS TRES DEDOS

Ante cada mención, pregúntate *qué* se está nombrando:

- **¿De qué tipo es?** → K (la categoría).
- **¿Qué ejemplar, o qué evento, es?** → O (la instancia concreta o la acción reificada).
- **¿Quién actúa en esta situación?** → Q (el agente, por contexto, no por naturaleza).

La misma disciplina ordena a los demás híbridos del libro: `transformer` es la arquitectura (K), `modelo_lumen_2026` el modelo desplegado (O), *«el modelo resumió la consulta»* el evento (O), y el modelo en su papel de redactor del informe el agente (Q).

Por qué K necesita un eje propio

Hay una objeción legítima que cualquier ingeniero de bases de datos plantea al llegar aquí: *¿para qué crear todo un eje nuevo? ¿Por qué no tratar las categorías como simples cadenas de texto guardadas en una columna? Que el atributo sea* `estado_civil` *y el valor sea la palabra* `"casado"`*, y asunto resuelto.*

A corto plazo, parece pragmático. A escala de arquitectura empresarial, es la receta perfecta para la fragmentación de datos. La objeción del texto plano no se sostiene por tres razones técnicas que se acumulan hasta colapsar los sistemas tradicionales.

Razón 1 · Las categorías tienen estructura interna

La palabra `genero_drama` no es solo una secuencia de letras. Es un nodo semántico con una enorme riqueza de propiedades. Tiene equivalentes en otros idiomas (*drama*, *Drama*). Tiene dependencias lógicas con otras categorías (el drama es un tipo de *género narrativo*). Tiene contextos de aplicabilidad. Si guardamos «drama» como un simple *string* en una tabla SQL, toda esa estructura queda inaccesible para el motor: la palabra está «ciega». En cambio, si la tratamos como un individuo con derechos propios dentro de K, esa categoría puede tener sus propios hechos conectados:

TRIPLETAS

```
(genero_drama, etiqueta_en,  "drama")           ∈ M(K, K)
(genero_drama, subtipo_de,   genero_narrativo)  ∈ M(K, K)
(genero_drama, contrasta_con, genero_comedia)   ∈ M(K, K)
(genero_drama, uri_canonica, "schema:DramaSeries") ∈ M(K, K)
```

Visto así, K no es un archivo plano: es una **red de conceptos interconectados** (exactamente la red que dibujamos en la Figura 3.1). Los individuos de K son ciudadanos de primera clase del modelo. Esta capacidad de vincular conceptos entre sí será la clave cuando analicemos cómo WQuestions se integra con los diccionarios de otras industrias.

Razón 2 · El vocabulario serio exige autoridad externa

Las categorías de alto nivel que usan hospitales, bancos o gobiernos no se inventan en una lluvia de ideas de programadores; provienen de autoridades internacionales que las publican con identificadores únicos, los **URI**. **QUDT** regula las unidades; **Schema.org**[30] da los estándares del comercio web; **SNOMED** dicta los códigos médicos; la **ICAO** define las siglas de los aeropuertos. Tratar una de estas categorías como texto ignora por completo su peso legal e internacional. Al registrarla como individuo formal en K, podemos anexarle su URI canónica como atributo permanente:

TRIPLETAS

```
(qudt_miliseg,   uri_canonica, "http://qudt.org/vocab/unit/MilliSEC") ∈ M(K, K)
(snomed_infarto, uri_canonica, "http://snomed.info/id/22298006")     ∈ M(K, K)
(cie10_I21,      uri_canonica, "http://id.who.int/icd/release/10/2019/I21") ∈ M(K, K)
```

El caso del **CIE-10** (la *Clasificación Internacional de Enfermedades* en su décima edición, publicada por la Organización Mundial de la Salud) es el más nítido. Cuando un médico anota el código `I21` no escribe una palabra cualquiera: apunta a una entrada formal, internacional y traducida a decenas de idiomas, que significa «infarto agudo de miocardio». Si tu sistema guarda el diagnóstico como el texto libre «infarto», pierdes esa conexión: un hospital alemán escribirá «Herzinfarkt», uno brasileño «infarto do miocárdio», y nadie podrá cruzar datos. En cambio, si el sistema guarda la categoría como un individuo de K con su URI del CIE-10 anexada, todos los hospitales del mundo hablan el mismo idioma sin esfuerzo. La estadística global de salud, los estudios epidemiológicos, las alertas sanitarias internacionales: todo descansa sobre este mecanismo.

Razón 3 · Las consultas sobre categorías son el corazón del sistema

Las preguntas de negocio más valiosas siempre cruzan información categórica. Un analista no pide buscar un identificador concreto; pide *«todos los largometrajes de género drama estrenados después de 2024»*, o *«todas las sesiones del modelo Lumen con modo de entrega* `streaming` *y más de mil tokens de salida»*.

Si las categorías estuvieran guardadas como texto libre, estas búsquedas serían un campo minado: un usuario escribió «Drama», otro «drama», otro «Dram.». Al convertir las categorías en nodos estructurados de K, las consultas dejan de depender de la ortografía y pasan a depender de la matemática relacional. Se vuelven exactas, predecibles y combinables.

Dos relaciones canónicas: `instancia_de` y `subtipo_de`

Para que esta red de conceptos funcione y sea navegable, hay dos relaciones fundacionales que organizan la estructura interna de K. Es vital entender cómo operan.

La primera es `instancia_de`. Como adelantamos, es la relación de uso más intensivo en todo el modelo. Su trabajo exclusivo es actuar como puente, atando los individuos del mundo físico (los pilares) con sus definiciones conceptuales (el eje K). Y un punto crucial: un individuo puede ser instancia de varios conceptos a la vez.

TRIPLETAS

```
(messi, instancia_de, jugador_de_futbol)     ∈ M(Q, K)
(messi, instancia_de, capitan_de_seleccion)  ∈ M(Q, K)
(messi, instancia_de, persona_humana)        ∈ M(Q, K)
(messi, instancia_de, jugador_en_activo)     ∈ M(Q, K)
```

Estas cuatro asignaciones no compiten ni generan errores lógicos: coexisten en paralelo. Si el motor recibe una consulta filtrando por cualquiera de esas categorías, `messi` aparecerá como resultado válido.

La segunda relación es `subtipo_de`. A diferencia de la anterior, este conector *nunca* sale de las fronteras de K. Se usa en exclusiva para enlazar categorías entre sí, creando jerarquías y árboles de conocimiento (taxonomías):

TRIPLETAS

```
(jugador_de_futbol,   subtipo_de, atleta_profesional)           ∈ M(K, K)
(atleta_profesional,  subtipo_de, persona_humana)               ∈ M(K, K)
(modelo_de_lenguaje,  subtipo_de, modelo_de_aprendizaje_automatico) ∈ M(K, K)
(modelo_transformer,  subtipo_de, modelo_de_lenguaje)           ∈ M(K, K)
```

¿Por qué importa tanto a nivel tecnológico? Porque cuando declaras bien las reglas `instancia_de` y `subtipo_de`, le otorgas al sistema la capacidad de realizar **inferencias transitivas**. Si la máquina sabe que Messi es un `jugador_de_futbol`, y por otro lado sabe que todo jugador es un `atleta_profesional`, deduce sola que Messi es un atleta profesional. A esa habilidad se la conoce como **cierre transitivo** (*transitive closure*). Es el mecanismo de razonamiento más simple que existe, pero es el superpoder que evita que los programadores codifiquen a mano miles de reglas lógicas.

> *Una taxonomía bien declarada es razonamiento gratis: el sistema deduce lo que nunca le dijiste, solo porque supo encadenar «es un».*
>
> EL CIERRE TRANSITIVO, EN UNA FRASE

Lo que ya se intentó: tres puertas, ningún piso

Antes de ver cómo K aloja las ontologías del mundo, conviene entender por qué nadie lo había resuelto antes. No fue por falta de intentos: la industria abrió tres grandes puertas, y cada una se quedó a un paso.

PRECEDENTE · LA PRIMERA PUERTA: LAS 5W1H COMO EXTRACCIÓN

A fines del siglo XX, varios investigadores vieron las seis preguntas periodísticas como una herramienta para extraer datos de texto: un programa lee una noticia y acomoda las respuestas en casilleros (*quién: el alcalde; qué: promulgó una ordenanza; cuándo: ayer*). El resultado luce ordenado, pero estalla apenas se intenta **almacenarlo y cruzarlo**: para la máquina, «el alcalde» y «el titular del municipio» son dos cadenas distintas. Sin una capa de tipos ni un vocabulario canónico, las 5W1H son un buen *checklist* para no olvidar nada, pero no una arquitectura: les faltan justo las dos piezas que este capítulo introduce —una estructura de tipos (K) y un vocabulario oficial.

PRECEDENTE · LA SEGUNDA PUERTA: LA WEB SEMÁNTICA

En 2001, Tim Berners-Lee[31] propuso la **Web Semántica**, con **RDF**[8] como pieza maestra: toda la información reducida a tripletas *sujeto–predicado–objeto*. La idea es de una elegancia impecable y sostiene proyectos titánicos como Wikidata[32] y DBpedia[33]. Pero RDF resolvió la *sintaxis* y dejó libre la *semántica*: un sistema escribe `(serra, dirigio, pelicula_marea)`, otro `(serra, directora_de, …)`, un tercero `(serra, realizo, …)` —las tres correctas, las tres incompatibles—. Sin un diccionario mínimo común, la diversidad de lenguajes simplemente se mudó a otra capa.

PRECEDENTE · LA TERCERA PUERTA: LAS ONTOLOGÍAS DE DOMINIO

Ante ese caos, el tercer enfoque eligió el control estricto: reunir a los expertos de cada industria y publicar un vocabulario oficial y obligatorio. Así nacieron obras de arte de la ingeniería como **CIDOC CRM**[4] (patrimonio), **Biolink**[5] (biomedicina), **HL7 FHIR**[6] (historias clínicas) y **Schema.org** (web comercial). Cada una es excelente dentro de su perímetro. El problema asoma apenas hay que **vincular profundamente** ramas distintas: la película con la biografía de su directora (`Person`) y el lugar de rodaje (`Place`). Las ontologías crean los nodos, pero no estandarizan los cables entre ellos, y atarlos vuelve a recaer en código manual. Peor: como cada una se construyó aislada, todas tuvieron que modelar desde cero lo universal. Para decir «una persona», los museos usan `E21_Person`, la genética `biolink:Agent`, el comercio `Person` y las clínicas `Patient`: cuatro etiquetas incompatibles para un mismo ser humano físico.

A estas tres puertas se sumaron dos variantes que chocaron con el mismo muro: los **estándares de intercambio** (FHIR, EDI[20], ISO 20022[21]), que funcionan como un servicio de mensajería (traducen para el transporte, no unifican); y la **canonicalización a posteriori** (OpenIE[23], sistemas de limpieza de datos), que intenta reconciliar el caos *después* de que ocurrió, con un costo computacional que se vuelve inmanejable al cruzar decenas de sistemas.

El balance, llevado a cuatro dominios concretos, es elocuente:

DOMINIO	1 · 5W1H (HEURÍSTICA)	2 · RDF / WEB SEMÁNTICA (LIBRE CONEXIÓN)	3 · ONTOLOGÍA DE DOMINIO (DICCIONARIO ESTRICTO)
👕 **La camiseta**	Insuficiente: no entiende talla, color ni monto exacto de la venta.	Factible, si los programadores no usan verbos distintos para «vender» o «despachar».	Schema.org modela el producto, pero cuesta cruzarlo con datos del vendedor o de la tienda.
⚽ **El gol**	Útil para la crónica deportiva, inútil para armar estadística del partido.	Posible vía Wikidata, aunque los términos para describir la jugada varían entre bases.	Existe `SportsEvent`, pero no llega al detalle de con qué pierna se ejecutó el remate.
🎬 **La película**	Sirve para una nota *sobre* la película, no para modelar la obra.	Soportable, pero los vocabularios de cada base de cine chocan entre sí.	Modelos robustos en la industria, pero operan como burbujas difíciles de integrar.
🏛️ **La ordenanza**	Excelente: el dominio exacto para el que nació el modelo.	Funcional, pero sin acuerdo sobre cómo nombrar «promulgar» o «derogar», las búsquedas se arruinan.	`Legislation` estructura el contenedor, pero no entiende qué dice adentro.

Figura 3.3. Cada enfoque previo resuelve una parte y deja otra intacta: las 5W1H identifican las dimensiones pero no construyen arquitectura; RDF construye la conexión pero sin vocabulario base; las ontologías construyen vocabularios de lujo pero sin un piso común.

El patrón de las fallas es nítido. Todas levantaron **techos** espléndidos; ninguna construyó el **piso**. Y un piso es justo lo que aporta el eje K: una capa fundacional, por debajo de todos los vocabularios, lo bastante sutil para no estorbar la jerga de médicos o arquitectos y lo bastante firme como para que la información fluya sin traductores. Las preguntas cognitivas (que el capítulo 1 postuló como universales y que el capítulo 6 fundamentará en detalle) son ese piso. Veamos, entonces, cómo K abraza las ontologías existentes en lugar de competir con ellas.

K como zócalo para las ontologías existentes

Quizá la promesa de mayor impacto industrial del eje K es que **no obliga a ninguna empresa a reinventar su terminología**. Los diccionarios masivos que ya rigen en el mundo (Schema.org, QUDT, SNOMED, CIDOC CRM, Biolink) se mapean dentro de K como subconjuntos de datos, manteniendo intactas sus relaciones internas. WQuestions no llega a competir con estas herramientas ni a reemplazarlas: su objetivo explícito es **abrazarlas**.

Esta integración pacífica ocurre en tres niveles técnicos.

1 IMPORTAR URIS CANÓNICAS

En vez de transcribir un diccionario entero, el sistema aloja los conceptos esenciales y ancla a cada uno su URI internacional como validador de identidad.

2 MAPEAR DIALECTOS LOCALES

Cada organización define alias que apuntan al término oficial. La jerga interna sigue viva, pero por debajo todo señala al concepto canónico.

3 FEDERAR EQUIVALENCIAS

Cuando dos autoridades describen lo mismo con URIs distintas, K declara una equivalencia explícita y se vuelve la red de traducción entre ellas.

Nivel 1 · Importar URIs canónicas

En lugar de copiar un diccionario completo, el sistema aloja los conceptos que necesita y ancla a cada uno su URI internacional:

JSON

```
{
  "concepto": "infarto_agudo_de_miocardio",
  "eje": "K",
```

```
  "uri_snomed": "http://snomed.info/id/22298006",
  "uri_icd10": "I21",
  "etiqueta_es": "infarto agudo de miocardio",
  "etiqueta_en": "acute myocardial infarction"
}
```

El beneficio es inmediato: cuando el software de un laboratorio y el de un hospital referencian la misma URI, la sincronización de los historiales es perfecta, por más que una base esté en español y la otra en inglés.

Nivel 2 · Mapear el vocabulario de dominio

En la práctica, las empresas se resisten a usar términos canónicos largos o formales: prefieren su jerga. K lo resuelve permitiendo que cada organización defina alias que apunten al término oficial. Si en una clínica los médicos registran la enfermedad con las siglas «IAM», el sistema lo mapea así:

TRIPLETAS

```
(iam_clinica_norte, alias_de, infarto_agudo_de_miocardio)   ∈ M(K, K)
```

Con esa única instrucción, todas las búsquedas que el médico haga tecleando «IAM» apuntarán limpiamente al concepto global de SNOMED. Conviene adelantar que este principio sentará una decisión de diseño de fondo (**D9**, que veremos en los capítulos de lingüística computacional): la idea rectora de que **el usuario final nunca debe verse obligado a tocar etiquetas canónicas**. El usuario emplea su vocabulario natural; la capa estructural se encarga de la traducción matemática. K es la maquinaria que lo hace posible.

Nivel 3 · Federar conceptos equivalentes

El caos semántico alcanza su punto máximo cuando dos entidades internacionales describen exactamente la misma enfermedad pero publican URIs distintas. K resuelve el conflicto permitiendo declarar una equivalencia explícita:

TRIPLETAS

```
(snomed_infarto, equivalente_a, icd10_I21)   ∈ M(K, K)
```

Con el tiempo, a medida que el sistema se alimenta, K se transforma orgánicamente en una **red maestra de equivalencias** entre normas heterogéneas. Actúa como el puente de traducción universal que las ontologías de dominio nunca lograron —o nunca quisieron— construir entre ellas.

El enchufe: las ontologías ponen los nodos; WQuestions, los cables

El mecanismo no se limita a K. Una entidad de cualquier ontología externa se *enchufa* en el eje que le corresponde por naturaleza —una persona de CIDOC CRM (`E21_Person`) en Q, un evento de Schema.org en O, un lugar de GeoNames en L— conservando su URI canónica como ancla de identidad. Lo que ninguna de esas ontologías sabe hacer por sí sola (el problema que diseccionamos en «Tres puertas, ningún piso») es **vincular profundamente nodos de catálogos distintos**. Ahí entra WQuestions: las ontologías aportan los **nodos**; el eje M aporta los **cables** que cruzan de una a otra.

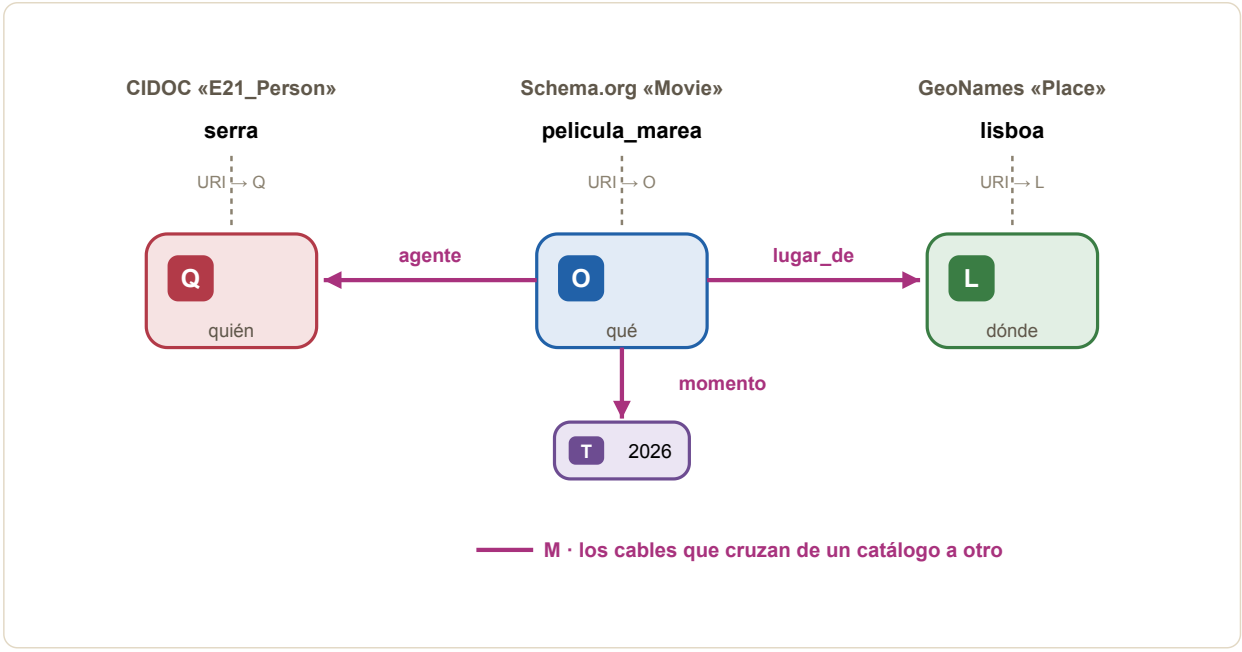


Figura 3.4. El hecho «*La película* Marea *fue dirigida por Serra (CIDOC) en Lisboa (GeoNames) en 2026*» queda como un solo grafo. Cada catálogo sigue siendo dueño de sus nodos y sus URIs; WQuestions solo añade la capa de predicados M que los estándares dejaron sin estandarizar —y con eso neutraliza la fragmentación—.

Cuatro dominios, cuatro vistas de K

Para terminar de materializar la función del eje categórico, resulta ilustrativo ver qué información específica almacena K en distintos sectores. Recorramos nuestros cuatro ejemplos y añadamos el entorno de la inteligencia artificial como caso de estudio.

👕 **La camiseta.** En K guardaríamos los tipos de prenda (`camiseta`, `pantalon`, `chaqueta`), los tipos de camiseta (`camiseta_deportiva`, `camiseta_casual`), las tallas (`talla_s`, `talla_m`, `talla_l`) y los colores. Buena parte de estos identificadores existe en Schema.org; el léxico de tejidos o materiales suele importarse de Wikidata.

⚽ **El gol.** El eje alojaría las taxonomías de los tipos de gol (`gol_jugada_abierta`, `gol_tiro_libre`, `gol_penal`, `gol_en_contra`), las partes del cuerpo utilizadas (`pierna_derecha`, `pierna_izquierda`, `cabeza`) y las zonificaciones del campo (`area_grande`, `fuera_del_area`). Parte emana de la FIFA; el resto es jerga analítica de las empresas de estadística deportiva.

🎬 **El cine.** Aquí vivirían los géneros (`genero_drama`, `genero_comedia`, `genero_documental`), los formatos (`largometraje`, `cortometraje`, `serie`) y los roles de autoría (`rol_direccion`, `rol_guion`, `rol_fotografia`). Schema.org y bases especializadas de cine aportan catálogos maduros, consumibles vía API.

🏛️ **La ordenanza.** K clasificaría la jerarquía institucional (`municipalidad`, `gerencia`, `concejo`), la tipología de actos administrativos (`ordenanza_municipal`, `decreto_de_alcaldia`, `resolucion`) y la escala de jurisdicciones (`nivel_nacional`, `nivel_regional`, `nivel_municipal`).

🤖 **La llamada a un modelo de lenguaje.** K ordenaría las arquitecturas subyacentes (`transformer`, `mamba`, `mixture_of_experts`), las familias comerciales, los tipos de tarea que pide el usuario (`resumir`, `clasificar`, `traducir`, `generar_codigo`) y la modalidad de entrega (`sincrono`, `streaming`, `batch`). Es un dominio fascinante porque su vocabulario está en plena expansión: cada laboratorio inventa sus propios términos comerciales, y K ofrece el sustrato donde todas esas convenciones podrán converger a medida que la industria madure.

D1 · La plantilla y la instancia

Para blindar la consistencia de la base de datos existe una regla estricta que gobierna la frontera entre el eje abstracto K y los ejes del mundo físico (en particular el eje O). Es un principio que se respeta sin excepciones y, además, es la **primera decisión de diseño que el libro registra formalmente**.

D1 **LA PLANTILLA Y LA INSTANCIA**

En el eje **K** habitan exclusivamente los conceptos **atemporales y categóricos** (las plantillas). En el eje **O** (y en el resto de los pilares) habitan las entidades **creadas, situadas geográficamente e instanciadas** (los objetos derivados).

SOBRE LA NUMERACIÓN DN

Las decisiones de diseño se numeran con **D** seguida de un número (D1, D2, D3…). Cada una se enuncia formalmente al aparecer por primera vez y luego se referencia por su número. Las irás encontrando en orden a medida que el modelo se construye.

La pregunta operativa que un desarrollador debe hacerse ante cualquier dato nuevo es directa: *¿este individuo tiene una fecha de creación concreta, una historia propia o una ubicación geográfica comprobable?* Si la respuesta es sí, su lugar es el eje O. Si carece de esas coordenadas, pertenece a K.

Observemos la diferencia en la práctica:

K (LA PLANTILLA ATEMPORAL) FRENTE A O (LA INSTANCIA SITUADA)

Pertenece a K

- El *modelo de camiseta deportiva*: una plantilla de producto replicable. No caduca ni tiene ubicación física.
- El concepto de *«modelo de lenguaje de arquitectura transformer»*: teoría informática pura.

Pertenece a O

- La *camiseta que el vendedor despachó el 14 de mayo*: se vendió a las 16:32, por 49,90 dólares, en la tienda del centro.
- La *versión ejecutable Lumen-2026-05, lanzada el 14 de mayo*: tiene fecha, consume energía, corre en servidores.

Esta dualidad produce un patrón arquitectónico muy robusto: existe siempre una **plantilla teórica anclada en K**, a partir de la cual se genera una **instancia concreta en O**, unidas por el cordón umbilical de `instancia_de`. El modelo de camiseta que diseña una marca vive abstractamente como tipo; pero cada vez que un vendedor despacha una unidad, el sistema crea un nuevo evento-instancia, con sus propios atributos: qué talla y color se vendió, a qué cliente, el horario exacto de la venta.

TRIPLETAS

```
# La plantilla teórica (atemporal) — vive en K
(camiseta) ∈ K
  tipo_camiseta : camiseta_deportiva
  talla_base    : talla_m
  color_base    : azul

# El evento instanciado (situado) — vive en O
(venta_001)  ∈ O
  instancia_de : venta
  objeto       : camiseta_88
  agente       : vendedor_17
  cliente      : cliente_1042
  monto        : 49.90 dolar
  cuando       : 2026-05-14T16:32:00
```

Esa danza constante entre la plantilla atemporal de K y la ocurrencia situada de O es uno de los patrones de modelado más poderosos en aplicaciones comerciales a gran escala. Lo veremos reaparecer una y otra vez cuando lleguemos a los casos prácticos de la Parte V.

Trampa de programación: la categoría guardada como texto

EL ATAJO QUE ENVENENA LA BASE

El atajo de todos los días: `estado = "promulgada"`, `pierna = "zurda"`, guardados como texto libre en una columna. Parece inofensivo hasta que el sistema crece: un operador escribe `"Zurda"`, otro `"zurda "` con un espacio, un tercero `"left foot"`, y la consulta «dame todos los goles de zurda» devuelve la mitad. No hay validación, no hay traducción, y no hay forma de saber que `"IAM"` e `"infarto agudo de miocardio"` son lo mismo. La categoría, tratada como *string*, **pierde su identidad**.

El eje K lo corrige tratando cada categoría como un **individuo formal** con su URI canónica: `pierna_zurda` es un nodo único, no una cadena de caracteres. Las búsquedas son exactas, los alias (`"zurda"`, `"izquierda"`) apuntan al mismo concepto, los idiomas conviven, y los diccionarios internacionales se enchufan sin reescribir nada. Un valor de una lista cerrada nunca debería ser texto libre: debería ser un punto en K.

Resumen del capítulo

El eje K opera como el segundo gran zócalo fundacional de la arquitectura, el complemento abstracto de los pilares físicos (Q, O, L, T). Hemos establecido que:

- Aloja **tipos conceptuales, unidades de medida, estados enumerativos y nomenclaturas oficiales** (las cuatro familias).
- Posee una rica **estructura interna**: los conceptos se vinculan entre sí generando redes semánticas, no listas planas.
- Proporciona el terreno donde aterrizan las **ontologías industriales existentes** (Schema.org, QUDT, SNOMED, CIDOC CRM, Biolink), importando su rigor sin exigir que el usuario final hable en código.
- Dota al sistema de inteligencia habilitando la **inferencia transitiva** mediante el uso cruzado de `instancia_de` y `subtipo_de`.
- Garantiza el orden con la regla fronteriza **D1**, que separa la **plantilla atemporal** de la **instancia situada**.

Con la formalización de K, el universo de individuos que el sistema puede modelar está casi mapeado. Las cajas de Q, O, L, T y K forman el grupo de cinco ejes encargados de dar hogar a **todas las cosas materiales del mundo y a todas sus categorías teóricas**.

Pero para cuantificar la realidad de forma matemática queda un último eje fundamental en este bloque: N, el de las magnitudes y los números. Y ese eje esconde más complejidad de la que aparenta, porque rige una regla física inviolable del diseño de datos serios: **un número desnudo no significa nada; todo número válido viene acoplado a una unidad de medida**. El único lugar donde pueden habitar las unidades es el zócalo categórico de K. Por esa interdependencia arquitectónica, K tenía que presentarse con todo detalle antes de que nos atreviéramos a hablar de números. A ese eje dedicamos el capítulo siguiente.

04

Cuánto: el eje cuantitativo (N)

Un número solo es ruido con apariencia de dato. El cuarto eje no guarda cosas: guarda valores. Y el día en que un valor viaja sin su unidad, algo siempre se rompe.

El vendedor entrega la camiseta sobre el mostrador y, sin pensarlo, recita una cifra: «cuarenta y nueve noventa». Para él la frase está completa. Sabe que son cuarenta y nueve con noventa *dólares*, porque lleva diez años cobrando en la tienda y la caja solo habla en dólares. Pero si esos cuarenta y nueve noventa viajaran por un cable hacia otro sistema (una app de inventario, un panel de ventas, un agente que recomienda promociones) llegarían como un número desnudo. El sistema receptor podría leerlos como soles, como euros, como unidades vendidas. El dato del vendedor es perfecto en su cabeza y peligroso en cuanto sale de ella. Ese desfase (entre un número que «todos saben qué significa» y un número que de verdad lo dice) es el tema de este capítulo.

> **DÓNDE ESTAMOS**
> Ya tenemos los cuatro pilares del mundo físico (Q O L T) y el zócalo de las clases (K, capítulo 3). Falta el eje que aloja las magnitudes.

La venta que registra (llamémosla `venta_001`) es, por sí misma, un manojo de números: 1 camiseta vendida, 49,90 dólares de monto, 18 % de impuesto, 10 % de descuento y 180 gramos de peso de la prenda. Cinco lecturas, cinco unidades distintas. Ninguna de ellas es una «cosa» con identidad propia, como lo es la camiseta o el propio vendedor. Son respuestas a una misma pregunta repetida cinco veces: **¿cuánto?** Ese es el cuarto eje de valor, el eje N cuánto, y este capítulo lo desarma pieza por pieza.

Por qué los números no caben en el eje de las cosas

La objeción es razonable y conviene enfrentarla de inmediato: ¿para qué un eje exclusivo para los números? ¿No bastaría con guardarlos dentro del eje de los objetos, junto a las camisetas, los estantes y las cajas registradoras? La respuesta es que no, y la razón se vuelve evidente apenas intentas forzarlo.

Una camiseta es un individuo físico: tiene identidad, persiste en el tiempo, puede aparecer en varios eventos (se fabrica, se exhibe, se vende, se devuelve) y sigue siendo *esa* camiseta. El número 9, en cambio, no es una cosa. No existe un «9» guardado en algún cajón del universo al que podamos apuntar. Lo que existe son innumerables hechos del mundo que *usan* el valor 9 para describir algo: los 9 dólares de un descuento, los 9 minutos de una espera, los 9 intentos de una herramienta. Cada aparición del 9 es del todo independiente de las demás.

> **IDEA CLAVE · ENTIDADES CONTRA VALORES**
> En los ejes Q O L y K cada elemento necesita una **identidad propia e inconfundible** (un código interno, un UUID) para no confundirse con ningún otro. El eje N no necesita eso: para un número, *su valor es su identidad*. Cuando un servidor anota que un agente respondió en 340 milisegundos, nadie le inventa un código secreto al 340; el dato puro basta.

Esta diferencia, que suena filosófica, tiene una consecuencia muy concreta en la arquitectura. El eje T comparte la misma naturaleza: las fechas y los instantes se comparan por su valor matemático, no por un identificador interno. De ahí la regla que separa el universo en dos familias.

IDEA CLAVE · EJES DE ENTIDAD Y EJES DE VALOR

Q, O, L y K son ejes de entidades (físicas o conceptuales) y cada elemento porta una identidad explícita. **N y T son ejes de valor puro**: el dato *es* la identidad, y dos lecturas idénticas son, a todos los efectos, la misma. Por eso un número o una fecha pueden almacenarse en bruto, pero un agente o una clase nunca.

Conviene precisar, además, que N responde a dos preguntas emparentadas pero distintas, y que el modelo no las confunde:

N MAGNITUD

Una medida sobre una escala continua, que *siempre* arrastra una unidad: 49,90 USD, 18 %, 10 %, 180 g. Preguntar «¿cuánto cuesta?» exige nombrar la escala.

N CARDINALIDAD

Un conteo de cosas discretas, sin dimensión física: 4 camisetas, 3 intentos de una herramienta, 1 agente. La «unidad» aquí es la cosa contada, y suele bastar con saber qué se cuenta.

El número que viajó desnudo

Hay un caso que la ingeniería repite en sus aulas como una fábula moral, y vale la pena contarlo con datos frescos porque su moraleja es exactamente la regla de este eje.

LA CONFUSIÓN DE UNIDADES QUE CUESTA CARÍSIMO

El verano pasado, un equipo desplegó un sistema de monitoreo para una flota de turbinas eólicas. Dos proveedores alimentaban el mismo tablero. El fabricante de los rotores reportaba el *par de torsión* de cada eje en **libra-pie**; el módulo de control que el equipo había escrito leía esos mismos campos asumiendo **newton-metro**. Ninguno de los dos estaba equivocado por separado: cada cifra era correcta en su propia unidad. Pero entre ambas hay un factor de 1,356, y nadie lo aplicó.

Durante semanas el sistema creyó que las turbinas trabajaban muy por debajo de su límite y subió la consigna de carga. El error se acumuló en silencio hasta que un rodamiento, sometido a un esfuerzo un 36 % mayor del que el tablero mostraba, se agrietó. Nadie hizo mal una sola multiplicación. El número, simplemente, había viajado sin su unidad pegada: el mismo pecado que, a escala interplanetaria, desintegró al *Mars Climate Orbiter* en 1999, cuando un módulo entregaba libra-fuerza y el otro leía newtons.

La moraleja no es sobre aritmética. Es sobre diseño de datos, y se enuncia en una sola línea.

> *Un número sin su unidad no es información: es ruido disfrazado de dígitos.*

LA REGLA DEL EJE N

De ahí la consecuencia arquitectónica que no admite excepciones: **un valor en el eje N nunca debe viajar solo**. Tiene que ir siempre acompañado por su unidad. Y aquí los ejes empiezan a trabajar en equipo, porque una unidad (*gramo*, *segundo*, *bar*, *token*) no es un objeto físico ni un valor numérico: es una **categoría abstracta**. Y las categorías, como vimos en el capítulo anterior, viven en el eje K. El número habita en N; la unidad que lo interpreta habita en K; un predicado los enlaza.

TRIPLETAS

```
(venta_001, monto,         49.90)     ∈ M(O, N)
(venta_001, unidad_monto,  Currency:USD) ∈ M(O, K)

(sesion_ia_5521, latencia_ms,   2100)     ∈ M(O, N)
(sesion_ia_5521, unidad_lat,    milisegundo) ∈ M(O, K)
```

A primera vista parece burocracia. ¿No es obvio que el monto de una venta se cobra en dólares? Hoy sí. Pero deja de serlo el día en que una nueva sucursal reporta los montos en *soles*, y tu sistema (que asumía dólares en silencio) suma 49,90 USD con 49,90 PEN como si fueran la misma escala. La unidad explícita es el seguro contra ese día.

Cuándo conviene reificar una medición

Atar cada número a su unidad con un par de tripletas resuelve la mayoría de los casos. Pero cuando una medición va a sufrir conversiones frecuentes, o cuando importa *de dónde salió* (con qué instrumento, en qué momento, sobre qué evento), la estrategia más limpia es **reificar la medición**: elevar la medida misma al estatus de una entidad formal en el eje O, con sus propias propiedades.

> TÉRMINO
> **Reificar** es convertir lo que era una propiedad (un simple dato pegado a un sujeto) en una entidad de pleno derecho, con identidad propia y capaz de tener, ella misma, propiedades y relaciones.

Compara las dos formas de registrar el mismo monto de una venta. La versión *simple* cuelga el valor y la unidad directamente del evento; la versión *reificada* crea una entidad `medicion_monto_001` que reúne cantidad, unidad, instrumento y momento, y la enlaza al evento de la venta. La figura las pone lado a lado.

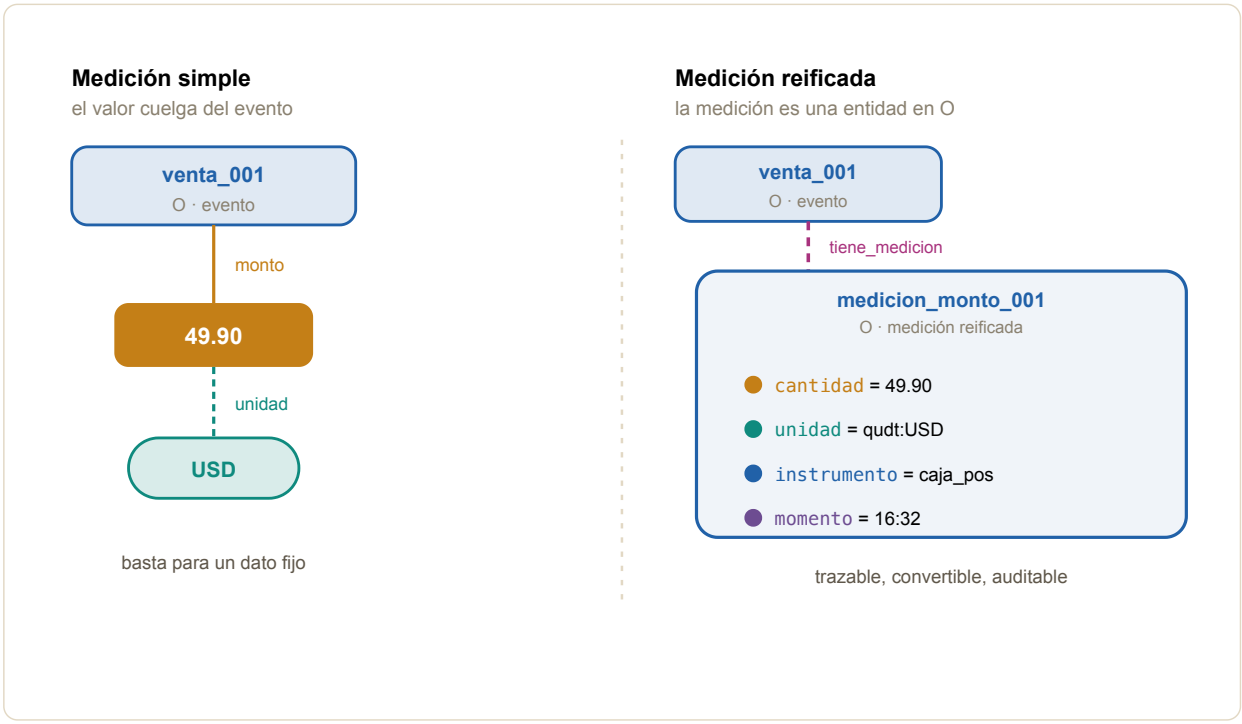


Figura 4.1. Dos maneras de modelar una misma medición. A la izquierda, el valor (N) y la unidad (K) cuelgan directamente del evento: suficiente cuando el dato es fijo y obvio. A la derecha, la medición se eleva a entidad en O y gana cantidad, unidad, instrumento y momento propios: el dato pasa de número suelto a hecho trazable.

La versión reificada es lo que separa a un sistema que escupe «49.90» de uno que afirma, con todas las letras: *«49,90 USD, cobrados por la caja del punto de venta sobre la venta `venta_001`, a las 16:32»*. El primero es un número flotando en el vacío; el

segundo es un hecho que puede auditarse, convertirse y agregarse sin riesgo.

El catálogo oficial: QUDT

Llegados aquí surge una duda práctica: ¿hace falta sentarse a escribir un diccionario con todas las unidades del mundo? La respuesta es un no rotundo. Este es uno de esos raros problemas que la industria ya resolvió de forma definitiva.

> **PRECEDENTE · QUDT** [18]
>
> La ontología pública **QUDT** (*Quantities, Units, Dimensions and Types*) `[18]` cataloga miles de unidades reconocidas: sus dimensiones físicas, sus factores de conversión exactos y sus identificadores oficiales (URIs). Para milisegundos usas `qudt:MilliSEC`; para un archivo de computadora, `qudt:Byte`; para el peso de la prenda, `qudt:GM`; para los dólares de nuestra venta, `qudt:USD`. Cuando WQuestions adopta una unidad, no la inventa: se enlaza a este catálogo.

El enganche es directo: la categoría que vive en el eje K (digamos `K:USD`) declara su anclaje canónico mediante una propiedad que apunta al recurso de QUDT. Así, dos sistemas que jamás se han visto pueden coincidir en que ambos hablan de *dólares* porque ambos apuntan al mismo URI.

JSON

```
{
  "id": "K:USD",
  "eje": "K",
  "etiqueta": "dólar estadounidense (moneda)",
  "ancla_qudt": "http://qudt.org/vocab/unit/USD",
  "dimension": "moneda",
  "convertir_a": { "unidad": "qudt:PEN", "factor": 3.75 }
}
```

Unidades que la IA tuvo que inventar

Hay industrias tan recientes que QUDT todavía no tiene palabras para ellas. El ecosistema de la inteligencia artificial es el ejemplo perfecto. ¿En qué unidad se mide la longitud de un texto que un modelo procesa? ¿Cómo se cuantifica el «tamaño» de un modelo, o lo que cuesta cada consulta? Algunas de estas magnitudes nacieron hace pocos años y aún carecen de nombre oficial en cualquier catálogo formal.

> **LA REGLA PARA UNA UNIDAD SIN CATÁLOGO**
>
> 1. Si la unidad existe en QUDT, se usa QUDT. Sin discusión.
> 2. Si es un concepto nacido ayer, el desarrollador define una unidad nueva en el eje K, la describe con precisión y, cuando es posible, le programa una fórmula que la convierta a una unidad conocida.

Para una sesión de agente como `sesion_ia_5521`, el eje K tendría que alojar unidades que ningún físico del siglo XX habría imaginado:

UNIDADES NUEVAS EN K

```
K:token                  — fragmento de texto con que un modelo lee y escribe.
K:parametro_modelo       — mide el "tamaño" de un modelo (p. ej. 7B, 70B).
```

```
K:precision_clasificador  — tasa de acierto, escala continua entre 0 y 1.
K:USD_por_millon_tokens   — unidad compuesta: con ella facturan los proveedores.
```

La última es la más interesante: una **unidad compuesta** que entrelaza dinero y capacidad de procesamiento. Un sistema maduro permite armar estas unidades híbridas y guardarlas en K sin que la arquitectura tiemble, porque «dólar por millón de tokens» no es más que una razón entre dos magnitudes que el eje ya sabe nombrar por separado.

> **NOTA**
> El precio por millón de tokens es además una magnitud *volátil*: cambia con cada nueva versión del modelo. Por eso, igual que el dinero indexado por inflación, gana mucho cuando se reifica con su fecha de denominación.

Y como toda magnitud volátil, su evolución se ve mejor en una serie temporal que en una tabla. La caída del costo por millón de tokens en los modelos abiertos del último año cuenta, por sí sola, buena parte de la historia económica de la IA reciente.

Figura 4.2. Costo por millón de tokens de un modelo abierto a lo largo de un año, en la unidad compuesta `K:USD_por_millon_tokens`. Cada cifra es una magnitud con unidad y fecha; modelarlas así permite comparar precios entre proveedores sin sumar peras con manzanas. Pasa el cursor sobre los puntos para ver los valores.

Cuando los ejes se sincronizan, las matemáticas dejan de ser ciegas

En cuanto cada número lleva su unidad pegada, operaciones que antes eran trampas mortales se vuelven seguras y automáticas. Tres ejemplos lo dejan claro.

Conversión. Imagina un tablero que recibe la latencia de cuatro agentes: uno reporta en milisegundos, dos en microsegundos y otro en segundos. Si las unidades están mapeadas en K con su factor a la unidad base, el sistema convierte todo a una escala común antes de operar. La regla de conversión vive centralizada en K, y nadie escribe traductores a mano.

Agregación. Pídele a un sistema mal diseñado el promedio de «340 ms, 1200 µs y 0,7 s» y sumará los números desnudos (340 + 1200 + 0,7) para devolverte una cifra ridícula y falsa. Con unidades explícitas, el motor sabe que no puede mezclar escalas y homogeneiza antes de calcular. Por increíble que parezca, la suma ciega ocurre a diario en tableros corporativos.

El tiempo dentro del dinero. Cien dólares de 2020 no son cien dólares de hoy. Si modelas un contrato indexado, registrar «100 USD» no alcanza: el número debe arrastrar la unidad *y* la fecha de denominación. El valor del dinero cambia con el tiempo, y el modelo lo gestiona apoyándose en la bitemporalidad que veremos más adelante.

El motor que intente sumar 100 USD con 100 JPY debe detenerse en seco, igual que se negaría a sumar camisetas con porcentajes de descuento. La cantidad deja de ser un escalar y pasa a ser un **hecho completo y autoexplicativo**: el par valor + unidad, con la unidad anclada en K mediante su URI canónica.

Incertidumbre: cuando «cuánto» no es un único número

En el mundo real los números casi nunca son exactos. Un estudio sobre un modelo de lenguaje informa una precisión del *78,3 % ± 2,1 %*; un agente no devuelve una latencia fija sino una campana de valores con su media y sus percentiles. El eje N está preparado para recibir la respuesta a «¿cuánto?» en tres formatos.

1. **Valor puntual.** Un número fijo y simple, con su unidad: `2100 ms`.
2. **Rango.** Un intervalo con piso y techo: `[1800 ms, 2400 ms]`, útil cuando solo conoces los extremos.
3. **Distribución.** Aquí el modelo se pone serio: la distribución estadística se *reifica* como entidad en el eje O y gana propiedades (media, desviación, tipo de curva, tamaño de muestra). El valor deja de ser un punto y se vuelve un objeto con estructura.

POR QUÉ ESTO IMPORTA CON LA IA

Si le haces dos veces la misma pregunta a un modelo generativo, obtienes dos respuestas distintas: su «calidad» no es un número grabado en piedra, sino una distribución de probabilidades. Cuando un informe anuncia que un modelo logra un 78,3 % de eficacia, te está dando el promedio de miles de pruebas. Modelar bien el dato significa que el sistema *sepa* que ese 78,3 % es la punta de un evento estadístico, no una métrica plana.

La venta y el agente, eje a eje

Conviene ver el eje N en acción sobre nuestros dos casos vivos. Cada uno es, mirado de cerca, un pequeño enjambre de magnitudes y conteos, cada cual con su unidad.

👕

LA VENTA DE UNA CAMISETA

Monto de 49,90 USD, descuento del 10 %, impuesto del 18 %, peso de 180 g y una talla M. Cuatro magnitudes con cuatro unidades, más un conteo solitario (1 camiseta) que es pura cardinalidad y no necesita escala. Si llamáramos a la prenda «talla M», ojo: eso no entra en N, porque la talla es una categoría del eje K, no un número.

🤖

UNA SESIÓN DE AGENTE DE IA

Entrada de 4180 tokens, salida de 920 tokens, latencia de 2100 ms, costo de 0,015 USD y precisión del 78,3 % ± 2,1 %. Seis magnitudes, seis unidades (dos de ellas inventadas por la propia industria) y dos que llegan con su margen de incertidumbre a cuestas. Ninguna de estas unidades existía en un catálogo físico clásico.

La verdadera prueba llega cuando alguien hace una pregunta que cruza ambos mundos. Por ejemplo: *«¿cuántos dólares y cuántos tokens cuesta pedirle al agente que redacte la ficha técnica de un catálogo de cuarenta camisetas?»*. Es una pregunta perfectamente razonable para un negocio, y mezcla unidades de páginas, tokens, dólares y tiempo. Responderla exige que el sistema sepa exactamente qué unidad acompaña a cada número y cómo se convierten entre sí. Sin esa capa de inteligencia estructural, cualquier respuesta es una adivinanza disfrazada de cálculo.

Un número solo dice poco; muchos cuentan el negocio

Una magnitud aislada (los 49,90 dólares de `venta_001`, su 10 % de descuento) describe un solo hecho y poco más. El eje cuantitativo cobra todo su sentido cuando se acumula: el total facturado del día, el descuento promedio de la campaña, el ticket medio por cliente. Y como cada importe es un valor tipado colgado de una situación, sumarlos o promediarlos sobre miles de ventas no es una capa que se añade por fuera: es la operación más natural del eje. Sumar montos es, sencillamente, recorrer el mismo cable `monto` sobre todas las situaciones de venta a la vez.

PYTHON

```
# Total facturado — suma del monto sobre todas las ventas
suma(u, "monto", Pattern(type_constraint=u.ind("venta")))
```

La trampa de programación: el escalar pelado

El error clásico (el que asoma en producción y cuesta dinero real) es guardar `100` en una columna llamada `monto` y dar por sentado que «ya se sabe» que son dólares. Un día el sistema integra un proveedor que factura en yenes; otro operario carga `100` pensando en soles; y la suma `100 + 100` arroja `200` de una moneda que no existe. Es la misma falla que, a otra escala, agrietó el rodamiento de la turbina y desintegró una sonda en Marte.

> **LA DESNUDEZ NUMÉRICA ESTÁ PROHIBIDA**
>
> Una magnitud jamás se almacena como un escalar pelado, sino como el par **valor + unidad**, con la unidad anclada en K por su URI canónica (QUDT). Un motor que intente sumar `100 USD` con `100 JPY` se detiene en seco, igual que se negaría a sumar peras con tornillos. El número, solo, *miente por omisión*.

El inventario de cajas está completo

Con el eje N instalado, el universo de catalogación queda formalmente cerrado. Tenemos los cuatro pilares del mundo físico (quién, qué, dónde, cuándo), el zócalo intelectual de las categorías (K) y, ahora, el terreno firme de las magnitudes (N). Ya están listos los cajones para alojar cualquier valor que exista.

Recapitulemos lo construido antes de seguir:

1. N es un eje de valor puro. A diferencia de los objetos, un número es su propia identidad y no necesita un código interno (UUID) para existir.

2. La desnudez numérica está prohibida. Todo número entra con una unidad pegada, y esa unidad habita siempre en el catálogo del eje K.

3. La medición como evento. Si la unidad no es obvia, o si habrá conversiones, la medición se reifica: se eleva a entidad formal en el eje O.

4. Estándares mundiales. Usamos QUDT para no reinventar unidades básicas; para los conceptos emergentes (token, parámetros, precisión) creamos unidades propias en K.

5. Tolerancia estadística. El eje admite la incertidumbre como rango o como distribución reificada, no solo como un promedio plano.

Lo que falta ya no son más cajones. Lo que falta son los **cables** que conecten todos estos valores entre sí: las etiquetas que le explican a la máquina *cómo* un agente se relaciona con un objeto, y cómo un objeto sostiene una magnitud. Esos cables viven en el eje M cómo, el de los predicados. Y como veremos enseguida, la frontera entre «tener una propiedad» y «estar relacionado con algo» es, a nivel informático, mucho más delgada de lo que dicta el sentido común: es solo cuestión de cardinalidad.

05

Cómo: los predicados (P y M)

Ya tenemos seis cajas donde alojar cualquier cosa del mundo. Lo que aún no tenemos son los cables que las unen. Y al examinarlos de cerca descubriremos que la vieja frontera entre «propiedad» y «relación» nunca fue más que gramática.

En el mostrador acaba de cerrarse una venta, `venta_001`, y junto a ella una pantalla donde un agente de inteligencia artificial acaba de cerrar una consulta, `sesion_ia_5521`. Tienes delante todos los datos: sabes que la operación la atendió el vendedor `vendedor_17`, que el comprador fue `cliente_1042`, que lo vendido fue una camiseta `camiseta_88` y que el monto fue 49,90 dólares; sabes que la sesión de IA corrió sobre el modelo `modelo_lumen_2026`, que consumió 4180 *tokens* de entrada y que invocó dos herramientas. Cada dato, por separado, ya tiene su casillero en alguno de los seis ejes que construimos en los capítulos anteriores. Y, sin embargo, si te detienes a mirarlos, son un montón de fichas sueltas sobre el mostrador. Nada dice todavía que la venta sea *del* vendedor 17, ni que los 49,90 dólares pertenezcan a *esa* venta y no a la de la caja de al lado.

Ese es el agujero que cierra este capítulo. Hemos llenado el universo de valores (personas, objetos, lugares, instantes, números, clases), pero los valores, por sí solos, no se conocen entre ellos. Faltan los **enlaces**: los «cables» lógicos que le dicen a la máquina qué tiene que ver cada cosa con cada cosa. En lógica y en programación esos enlaces se llaman **predicados**, y en nuestro modelo viven en el séptimo eje, el que responde a la pregunta *¿cómo?*: cómo se conecta cada cosa con cada otra.

> **DÓNDE ESTAMOS**
> Con este eje cerramos el inventario de las siete coordenadas. Los seis anteriores guardan *individuos*; el séptimo no guarda cosas, guarda los *vínculos* entre ellas. Es estructural, no de valor.

Pero antes de declararlo cerrado, el capítulo nos obliga a una pregunta técnica que parece ingenua y resulta profunda: en el fondo de una base de datos, *¿qué diferencia real hay entre una propiedad y una relación?* La respuesta (que ambas son el mismo tipo de cable y que lo único que las separa es su cardinalidad) se volverá nuestra decisión de diseño **D2**, y nos ahorrará miles de horas de programación innecesaria.

Una distinción que parece de sentido común

Calentemos motores con cinco datos sueltos, repartidos entre la venta del mostrador y la sesión del agente de IA:

CINCO ENLACES

```
1.  venta_001  tiene de monto         49.90 (USD)
2.  venta_001  se cerró a las         16:32
3.  venta_001  la atendió            vendedor_17
4.  sesion_ia_5521  consumió de entrada  4180 (tokens)
5.  sesion_ia_5521  usó la herramienta   busqueda_web, consulta_grafo
```

Pídele a un programador que empieza que divida estos cinco enlaces en dos montones: **propiedades** (atributos propios de la cosa) y **relaciones** (vínculos con otra cosa externa). Lo hará sin titubear. El *monto* y los *tokens* de entrada son, dirá, claramente propiedades: cifras que describen el objeto por dentro. En cambio *la atendió* y *usó la herramienta* son relaciones, porque atan el sujeto a otra entidad separada (un vendedor, una herramienta de software). La intuición es limpia: *propiedad = atributo interno; relación = puente hacia afuera*.

Suena impecable. Pero basta someterla a un poco de presión arquitectónica para verla desmoronarse:

- Decimos que *«la atendió vendedor_17»* es una relación externa. ¿Y si diseñamos la tabla con un campo interno `vendedor_responsable` cuyo valor es el vendedor 17? Bajo esa óptica, se vuelve una propiedad.
- Decimos que *«el monto es 49,90 dólares»* es una propiedad interna. Pero, matemáticamente, podemos leerlo como que la venta «está conectada» con el número `49.90`, que vive con derechos propios en el eje N. Bajo esa óptica, se vuelve una relación externa.
- Decimos que *«se cerró a las 16:32»* es casi un sello de tiempo, una propiedad. Pero `16:32` es un instante que habita el eje T: la venta, en realidad, se relaciona con un punto del tiempo.

Si miramos estos cinco enlaces con frialdad informática, aflora una verdad incómoda: **la diferencia entre propiedad y relación no existe en la realidad material**. Es una diferencia *gramatical*, sobre cómo los humanos redactamos las oraciones. A veces usamos verbos posesivos («tiene un monto de 49,90») y a veces verbos relacionales («la atendió el vendedor 17»). Para el disco que guarda el dato, la estructura técnica es idéntica: hay un sujeto de partida, un cable conector y un objeto de destino.

> *La frontera entre tener un atributo y estar relacionado con algo no la dibuja la realidad: la dibuja la gramática. La base de datos nunca la vio.*
>
> EL PUNTO DE PARTIDA DE D2

Esa observación es el germen de la regla D2. Pero antes de formalizarla, miremos la «estructura técnica idéntica» que comparten todos los cables.

La anatomía del cable: la signatura tipada

Todo enlace de nuestro modelo (lo llames propiedad o relación) obedece a una misma forma de tres partes:

LA FORMA DEL HECHO

```
hecho = (sujeto, predicado, objeto)
```

Hasta aquí, nada que no tenga cualquier grafo. El truco (y la diferencia de seguridad frente a un grafo libre) está en que el predicado no es un texto ciego. Viene de fábrica con una **signatura tipada**: una etiqueta, como la de un enchufe, que le dice a la máquina *de qué eje* tiene que venir obligatoriamente el sujeto y *a qué eje* debe ir a parar el objeto. La forma de esa etiqueta es:

FORMA DE UNA SIGNATURA

```
predicado : eje_sujeto → eje_objeto
```

Vista así, cada cable es una pequeña función tipada. Los enlaces de nuestras cuatro escenas (venta, agente de IA, fútbol y cine) quedan con estas etiquetas de seguridad:

Figura 5.1. Cinco predicados de cuatro dominios. Cada uno es un cable magenta que sale de un eje de sujeto y entra en un eje de objeto, según su signatura. La etiqueta no es decoración: es la que la máquina lee para decidir qué conexión es legal y cuál no.

La signatura es la magia que convierte un montón de datos desordenados en una estructura **predecible y validable**. Si un programador comete un error y trata de guardar `(venta_001, monto, "color rojo")`, el sistema lo bloquea solo: la signatura de `monto` exige que el destino esté en el eje N (los números), y «color rojo» es un texto. Esa capacidad de rechazar el absurdo desde el instante en que entra el dato es lo que separa la arquitectura sólida de WQuestions del caos de los grafos libres, donde cualquiera puede conectar una venta con un color usando el verbo «monto» sin que nada chille.

La diferencia verdadera: ¿funcional o múltiple?

Si la forma del cable es siempre la misma (sujeto, cable, objeto), ¿queda alguna razón técnica para distinguir entre «propiedades» y «relaciones»? Sí, una sola, y es estrictamente matemática: la **cardinalidad**.

Vuelve a mirar las signaturas de la figura. Cuatro de ellas se comportan como lo que en matemáticas llamamos **funciones**: dado un sujeto concreto, el cable solo puede conducir a *un único* destino en ese instante. Una venta tiene *un* monto; se cerró en *un* instante; la atendió *un* vendedor; una sesión de IA consumió *un* número exacto de *tokens* de entrada.

El quinto cable (`uso_herramienta`) rompe la regla. Un mismo agente de IA pudo invocar la búsqueda web *y* la consulta al grafo *y* una calculadora, todo en la misma sesión. Un solo sujeto dispara varios cables hacia múltiples objetos a la vez, y todos son correctos.

Esa es la única diferencia estructural, en la base de datos, entre lo que solemos llamar «propiedades» y «relaciones»:

CABLE FUNCIONAL — P

Admite **un único destino** por sujeto. Es lo que el sentido común llama una *propiedad*.

`monto`, `se_cerro_a`, `tokens_entrada`, `latencia_ms`.

CABLE MÚLTIPLE — M

Permite **varios destinos** a la vez. Es lo que el sentido común llama una *relación*.

`uso_herramienta`, `ingrediente`, `gol_anotado`, `reparto`.

POR QUÉ DOS LETRAS

A los predicados funcionales los marcamos como **P** y a los múltiples como **M**. No son dos ejes distintos: son dos *modos* de un mismo eje. La letra solo nombra la cardinalidad declarada en la signatura.

Pero ambos son el mismo tipo de cable y viven en el mismo eje. La cardinalidad no es un eje aparte: es un **atributo de cada cable**, escrito en su signatura. Y lo único que ese atributo le dice a la base de datos es **cómo comportarse cuando el dato se actualice**. Aquí está la razón de fondo por la que conviene conservar las dos marcas (P y M) aunque algebraicamente sean la misma familia de funciones tipadas: cada una gobierna una *lógica de actualización* distinta.

> **LA LÓGICA DE ACTUALIZACIÓN**
>
> Si el cable es **funcional (P)** y llega información nueva, el sistema **borra y reemplaza**. Si el cajero corrige el monto de `venta_001` de 49,90 a 45,90 dólares, el 49,90 desaparece; no acumulamos dos montos.
>
> Si el cable es **múltiple (M)** y llega información nueva, el sistema la **agrega a la lista**. Si el agente de IA invoca una tercera herramienta, esa herramienta se suma al historial sin borrar a las dos anteriores.

Y aquí viene el alivio para quien programa: **el motor de consulta no hace la diferencia**. Da igual que preguntes «¿qué monto tiene `venta_001`?» o «¿con qué herramientas trabajó `sesion_ia_5521`?»: la máquina ejecuta exactamente el mismo código de búsqueda. Solo que en el primer caso te devuelve una respuesta y en el segundo, una lista.

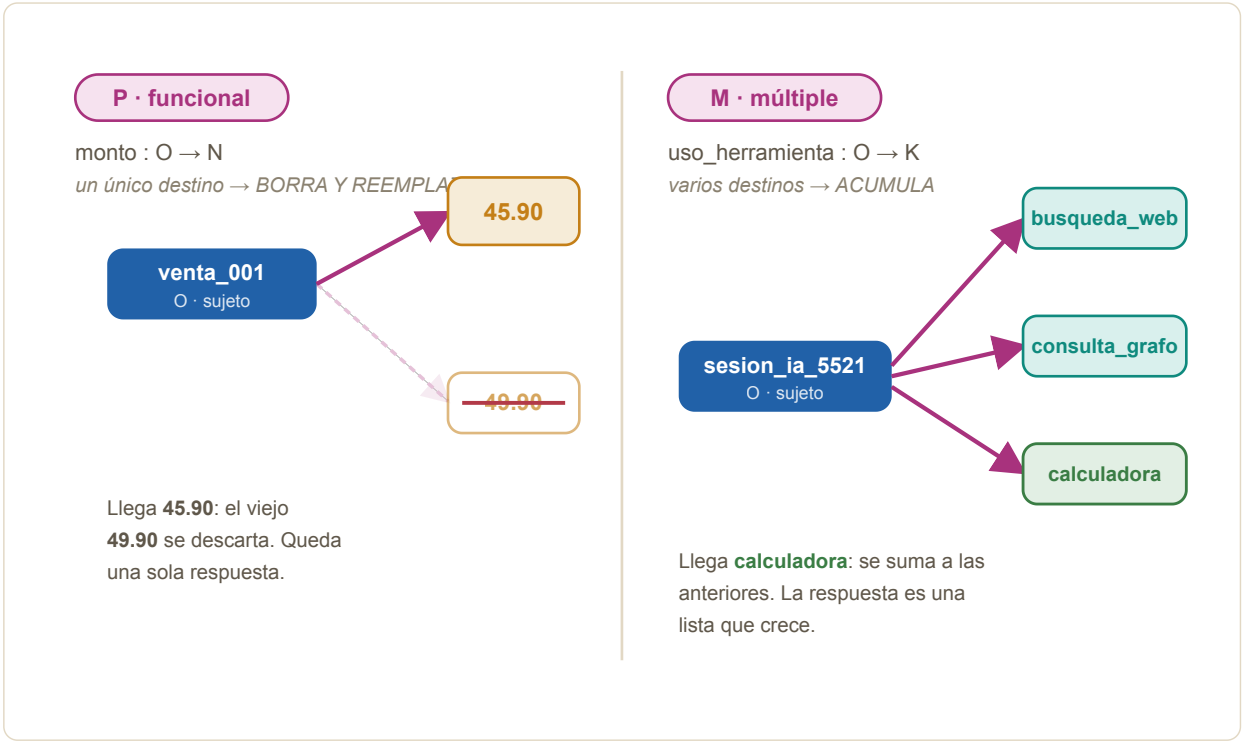


Figura 5.2. La única diferencia real entre una propiedad y una relación. A la izquierda, un cable **funcional (P)**: cuando llega un valor nuevo, el anterior se descarta. A la derecha, un cable **múltiple (M)**: el valor nuevo se acumula. Misma forma de cable, misma consulta; cambia solo qué hace el motor al actualizar.

Un cable conecta dos cosas, sí; pero el mismo cable, repetido sobre miles de sujetos, es una relación recorrible de punta a punta. «¿Qué sesiones usaron la búsqueda web?» no se responde mirando a `sesion_ia_5521`: se responde recorriendo el cable `uso_herramienta` hacia atrás, desde todas las sesiones que lo tienen.

La regla de diseño D2: la unificación algebraica

Esta revelación nos permite enunciar de manera formal una de las decisiones de diseño más elegantes del modelo. Reúne todo lo anterior en una sola frase:

D2 PREDICADOS UNIFICADOS

Las **propiedades** y las **relaciones** se unifican bajo el mismo concepto: son simplemente cables (predicados) del eje *cómo* (M) con etiquetas de seguridad (signaturas tipadas). La única diferencia entre ellas es la **cardinalidad** (si aceptan uno o varios destinos simultáneos), y esa cardinalidad es un atributo de la signatura de cada cable, no un eje aparte. Le indica a la base de datos cuándo debe «borrar y reemplazar» (cable funcional) y cuándo debe «acumular» (cable múltiple) nueva información.

Conviene precisar la relación entre D2 y la doble marca P / M, porque a primera vista parecen tirar en sentidos opuestos. **Algebraicamente, todo es un mismo objeto**: un predicado tipado con la forma `predicado : eje_sujeto → eje_objeto`. No hay dos familias de cables ni dos lenguajes. **Operativamente, conviene seguir distinguiendo dos modos** (P para los funcionales, M para los múltiples) porque la cardinalidad no es un adorno: gobierna la lógica de actualización, y un sistema que la ignore no sabría si un dato nuevo pisa al viejo o se le suma. P y M no son, pues, dos ejes: son la misma coordenada vista bajo su único atributo que de verdad cambia el comportamiento del motor.

Entender D2 significa comprender que el abismo técnico que se enseña en la universidad (donde nos obligan a tratar los atributos internos como «columnas en una tabla» y los vínculos externos como «tablas conectadas o *joins*») es, en realidad, una **ilusión pedagógica**. Una complicación heredada de la programación tradicional. A nivel matemático, todo es un sujeto, un cable tipado y un destino. Y esa simplicidad es la gasolina que hace volar al modelo.

Cuatro cosas que gana una empresa al unificar

Adoptar D2 no es una elegancia teórica: tiene cuatro consecuencias prácticas, directas y medibles en producción.

1 UN MOTOR DE CONSULTA ÚNICO

El sistema deja de necesitar «un lenguaje para datos internos» y «otro para cruces de tablas». Todo se reduce a una instrucción maestra: *dado este sujeto y este cable, devuélveme el destino*. Adiós a los `SELECT` y los `JOIN` enredados del SQL tradicional.

2 UN JSON ÚNICO

Cuando un agente de IA pide datos por *function calling*, los recibe siempre con la misma estructura limpia. No gasta *tokens* averiguando si el monto viene en un formato y la lista de herramientas en otro. La uniformidad abarata la integración.

3 EXTENSIBILIDAD SIN MIGRACIONES

Para registrar un dato nuevo, nadie debate si crear una columna o una tabla intermedia. Se añade el cable al diccionario, se decide si su respuesta es única o múltiple, y el motor ya sabe qué hacer.

4 LENGUAJE NATURAL, GRATIS

Cada verbo humano («vender», «anotar», «dirigir») exige ciertos roles. Como el sistema no separa propiedades de relaciones, leer «el vendedor 17 vendió una camiseta» y volcarlo a cables uniformes es casi mecánico.

Esa última consecuencia merece subrayarse, porque es el puente hacia toda la Parte IV. Cuando le decimos al sistema *«Messi anotó el gol tras la asistencia de Di María»*, la máquina registra al goleador, el evento y el asistente usando la misma estructura uniforme. Al no distinguir entre propiedades y relaciones, lee frases humanas casi como si leyera código de máquina.

Tres dominios bajo el microscopio

Para asentar el concepto, veamos cómo se reparten los cables funcionales y los múltiples al diseñar tres mundos sin relación entre sí. Fíjate en el patrón: lo que el sentido común considera «atributos estáticos» tiende a ser funcional; lo que son «piezas que se suman» tiende a ser múltiple.

La venta del mostrador

SIGNATURAS · VENTA

```
# funcionales (P) — respuesta única, borra y reemplaza
monto          : O → N    P   # 49.90
atendida_por   : O → Q    P   # vendedor_17
comprada_por   : O → Q    P   # cliente_1042
categoria      : O → K    P   # venta

# múltiples (M) — acumulan
articulo       : O → O    M   # varios objetos en una venta
etiqueta       : O → K    M   # camiseta, promocion…
```

La sesión del agente de IA

SIGNATURAS · AGENTE IA

```
# funcionales (P)
tokens_entrada : O → N    P   # 4180
tokens_salida  : O → N    P   # 920
latencia_ms    : O → N    P   # 2100
costo_usd      : O → N    P   # 0.015
modelo_usado   : O → K    P   # modelo_lumen_2026

# múltiples (M)
uso_herramienta : O → K   M   # busqueda_web, consulta_grafo…
consultó_fuente : O → O   M   # la IA pudo leer 5 documentos
```

El consumo exacto en dinero es uno solo (funcional); las herramientas invocadas y los documentos leídos durante la consulta pueden ser docenas (múltiples).

El partido de fútbol

SIGNATURAS · FÚTBOL

```
# funcionales (P)
resultado    : O → K    P   # victoria_local
asistencia   : O → N    P   # 52000 espectadores

# múltiples (M)
juega        : O → Q    M   # SON SIEMPRE 2 equipos — y aun así, múltiple
gol_anotado  : O → O    M   # varios goles
tarjeta      : O → O    M   # decenas de tarjetas
```

LA TRAMPA DE `JUEGA`

El cable `juega` engaña. Podrías creer que es **funcional** porque el número de equipos en un partido *siempre* es exactamente dos: un número fijo. Pero es **múltiple**. La regla no trata sobre si el número es fijo o infinito, sino sobre si hay **una única respuesta posible**. Como hay dos respuestas legítimas para el mismo partido (Argentina *y* Perú), matemáticamente el cable es múltiple. «Fijo en dos» no es lo mismo que «único».

Trampa de programación: la relación disfrazada de columna

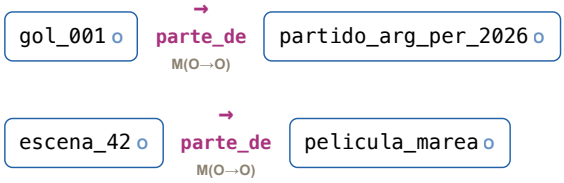
Hay un error que parece inocente y se paga carísimo al escalar: modelar una relación como una **columna** de la tabla. La directora `serra` tiene un teléfono, así que el sistema crea una columna `telefono`. Funciona —hasta el día en que alguien tiene dos números—. Entonces empieza el parche: `telefono2`, `telefono_alt`, un campo de texto con comas, una tablita improvisada a las apuradas. La columna había asumido, sin decirlo, una **cardinalidad** (un solo valor) que el mundo no respeta.

El eje M evita el problema de raíz: pone la cardinalidad **en la signatura del predicado**, no en la forma de la tabla. `telefono` se declara de antemano como funcional (un valor) o múltiple (varios), y el modelo lo respeta sin rediseños. Pasar de «un teléfono» a «muchos» deja de ser una migración de tabla y se vuelve cambiar una marca en M (de P a M). Distinguir «tener un atributo» de «estar relacionado con algo» deja de ser una decisión rígida de esquema y pasa a ser un dato más: un cable con su cardinalidad escrita en la firma.

Un cable especial: `parte_de` y la puerta a las situaciones

Antes de cerrar la Parte II hace falta presentar un predicado que merece trato aparte, porque será una bisagra del resto del libro. Hasta aquí hemos tratado cada hecho como un punto suelto. Pero el mundo no viene en puntos sueltos: viene en escenas. El gol de Messi no flota en el vacío —ocurre *dentro* de un partido—. La escena 42 no es una isla —es una pieza *de* la película `pelicula_marea`—. El acto de cobrar la venta pertenece *a* la jornada de caja de toda la mañana.

Ese vínculo de pertenencia tiene su propio cable. Es un predicado múltiple, de objeto a objeto, y lo llamamos `parte_de`:



> **SUBOBJETO CONTINGENTE**
> Llamamos **subobjeto contingente** a una entidad que existe como pieza de otra mayor y cuya identidad *depende* del marco que la contiene. La escena 42 no significa nada fuera de *su* película.

Lo notable de `parte_de` no es su forma (es un cable múltiple corriente), sino lo que **habilita**. Cuando muchos hechos pequeños cuelgan, por este predicado, de un mismo hecho contenedor, ese contenedor deja de ser un objeto cualquiera y se convierte en una **situación**: un marco amplio dentro del cual ocurren eventos interconectados. El partido entero, el rodaje completo, la jornada de caja. Es la entidad *compuesta* que vimos asomar al hablar del eje *qué*, y que ahora tiene el cable que la ensambla.

Por eso `parte_de` es un **subobjeto contingente**: la pieza existe en función del todo. Y por eso este predicado es el puente entre lo que hemos hecho hasta ahora (hechos atómicos, sueltos) y lo que viene en la Parte III: las situaciones, los contextos y la agencia. Dejamos puesto el cable; en la sala de máquinas lo veremos sostener escenas enteras.

Cierre de la Parte II: el tablero está listo

Con la formalización del eje *cómo*, terminamos de armar el andamiaje del modelo. El inventario de las siete coordenadas queda oficialmente cerrado:

- Q quién: las personas y agentes que actúan.
- O qué: los objetos físicos, los eventos y las situaciones.
- L dónde: las locaciones.
- T cuándo: el flujo del tiempo.

N cuánto: los números y las magnitudes.
K cuál: los conceptos y las categorías: tipos, unidades, estados, vocabularios.
M cómo: los cables o predicados que conectan los ejes de valor, funcionales (P) o múltiples (M).

Seis cajas gigantes para guardar «cosas», entrelazadas por una red de cables estructurales (el eje *cómo*). Cualquier fenómeno, suceso, registro bancario o reporte médico que ocurra en el mundo cabe, sin distorsiones, dentro de esta matriz de siete ejes. Cumplimos la promesa con la que abrió el libro: el universo de los datos puede mapearse usando únicamente las preguntas fundamentales del cerebro humano.

EL UNIVERSO V

Bautizamos como **V** a la unión de los seis ejes de valor (Q, O, L, T, N, K). No es un séptimo eje ni una caja nueva: es *todo lo que puede ser valor* de un hecho, frente a M, que son los cables y no los valores.

Conviene dejar nombrada una abreviatura que hará falta más adelante. Al conjunto de esas seis cajas de valor (Q, O, L, T, N, K tomadas juntas) lo llamamos **V**, el *universo de valores*. Lo nombramos desde ya porque cuando un predicado no se ate a un eje concreto, sino que admita *cualquiera* de los seis, diremos que su dominio o su rango es V. Lo veremos al definir las signaturas con todo rigor, en el Capítulo 13.

IDEA CLAVE

Con los seis ejes de valor llenamos el universo de cosas; con el séptimo (los predicados) lo conectamos. Y al examinar de cerca esos enlaces descubrimos que «propiedad» y «relación» nunca fueron dos especies distintas, sino el mismo cable tipado bajo dos cardinalidades.

Pero tener el plano del motor en la mano no es lo mismo que encenderlo. Hasta aquí vimos los ejes por separado, como piezas sueltas de un reloj. Antes de armarlo y verlo girar, conviene cerrar una pregunta que dejamos abierta en el primer capítulo: *¿por qué estas preguntas, y no otras?* Es una pausa breve (el próximo capítulo) y enseguida entramos en la etapa operativa: el modelo en movimiento, observando cómo interactúan todos los ejes juntos para estructurar escenarios reales de altísima complejidad.

06

Las raíces de las preguntas

Hemos visto cuáles son las coordenadas. Falta lo más incómodo: por qué son estas y no otras. La respuesta no está en el diseño, sino en cuatro tradiciones que llegaron, sin hablarse, a la misma lista.

Es una noche de invierno de 1917, en el aula de una escuela de periodismo del medio oeste norteamericano. El profesor entra, toma la tiza y, antes de decir una palabra, escribe seis letras en la pizarra: **W W W W W H**. Debajo de cada una anota su nombre en inglés (*who, what, where, when, why, how*), que para nosotros son *quién, qué, dónde, cuándo, por qué* y *cómo*. Luego reparte recortes de diario y dicta la consigna de la noche: subrayar, con un color por letra, dónde responde cada texto a cada una de las seis preguntas. Si a un recorte le falta una sola respuesta, la noticia se descarta por incompleta. Nadie aprueba con cinco de seis.

> **DE DÓNDE VENIMOS**
> En los capítulos anteriores diseccionamos Q, O, L, T, el zócalo categórico K, las magnitudes N y la urdimbre de predicados M. Tenemos las piezas. Este capítulo pregunta por qué son *estas* piezas.

Ese ejercicio no era la ocurrencia de un profesor. Era doctrina. Cuatro años antes, el manual *Newspaper Writing and Editing* de Willard Bleyer había codificado una regla que toda nota informativa bien construida debía cumplir: responder, idealmente en el primer párrafo, a esas seis preguntas. La fórmula se bautizó **5W1H**, atravesó el siglo XX sin un rasguño, sobrevivió a la radio, a la televisión y al salto digital, y todavía se enseña hoy como el abecé del oficio en cualquier escuela de comunicación del planeta.

Lo interesante no es preguntar de dónde sacó Bleyer la regla. Es preguntar por qué *funcionó*. Cuando un académico inventa una norma por capricho, la profesión la olvida en una generación. Cuando una regla resiste más de cien años en un entorno que cambió todo lo demás, es porque tropezó con algo que estaba ahí antes que ella. Las 5W1H no sobrevivieron por ser un hallazgo brillante del periodismo. Sobrevivieron porque eran el redescubrimiento moderno de un patrón que la humanidad ya había encontrado (y vuelto a encontrar) muchas veces.

> **BLEYER Y LAS 5W1H · PERIODISMO, 1913** (3)
>
> Willard G. Bleyer, en *Newspaper Writing and Editing* (Houghton Mifflin, 1913), fija como obligación de toda nota informativa responder a *who, what, where, when, why, how*. La regla no nace del análisis lingüístico ni del filosófico: nace de la práctica de la redacción. Y aun así aterriza, sin saberlo, sobre la misma lista a la que habían llegado los juristas romanos y los filósofos griegos. Esa coincidencia es justamente el dato a explicar.

Antes de seguir, conviene nombrar la pregunta que un lector escéptico tiene todo el derecho a hacer en este punto del libro: *¿por qué estas siete preguntas y no otras seis, u ocho? ¿No será este reparto un capricho elegante del autor?* La respuesta honesta es la única que vale la pena dar:

> *No elegimos estas preguntas. Las encontramos.*

LA TESIS DE ESTE CAPÍTULO

Veinte siglos antes del aula

Bleyer no era un erudito en historia antigua, pero la matriz que acababa de plasmar en su manual llevaba más de dos mil años en circulación. Si rastreamos la idea hacia atrás en el tiempo, damos de bruces con la retórica de la antigua Roma y, en particular, con Cicerón.

A mediados del siglo I antes de Cristo, en su tratado *De inventione*, Cicerón (recogiendo ideas del griego Hermágoras) sostuvo que cualquier análisis serio del acto de una persona debía organizarse en torno a un repertorio fijo de *circumstantiae*, las circunstancias del hecho:

> *quis, quid, ubi, quibus auxiliis, cur, quomodo, quando.*
>
> *Quién, qué, dónde, con qué medios, por qué, cómo, cuándo.*

Para Cicerón esto no era adorno retórico, sino ingeniería legal. Si un abogado pretendía absolver a un acusado alegando que obró por necesidad, o condenarlo probando malicia, tenía que reconstruir los hechos circunstancia por circunstancia. Olvidar una de ellas dejaba una grieta estructural en el argumento, y por esa grieta la contraparte desarmaba el caso entero.

CICERÓN · *DE INVENTIONE* , LIBRO I (S. I A.C.) (2)

La lista canónica de las *circumstantiae* romanas mapea, una a una, sobre nuestras coordenadas (con una excepción reveladora):

- Q *quis* · quién
- O *quid* · qué
- L *ubi* · dónde
- T *quando* · cuándo
- M *quomodo* · cómo
- · *cur* · por qué
- · *quibus auxiliis* · con qué medios

El *cur* («por qué») no es una coordenada que fije la posición de un hecho: es lo que enlaza un hecho con otro. Lo retomaremos en el capítulo 10. Y los «medios» (*quibus auxiliis*) son, en nuestro lenguaje, un objeto más que cumple un rol instrumental: caen dentro de O.

Un siglo después, Quintiliano[25] perfeccionó el método en su *Institutio Oratoria*, y desde ahí la fórmula viajó intacta hasta la Edad Media. En la *Summa Theologica*, Tomás de Aquino[26] retomó las mismas circunstancias del acto, esta vez para evaluar si una acción era moralmente buena o mala. La cadena de transmisión es nítida: los juristas romanos le pasaron el esquema a los teólogos medievales, estos a los pensadores modernos, y a fines del siglo XIX la misma lista reapareció (ya sin sus citas en latín) como manual práctico para las redacciones norteamericanas.

NOTA

Visto así, resulta casi cómico que Bleyer creyera estar patentando la rueda del periodismo cuando en realidad solo la estaba desenterrando. La lista llevaba veintidós siglos esperándolo.

La lección que importa no es histórica, sino arquitectónica. Si la misma lista exacta reaparece sola en escenarios tan dispares como un tribunal romano, una iglesia medieval y un periódico moderno, es porque hay una fuerza de gravedad que la obliga a volver. Y esa fuerza no es la tradición (las tradiciones se diluyen y se olvidan): la lista vuelve porque es la única solución lógica a un problema universal de procesamiento de información. ¿Cómo se describe un evento del mundo sin dejar puntos ciegos?

Aristóteles, todavía más atrás

Y la historia no empieza con Cicerón. Trescientos años antes de que Roma sistematizara el derecho, Aristóteles ya había hecho el mismo ejercicio analítico, con otro vocabulario. En la *Ética a Nicómaco* se enfrentó a un problema concreto: distinguir una acción hecha a propósito de una ocurrida por accidente. Para resolverlo inventarió las variables que una persona debe conocer al actuar para que se la pueda considerar plenamente responsable.

ARISTÓTELES · *ÉTICA A NICÓMACO* , LIBRO III (S. IV A.C.) (1)

Para juzgar si alguien actuó en ignorancia, escribe Aristóteles, hay que verificar si desconocía: **(a)** quién es él mismo, **(b)** qué está haciendo, **(c)** a quién o sobre qué recae la acción, **(d)** con qué instrumento, **(e)** por qué, y **(f)** cómo: si aplicó fuerza moderada o violencia. El desglose es el mismo que venimos rastreando: agente, acción, paciente, instrumento, motivo y modo.

Falta el *cuándo*, y su ausencia aquí es perfectamente lógica: para decidir si alguien es responsable de haber golpeado a otro, la hora exacta del reloj es casi siempre irrelevante. Pero el *cuándo* reaparece con fuerza en otros textos de Aristóteles, justamente donde sí importa: cuando teoriza sobre el tiempo, el movimiento y el cambio físico. La dimensión existe; lo que cambia es si el problema en cuestión la activa o no.

IDEA CLAVE

La filosofía clásica ya operaba con una intuición metodológica poderosa: para comprender cualquier fenómeno hay que desensamblarlo en sus **dimensiones naturales**. Y esas dimensiones, sin falta, terminan mapeándose sobre las preguntas básicas. Lo que en este libro llamamos «ejes», Aristóteles lo llamaba «circunstancias del acto». El nombre cambió; la estructura, no.

El testimonio de la gramática

Hasta aquí, un lector suspicaz podría sospechar un sesgo cultural: tres testigos (Aristóteles, Cicerón, Bleyer) que beben, al fin y al cabo, de la misma tradición occidental. La objeción es justa, y la respuesta más contundente no viene de la historia, sino de una disciplina estrictamente técnica: la lingüística comparada.

Toma cualquier lengua viva del planeta (quechua, mandarín, suajili, euskera, árabe, español) y observa cómo sus hablantes interrogan la realidad. Encontrarás, sin excepción, el mismo puñado compacto de palabras: *quién, qué, dónde, cuándo, cómo, por qué, cuál* y *cuánto*. Cada idioma tiene sus rarezas (algunos distinguen el «quién» singular del plural, otros separan «dónde estoy» de «hacia dónde voy»), pero el núcleo semántico es asombrosamente idéntico en toda la especie.

A mediados del siglo XX, la lingüista Anna Wierzbicka dirigió un programa monumental para identificar conceptos que significaran exactamente lo mismo en todas las culturas: el **Metalenguaje Semántico Natural** (NSM). En su catálogo final de primitivos universales aparecieron, sin sorpresa, *alguien* (quién), *algo* (qué), *dónde*, *cuándo* y *por qué*.

Esos términos no entraron por conveniencia teórica: se ganaron el lugar tras superar pruebas de traducción exhaustivas en decenas de lenguas, muchas de comunidades aisladas que nunca tuvieron contacto con Occidente. Si una palabra falla la prueba en una sola de esas lenguas, queda fuera del catálogo. Las preguntas básicas la pasaron en todas.

WIERZBICKA · NATURAL SEMANTIC METALANGUAGE (1972–1996) (28)

El programa de Anna Wierzbicka (*Semantic Primitives*, 1972; *Semantics: Primes and Universals*, Oxford, 1996) busca el conjunto mínimo de conceptos compartidos por todas las lenguas humanas. Que los interrogativos sobrevivan a esa criba es la evidencia translingüística más fuerte de que las preguntas no son un artefacto cultural, sino un sustrato común a la especie.

Casi en paralelo, la lingüística formal le puso nombre técnico al mismo fenómeno, un nombre que todavía se usa en ciencias de la computación: los **roles temáticos**. En 1968, Charles Fillmore postuló que la mente, al procesar cualquier verbo, asigna automáticamente roles fijos a las entidades que lo rodean: un *agente* (quién), un *tema* o paciente (qué), una *locación* (dónde), una *temporalidad* (cuándo), un *instrumento* (con qué) y un *beneficiario* (para quién). Según la escuela, el modelo trae ocho o doce roles, pero el chasis estructural coincide al milímetro con el análisis que Aristóteles había hecho veintitrés siglos antes.

FILLMORE · «THE CASE FOR CASE» (1968) (24)

Charles J. Fillmore, en *Universals in Linguistic Theory* (Holt, Rinehart and Winston, 1968), funda la teoría de los roles temáticos. Es el puente directo entre la intuición filosófica antigua y la ingeniería lingüística moderna: los mismos «papeles» que un verbo reparte (agente, tema, lugar, instrumento) son, con otro nombre, los ejes Q, O, L y T. El verbo, lo veremos en el capítulo 13, es la signatura que dice qué ejes activa cada hecho.

El recuento, entonces, es elocuente: cuatro disciplinas, cuatro nomenclaturas distintas. Aristóteles hablaba de *circunstancias del acto*; Cicerón, de *circumstantiae*; el periodismo, de *5W1H*; la lingüística formal, de *roles temáticos*. Al tabularlas y compararlas, la conclusión es ineludible: todos descubrieron la misma estructura base de la información.

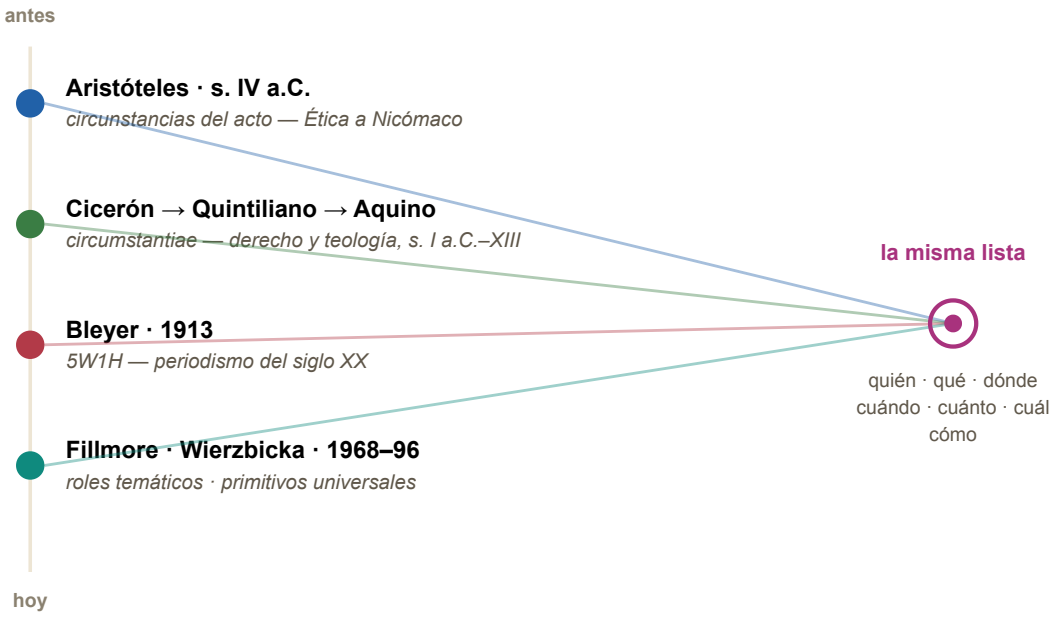


Figura 6.1. Cuatro tradiciones independientes (filosofía griega, derecho romano, periodismo moderno y lingüística formal), separadas por más de dos mil años y sin coordinarse entre sí, convergen en el mismo conjunto reducido de preguntas. Una coincidencia así no se explica por la herencia cultural: se explica porque la lista es la respuesta forzosa a un problema común.

El niño que pregunta

Queda una última pieza de evidencia, y es la más profunda, porque es biológica. Este inventario de preguntas no vive solo en tratados de filosofía o en la estructura de idiomas milenarios: brota de forma espontánea en el desarrollo cognitivo temprano de cualquier niño, y lo hace en un orden de aparición sorprendentemente estricto, sin importar el país donde nazca.

En 1973, Roger Brown publicó un estudio longitudinal clásico sobre la adquisición de la lengua materna. Documentó un patrón que después se replicó en niños hispanohablantes, franceses, japoneses y hebreos: los infantes no aprenden todas las preguntas a la vez. El cerebro las desbloquea en una secuencia que apenas varía.

BROWN · *A FIRST LANGUAGE* (HARVARD, 1973) (27)

El orden de adquisición que documentó Brown (y que estudios posteriores confirmaron entre lenguas) es:

1. **qué**: «¿Qué es eso?» El escaneo más básico: nombrar el entorno material y construir vocabulario.
2. **dónde**: «¿Dónde está mamá?» Emerge con la permanencia del objeto: las cosas siguen existiendo aunque salgan de la vista.
3. **quién**: «¿Quién hizo esto?» El cerebro aprende a separar un objeto pasivo de un agente con voluntad y fuerza.
4. **cuándo**: tarda bastante más (hacia los tres años): exige abstraer pasado, presente y futuro.
5. **por qué**: casi a la par; abre la célebre «etapa de los porqués» y la búsqueda de causas.
6. **cómo**: el último en consolidarse: requiere concebir un proceso secuencial completo, no un hecho aislado.

Figura 6.2. Edad aproximada de aparición de cada pregunta en el habla infantil, según el orden documentado por Brown. Las cifras son indicativas (varían entre niños) pero el *orden relativo* se mantiene estable a través de las lenguas. Lo último en llegar es el *cómo*, la pregunta que exige representar un proceso entero; no por casualidad es también el eje M, el de los predicados que enlazan.

La belleza arquitectónica del hallazgo está en lo que el orden *no* depende. No depende de la cultura ni de la dificultad gramatical de cada idioma: depende de la carga cognitiva del concepto. Identificar un objeto tangible («¿qué es eso?») cuesta menos que ubicar una acción en un continuo espacio-temporal («¿cuándo ocurrió?»). Si todos los humanos, vengan de donde vengan, despliegan estas preguntas en el mismo orden, la conclusión es difícil de esquivar: las preguntas no son una convención que la sociedad inventó. Son el firmware de fábrica con el que la mente le da sentido a la entropía de la realidad.

Las preguntas como invariantes cognitivos

Con todo el panorama empírico sobre la mesa, podemos por fin enunciar en voz alta la tesis que sostenía en silencio todo lo construido en los capítulos previos.

LA HIPÓTESIS DE LOS INVARIANTES COGNITIVOS

Las preguntas fundamentales (quién, qué, dónde, cuándo, cuánto, cuál, cómo) no son un capricho cultural ni un artefacto inventado por una disciplina académica. Son **invariantes cognitivos**: las dimensiones estructurales universales con las que el cerebro humano desensambla y almacena la realidad. Operaban con total precisión mucho antes del derecho, la estadística, las bases de datos relacionales, e incluso antes de que un niño sepa armar una oración fluida.

Es una afirmación audaz, así que conviene tratarla como lo que es: una hipótesis con predicciones contrastables. Una buena hipótesis hace cuatro apuestas; las cuatro tienen confirmación.

1 ¿APARECEN EN EL ANÁLISIS HUMANO?

Sí. Filósofos griegos, juristas romanos, periodistas y lingüistas convergieron en el mismo vector dimensional sin haber cruzado un solo dato metodológico entre ellos.

2 ¿SON INDEPENDIENTES DEL IDIOMA?

Sí. Los interrogativos forman una clase léxica universal: conservan su núcleo semántico hasta en lenguas sin parentesco etimológico alguno.

3 ¿EMERGEN DE FORMA INNATA?

Sí. Su aparición en el desarrollo infantil sigue un cronograma rígido, transversal a culturas y lenguas, dictado por la complejidad del concepto y no por la del idioma.

4 ¿PRECEDEN A LA CIENCIA?

Sí. Ningún niño aprende taxonomías biológicas ni física antes de dominar la operación básica del «¿qué?». La pregunta es anterior a toda especialización.

Cuatro predicciones, cuatro confirmaciones sólidas. Las ciencias cognitivas siguen debatiendo detalles finos de estos procesos (cuántos roles exactos, dónde trazar los límites), pero el volumen de evidencia apunta unívocamente al mismo centro de gravedad.

El cimiento de diseño más robusto

Llegados aquí, un arquitecto de software pragmático tiene derecho a impacientarse: *todo este recorrido por la antropología y la filosofía es fascinante, pero ¿de qué me sirve si lo único que quiero es diseñar una base de datos que escale o programar el backend de un agente?*

De mucho, y de forma estrictamente **arquitectónica**. Si las preguntas *son* la estructura nativa con la que la mente clasifica la información, entonces el modelo que venimos construyendo no descansa sobre una preferencia de diseño, sino sobre el cimiento más estable que existe. Anclarlo a las preguntas es más seguro que anclarlo a los vocabularios de cada industria (que mutan año a año), más duradero que apostar por los «estándares» de bases de datos (que caducan cada década) y, llevado al extremo, más perdurable que las propias lenguas maternas: el euskera y el español suenan incomprensibles entre sí, pero la matemática de sus preguntas es la misma.

LA VENTAJA EN LA ERA DE LOS LLMS

Y hay un dividendo que solo cobra sentido hoy. Cualquier IA moderna (entrenada devorando millones de textos escritos por mentes humanas) asimila este esquema de forma nativa e inmediata. No hay que invertir meses en enseñarle a navegar las tablas crípticas de una base corporativa: el modelo ya entiende la topología, porque está organizada con la misma lógica con la que aprendió a leer. Lo desarrollaremos en el capítulo 26.

Con esto cerramos la Parte II. Ya sabemos *cuáles* son las coordenadas y, ahora, *por qué* son esas y no otras: porque no las inventamos, las encontramos, al final de cuatro caminos que la humanidad recorrió por separado y que terminan en el mismo lugar. Lo que sigue es ver cómo se ensamblan entre sí: cómo un evento del mundo se convierte en la unidad atómica del sistema y cómo la intersección de los ejes da lugar a una geometría de datos que se puede consultar con rigor.

07

El hecho atómico

Seis ejes sueltos no describen nada. Hace falta una pieza que los conecte (una sola, repetida millones de veces) para que el modelo deje de ser una lista de columnas y empiece a ser un mundo.

Son las 16:32 de la tarde y el vendedor Marco cierra la venta de una camiseta. Ese gesto, que dura tres segundos, esconde más datos de los que parece: hubo alguien que la vendió, un cliente que la pagó, un artículo entregado, un monto en dólares y una hora exacta. En los seis capítulos anteriores aprendimos a clasificar cada uno de esos fragmentos en su eje: Marco es un Q, la camiseta un O, los 49.90 dólares un N. Pero clasificar no es describir. Tenemos las piezas separadas sobre la mesa; nos falta el gesto que las une.

Ese gesto es el tema de este capítulo. Para que las siete coordenadas dejen de ser cajones de un archivador y empiecen a contar la historia de la venta de Marco, necesitamos una unidad mínima de afirmación: la pieza más pequeña con la que se puede decir *algo* sobre el mundo. La llamaremos el **hecho atómico**, y su forma es desconcertantemente simple.

La forma de un hecho

Un hecho atómico es siempre una frase de exactamente tres partes (una *tripleta*): un sujeto, un predicado y un objeto.

Se lee de izquierda a derecha como una oración del castellano: «la venta 001 tiene como agente al vendedor 17». El **sujeto** y el **objeto** son individuos que sacamos de alguna de las cajas de valor (un evento del eje O, una persona del eje Q, un número del eje N). El **predicado** es el cable conector que vimos en el capítulo anterior: un habitante del eje M, el *cómo*, que une los dos extremos. Y eso es todo. No hay un cuarto campo escondido, ni una nota al pie, ni una excepción para los casos difíciles.

La afirmación que sostiene este capítulo (y que al principio cuesta creer) es esta: **absolutamente toda la información que el sistema sabe del mundo se construye con esta misma forma exacta**, repetida una y otra vez. Las ventas, las jugadas de fútbol, las películas, las ordenanzas de un municipio, las llamadas a un agente de inteligencia artificial. Todo. El resto del capítulo se dedica a demostrarlo y, sobre todo, a explicar por qué esa terquedad por una sola forma es justamente lo que hace que el sistema no se rompa nunca.

> **CONFESIÓN**
> Al empezar el libro yo solo tenía la intuición de que las siete preguntas podían modelar la información como un espacio de coordenadas (eje contra eje). La tripleta no estaba en el plano inicial: apareció al someter el modelo a las pruebas de laboratorio que veremos en la Parte V. Es un hallazgo, no un punto de partida.

La misma forma viaja entre dominios sin deformarse. Mira cinco hechos sacados de cinco industrias que no tienen nada en común. Cambian los individuos, cambian los cables; la estructura, jamás.

TRIPLETAS

```
(venta_001,      vendedor,       vendedor_17)          ∈ M(O, Q)   👕
(gol_001,        agente,         messi)                ∈ M(O, Q)   ⚽
(pelicula_marea, directora,      serra)                ∈ M(O, Q)   🎬
(ordenanza_142,  fecha_publicacion, 2026-05-14)        ∈ M(O, T)   🏛️
(sesion_ia_5521, tokens_entrada, 4180)                 ∈ M(O, N)   🤖
```

Cinco cables distintos (`vendedor`, `agente`, `directora`, `fecha_publicacion`, `tokens_entrada`) y cinco pares de cajas distintos. Pero las cinco líneas son, estructuralmente, la misma línea. Esa uniformidad no es un capricho estético para que el código se vea ordenado: es la propiedad arquitectónica de la que cuelga todo lo demás. Vale la pena, entonces, escribirla en piedra.

La decisión D3

Ya tomamos dos decisiones de diseño en los capítulos previos (D1 y D2). La que sigue es, con diferencia, la más pesada del libro, porque el resto de la arquitectura se apoya literalmente sobre ella.

> **D3** **EL ÁTOMO DE LA INFORMACIÓN**
>
> Cualquier hecho de la realidad se representa siempre como una **tripleta atómica** de la forma `(sujeto, cable, objeto)`. El cable debe estar **tipado** (saber qué cajas conecta) para que el sistema pueda validar si el dato es lógico o absurdo. Las descripciones de cosas complejas **jamás** se construyen inventando estructuras nuevas ni tablas más anchas: se construyen **apilando** decenas de estas tripletas simples sobre un mismo sujeto.

Léela despacio, porque tiene dos prohibiciones tan importantes como sus permisos. La primera: un cable sin tipo no es un cable válido. La segunda: ante un dato más rico, la respuesta nunca es «cambiemos la estructura», sino «agreguemos más tripletas». Todo lo que sigue es la consecuencia directa de habernos atrevido a tomar esta decisión.

Tres exigencias que el átomo debe cumplir

Para que esta unidad mínima funcione como cimiento de cualquier base de datos del mundo, debe satisfacer tres exigencias estrictas. No son adornos: si fallara cualquiera de las tres, el modelo se vendría abajo.

Exigencia 1 — Tipada: protección contra el caos

Cada cable lleva pegada su propia regla de seguridad, su **signatura**: la declaración de qué caja admite a la izquierda y qué caja admite a la derecha. El cable `monto_usd` sabe de fábrica que conecta un objeto del eje O con un número del eje N. Esa es exactamente la parte `∈ M(O, N)` que aparece al final de cada línea.

TRIPLETAS

```
(venta_001, monto_usd, 49.90)      ∈ M(O, N)   // válido: 49.90 vive en la caja N
(venta_001, monto_usd, "caro")     ✗ rechazada  // "caro" no es un número
```

Si un programador cansado, a las dos de la tarde, intenta registrar `(venta_001, monto_usd, "caro")`, el sistema lo rechaza al instante: la palabra «caro» no habita la caja de los números. Esta protección, que ocurre en tiempo real, es lo que convierte la tripleta en un bloque de información *seguro* y *validable*. Un dato absurdo no llega siquiera a guardarse.

Exigencia 2 — Independiente: cada hecho se sostiene solo

Cada hecho atómico debe tener sentido por sí mismo, sin depender de ningún otro. La tripleta `(gol_001, agente, messi)` significa exactamente lo mismo aunque nadie haya anotado todavía en qué minuto del partido ocurrió, o con qué resultado quedó el

marcador. Quitar contexto no la corrompe; solo la deja menos detallada.

Esa independencia es un superpoder silencioso: permite guardar el hecho en un archivo de texto, en una base de datos relacional o enviarlo por internet a otro continente sin que pierda una gota de su significado. El átomo es portátil porque no arrastra dependencias ocultas.

Exigencia 3 — Componible: los átomos se apilan

Aunque cada hecho es independiente, su verdadera potencia se libera cuando los apilas sobre un mismo sujeto. Una venta no se describe en una frase gigante; se describe con una lluvia de átomos que comparten a `venta_001` como sujeto, cada uno aportando una coordenada distinta.

TRIPLETAS

```
(venta_001, instancia_de,  venta)            ∈ M(O, K)
(venta_001, vendedor,      vendedor_17)      ∈ M(O, Q)
(venta_001, cliente,       cliente_1042)     ∈ M(O, Q)
(venta_001, objeto,        camiseta_88)      ∈ M(O, O)
(venta_001, talla,         talla_m)          ∈ M(O, K)
(venta_001, color,         azul)             ∈ M(O, K)
(venta_001, cantidad,      1)                ∈ M(O, N)
(venta_001, monto_usd,     49.90)            ∈ M(O, N)
(venta_001, medio_pago,    tarjeta)          ∈ M(O, K)
(venta_001, momento,       2026-05-14T16:32:00-05:00)  ∈ M(O, T)
```

Diez pequeñas piezas de Lego que, juntas, dibujan la operación entera: qué se vendió, quién la atendió, a qué cliente, en qué talla y color, cuántas unidades, por cuánto, con qué medio de pago y a qué hora. Cada pieza se entiende por separado; sumadas, forman el retrato completo de la venta.

Lo verdaderamente revelador llega cuando el negocio quiere ser más preciso. Supongamos que la tienda decide auditar el lote de inventario del que salió cada artículo, con su fecha de ingreso. En una base de datos tradicional esto obligaría a alterar la tabla de ventas, añadirle columnas y migrar miles de filas (tocar la estructura, con todo el riesgo que eso implica). En nuestra arquitectura, **la estructura no cambia jamás**. Simplemente se eleva la entrega del artículo a la categoría de evento (algo que ya aprendimos a llamar *reificación*) y se le cuelgan sus propias tripletas:

TRIPLETAS

```
(entrega_001, parte_de,    venta_001)        ∈ M(O, O)
(entrega_001, articulo,    camiseta_88)      ∈ M(O, O)
(entrega_001, lote,        lote_2026_044)    ∈ M(O, K)
(entrega_001, cantidad,    1)                ∈ M(O, N)
(entrega_001, ingresado_el, 2026-05-02)      ∈ M(O, T)
```

IDEA CLAVE

El sistema escala **acumulando bloques de información**, no obligando a nadie a rediseñar la base de datos. Más detalle siempre significa *más tripletas*, nunca *otra estructura*. Esta es, quizá, la ventaja más práctica de todo el modelo.

El universo de datos es una pizarra de tripletas

Cuando una empresa empieza a guardar miles (y luego millones) de estos hechos atómicos, va tejiéndose una red inmensa de nodos conectados por cables. A esa red se le llama un **grafo**. Y conviene precisar qué es, técnicamente, una base de datos WQuestions: no es más que **un conjunto gigantesco de oraciones de tres partes escritas en una pizarra común**. Ni más, ni menos.

> *El estado entero de la realidad, para el sistema, se reduce a contar cables sobre una pizarra. No hay archivos ocultos ni arquitecturas secretas que solo entiendan los creadores del software.*
>
> EL PRINCIPIO DE LA PIZARRA

Esa transparencia es liberadora. Si le preguntas a la máquina «¿quién dirigió la película *Marea*?», no hay misterio que resolver: solo busca cables que arranquen en `pelicula_marea` y lleven la etiqueta `directora`. Y si pides algo más ambicioso («dame todas las directoras de largometrajes estrenados después de 2020»), el sistema hace un trabajo tonto pero velocísimo: localiza los cables `año_estreno` cuyo número final supere 2020, mira a qué películas están pegados, y rastrea quién figura como su `directora` en cada una. Cada paso es, simplemente, saltar de una tripleta a la siguiente. Fin del enigma.

Cuando tres palabras no alcanzan: la reificación

A veces la realidad se niega a caber en una sola tripleta. Piensa en esta crónica deportiva, de las que se narran en una sola respiración:

> *«Messi le pasó el balón a Di María en el minuto 87 con un toque de pierna izquierda.»*

Esa frase es una veta de oro: tiene un autor (Messi), un beneficiario (Di María), un objeto (el balón), un momento (minuto 87) y un instrumento (la pierna izquierda). Si quisiéramos meterla a la fuerza en una tripleta simple (`(messi, paso_el_balon_a, di_maria)`) estaríamos tirando a la basura casi toda la información. ¿Cómo lo resolvemos sin violar la D3?

Con la **reificación**, que ya asomó en capítulos anteriores. La maniobra es elegante: elevamos ese pase al estatus de *evento con nombre propio* dentro del eje O (lo bautizamos `pase_001`) y colgamos a cada participante con su propia tripleta, todas apuntando a ese evento central.

TRIPLETAS

```
(pase_001, instancia_de,  accion_pasar)          ∈ M(O, K)
(pase_001, agente,        messi)                 ∈ M(O, Q)
(pase_001, beneficiario,  di_maria)              ∈ M(O, Q)
(pase_001, objeto_pase,   balon_partido_001)     ∈ M(O, O)
(pase_001, minuto,        87)                    ∈ M(O, N)
(pase_001, instrumento,   pierna_izquierda)      ∈ M(O, K)
```

Lo que era un nudo lingüístico imposible de meter en una fila de tabla se desarmó en seis hechos atómicos independientes, cada uno tipado, cada uno sostenido por sí solo. La estructura n-aria (un evento con muchos participantes) se preservó entera sin abandonar la forma de la tripleta. La figura siguiente muestra cómo se ve esa estrella de hechos alrededor del evento central.

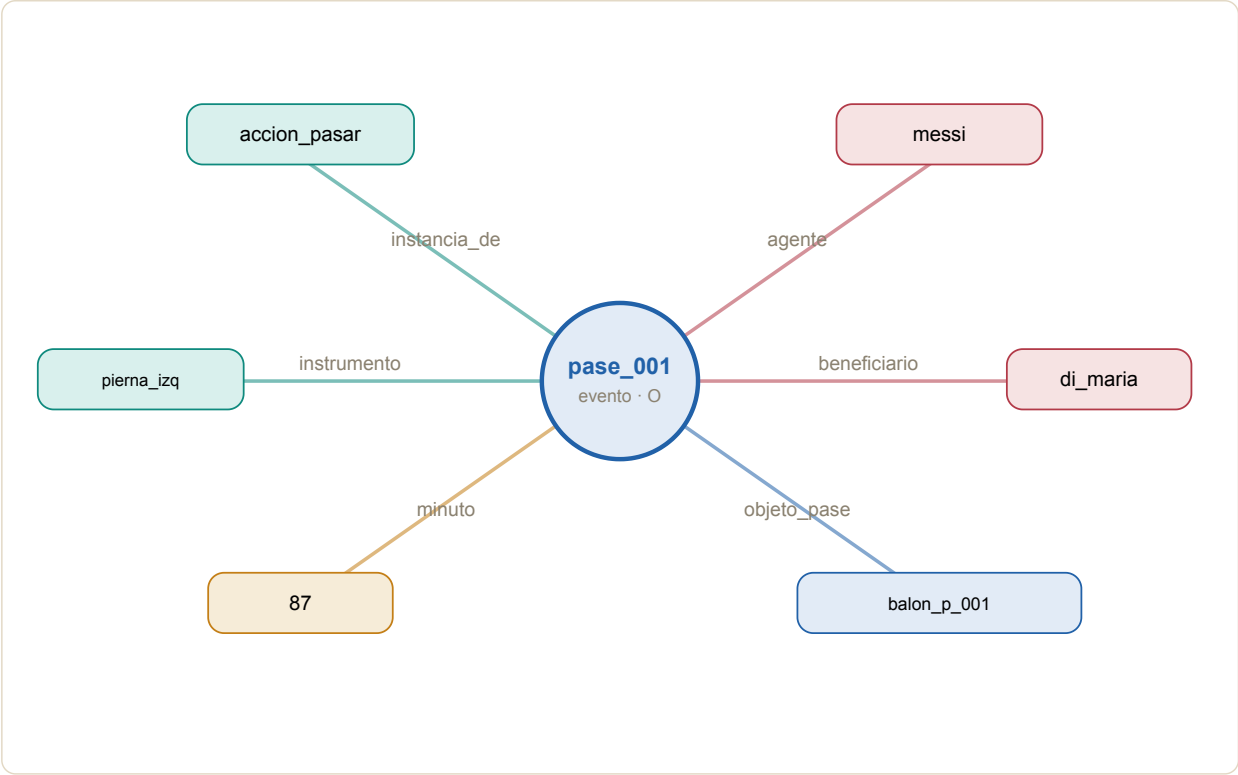


Figura 7.1. El pase de Messi, reificado como el individuo `pase_001` en el eje O, con sus seis participantes colgados por roles canónicos (*agente*, *beneficiario*, *objeto_pase*, *minuto*, *instrumento*, *instancia_de*). La estructura n-aria se conserva entera sin abandonar la forma de la tripleta.

Conviene una regla práctica, de las que un ingeniero de datos hace bien en tatuarse: **reifica solo cuando te falte espacio**. Si la relación es entre dos individuos y nada más (`(pelicula_marea, directora, serra)`), basta una tripleta directa. Si el hecho involucra a más participantes o arrastra detalles propios (tiempo, lugar, instrumento), entonces sí: elévalo a evento con nombre propio y cuélgale sus tripletas. Ni más reificación de la necesaria, ni menos de la que el detalle exige.

El modelo y el lenguaje humano son gemelos

Aquí ocurre algo que, la primera vez que lo notas, parece magia, y conviene resaltarlo de inmediato. Toma un artículo de prensa cualquiera y pásaselo a un modelo de lenguaje moderno (Claude, por ejemplo) con esta instrucción: «Resume paso a paso quién hizo qué en esta noticia». La respuesta casi siempre vuelve como una lista de líneas limpias, sin que nadie se lo haya pedido en ese formato:

LO QUE LA IA DEVUELVE, SOLA

El alcalde Reyes promulgó la ordenanza de micromovilidad.

La ordenanza entra en vigor a los 30 días de su publicación.

La medida afecta a la gerencia de tránsito.

La medida afecta a la gerencia de fiscalización.

Cuatro líneas, cuatro hechos. Y cada línea tiene su sujeto, su verbo y su objeto. La IA no responde así porque haya leído este libro: responde así porque **así funciona el lenguaje humano cuando intenta describir la realidad**. Las oraciones simples que nos enseñaron en la escuela (sujeto, verbo, predicado) son idénticas, molécula a molécula, a nuestros hechos atómicos tipados.

El impacto de esta coincidencia es enorme. Si tu base de datos guarda la información como tripletas, y la inteligencia artificial *piensa* en tripletas, entonces la comunicación entre la IA y tu empresa se vuelve casi automática. No hace falta programar costosos algoritmos de comprensión del lenguaje natural ni levantar capas de traducción frágiles: existe un enchufe directo entre lo que la IA dice y lo que tu sistema guarda. Esa es la llave maestra del agente que el mercado persigue (uno capaz de conversar, auditar y

registrar información en cualquier empresa) sin obligarlo a memorizar la estructura interna de la base de datos de cada cliente. Es el tema que el capítulo 26 lleva hasta el fondo.

Consultar es tapar un hueco

Si toda la información se guarda como tripletas, buscar en ella se vuelve un juego de niños. Hacer una consulta es, literalmente, escribir una tripleta pero **tapando uno de sus tres campos con un signo de interrogación** (dejando un hueco que el sistema debe rellenar). Mira tres patrones de búsqueda sobre nuestros ejemplos:

PATRONES

```
(serra,          ?,          ?)        // "Dime todo lo que sepas de Serra"
(?,              directora,  serra)     // "¿Qué obras dirigió Serra?"
(?,              agente,     messi)     // "Busca todas las jugadas de Messi"
```

Y si necesitas una consulta digna de un analista, no hay que aprender un lenguaje nuevo: basta encadenar varias «tripletas con huecos» y pedirle al sistema los datos que encajen en *todas* a la vez. Las variables que se repiten (las que empiezan con `?`) son las junturas: amarran un patrón con el siguiente.

PATRONES

```
(?gol,     agente,   messi)
(?gol,     parte_de, ?partido)
(?partido, fecha,    [2026-01-01, 2026-12-31])
// → "Todos los goles de Messi en partidos del año 2026"
```

Así de simple. No hacen falta diagramas enrevesados ni consultas eternas: **consultar es decirle a la máquina que busque un patrón de cables dentro de la red**. Y la operación matemática es idéntica en cualquier industria y a cualquier escala: exactamente lo que la vieja torre de Babel de los datos, que vimos en el capítulo 1, nos impedía hacer.

Un cable repetido es una columna

Apilar tripletas produce un efecto lateral que se ve mejor desde ya. Cuando mil ventas distintas llevan todas el mismo cable `monto_usd`, ese cable deja de ser un dato suelto de `venta_001` y se vuelve, de hecho, una columna que se puede recorrer entera. Lo mismo pasa con `agente`: escribirlo sobre cada gol es, sin proponérselo, dejar armada la lista de todas las jugadas que comparten ese cable. Por eso el patrón `(?, agente, messi)` de hace un momento no era un truco aislado, sino la forma natural de leer esa columna de punta a punta. Cada hecho que registras sobre un sujeto es, a la vez, una fila en el reporte de todos los sujetos que comparten ese cable: lo que parece guardar un dato individual está construyendo, sin avisar, la semilla de la consulta colectiva.

El contraste con SQL

Para medir la diferencia, traduzcamos esa última pregunta (*goles de Messi en 2026*) al mundo relacional clásico. Allí los hechos no viven en una pizarra común, sino repartidos en tablas que muchas veces ni siquiera se diseñaron para conversar. Recuperar el dato obliga a coserlas con `JOIN`s, una operación que el grafo no necesita porque nunca separó la información en primer lugar.

SQL

```
SELECT j.nombre, g.minuto
FROM goles g
JOIN jugadores j ON j.id = g.jugador_id
```

```
JOIN partidos  p ON p.id = g.partido_id
WHERE j.nombre = 'Messi'
  AND p.fecha BETWEEN '2026-01-01' AND '2026-12-31';
```

No es que el SQL esté mal escrito; es que carga con un trabajo que el modelo de tripletas ni siquiera tiene. Cada `JOIN` es la cicatriz de una decisión previa de partir el mundo en tablas; y si mañana el negocio quiere registrar el asistente de cada gol, habrá que alterar el esquema. En la pizarra de tripletas, recordemos, eso era una línea más.

El mismo átomo, viajando por el cable

Cuando un hecho necesita salir del sistema y viajar a otro (una API, un agente, otra empresa), se serializa sin perder un solo eje. La **reificación se vuelve un objeto JSON** casi sin esfuerzo: el sujeto es el `id`, cada cable es una clave y cada objeto es el valor. Así se ve una venta sobre el cable:

JSON

```
{
  "id": "venta_74921",
  "instancia_de": "venta",
  "agente": "vendedor_17",
  "cliente": "cliente_1042",
  "objeto": "camiseta_88",
  "monto": { "valor": 49.90, "unidad": "Currency:USD" },
  "momento": "2026-05-14T16:32:00-05:00"
}
```

Cada clave es un cable; cada valor, un individuo en su eje. El mismo átomo que dibujamos como una estrella de nodos en la Figura 7.1 es, sobre el cable, un objeto que cualquier sistema (o cualquier IA) lee sin manual de instrucciones. La figura siguiente cierra el círculo: una sola afirmación, vista de tres maneras que son la misma.

Figura 7.2. El hecho «la venta 001 tiene un monto de 49.90 dólares» expresado como tripleta, como arista de un grafo y como par clave-valor en JSON. Las tres notaciones son intercambiables: el modelo no privilegia ninguna, porque todas describen la misma coordenada.

El repertorio entero, pasado por el átomo

Para cerrar con los pies en la tierra, recorramos los cinco escenarios de este capítulo y veámoslos descompuestos en hechos atómicos. La lección, repetida cinco veces, es siempre la misma: basta apilar tripletas sobre un sujeto, y reificar solo cuando un evento lo pide.

👕 **La venta.** Ya la desmenuzamos: diez tripletas bastan para el retrato general de la operación, y si la tienda exige trazar el lote de inventario, se reifica la entrega del artículo sumando cinco tripletas más (sin tocar el diseño base).

⚽ **El gol.** El fútbol se describe apilando roles tácticos sobre el evento. Seis hechos atómicos conservan al autor, al asistente, el partido, el minuto y la pierna. Suficiente detalle para una casa de apuestas o un analista de rendimiento, sin reducir la jugada a una fila gris de hoja de cálculo:

TRIPLETAS

```
(gol_001, instancia_de, gol_jugada_abierta)    ∈ M(0, K)
(gol_001, agente,       messi)                 ∈ M(0, Q)
(gol_001, asistente,    di_maria)              ∈ M(0, Q)
(gol_001, parte_de,     partido_arg_per_2026)  ∈ M(0, O)
(gol_001, minuto,       87)                    ∈ M(0, N)
(gol_001, pierna,       zurda)                 ∈ M(0, K)
```

🎬 **La película.** Una obra cinematográfica se describe apilando autoría (que tiene roles distintos: dirigir no es escribir) y composición recursiva, porque una escena es «parte de» la película:

TRIPLETAS

```
(pelicula_marea, instancia_de, largometraje)         ∈ M(0, K)
(pelicula_marea, genero,       genero_drama)         ∈ M(0, K)
(pelicula_marea, directora,    serra)                ∈ M(0, Q)
(pelicula_marea, guionista,    haddad)               ∈ M(0, Q)
(pelicula_marea, año_estreno,  2026)                 ∈ M(0, N)
(escena_42,      parte_de,     pelicula_marea)       ∈ M(0, O)
(escena_42,      tiempo_narrativo, fin_segundo_acto) ∈ M(0, K)
```

Siete hechos en los que `directora` y `guionista` son cables distintos (Serra y Haddad cumplen papeles que no se confunden) y donde `escena_42` se cuelga de la película por `parte_de`, abriendo la puerta a describir cada escena con su propia lluvia de tripletas. La composición no tiene fondo.

🏛️ **La ordenanza.** Un acto de gobierno combina fechas, vigencias y entidades afectadas. Nótese que `afecta_a` es un cable *múltiple*: el mismo sujeto lo usa varias veces, una por cada gerencia tocada.

TRIPLETAS

```
(ordenanza_142, instancia_de,     ordenanza_municipal)   ∈ M(0, K)
(ordenanza_142, promulgada_por,   alcalde_reyes)         ∈ M(0, Q)
(ordenanza_142, organo,           municipalidad_centro)  ∈ M(0, Q)
(ordenanza_142, fecha_publicacion, 2026-05-14)           ∈ M(0, T)
(ordenanza_142, entra_en_vigor,   2026-06-13)            ∈ M(0, T)
(ordenanza_142, afecta_a,         gerencia_transito)     ∈ M(0, Q)
(ordenanza_142, afecta_a,         gerencia_fiscalizacion) ∈ M(0, Q)
```

La fecha de entrada en vigor (30 días después de la publicación) no se inventa a mano: es un *tiempo derivado por reglas*, un tema que el capítulo siguiente y los de la Parte V exploran. Si alguien pregunta «¿qué normas promulgó el alcalde Reyes este año?», la máquina cruza las primeras dos tripletas y responde sin demora.

🤖 **El agente de IA.** Y el dominio más moderno de todos: la telemetría de una sesión con un modelo de lenguaje. Diez hechos atómicos que registran qué usuaria preguntó, qué modelo respondió, cuántos tokens costó y qué herramientas tuvo que invocar el agente.

TRIPLETAS

```
(sesion_ia_5521, instancia_de,   sesion_modelo_lenguaje)  ∈ M(O, K)
(sesion_ia_5521, modelo,         modelo_lumen_2026)       ∈ M(O, K)
(sesion_ia_5521, usuaria,        paredes)                 ∈ M(O, Q)
(sesion_ia_5521, tokens_entrada, 4180)                    ∈ M(O, N)
(sesion_ia_5521, tokens_salida,  920)                     ∈ M(O, N)
(sesion_ia_5521, latencia_ms,    2100)                    ∈ M(O, N)
(sesion_ia_5521, costo_usd,      0.015)                   ∈ M(O, N)
(sesion_ia_5521, temperatura,    0.7)                     ∈ M(O, N)
(sesion_ia_5521, herramienta,    busqueda_web)            ∈ M(O, K)
(sesion_ia_5521, herramienta,    consulta_grafo)          ∈ M(O, K)
```

Toda la telemetría que un ingeniero moderno exige (quién, qué modelo, cuánto costó, si tuvo que buscar en la web o consultar el propio grafo) vive en diez líneas tipadas. Es incomparablemente más limpio, y más consultable, que un volcado de logs sin estructura.

EN LA PRÁCTICA

Fíjate en el patrón que se repite en los cinco dominios: la primera tripleta casi siempre es `instancia_de` (ancla el sujeto en su clase del eje K), y a partir de ahí cuelgan los atributos. Cuando empieces a modelar un dominio propio, abre por ahí: nombra el sujeto, declara su clase y deja que las coordenadas vayan cayendo, una tripleta cada vez.

Lo que llevamos, y lo que viene

El hecho atómico es la piedra fundacional de toda la arquitectura. Tiene siempre la misma forma rígida de tres partes (`(sujeto, cable, objeto)`); lleva protección interna que impide mezclar cosas absurdas; y se compone consigo mismo para describir realidades tan complejas como queramos, sin pedir jamás un cambio de programación. En este modelo, la base de datos es, sin metáfora, un océano de estos hechos.

Lo hermoso del hallazgo no es haber inventado un código nuevo, sino haber descubierto que esta forma matemática coincide al milímetro con la manera en que los seres humanos (y la inteligencia artificial) hablan cuando describen el mundo. El modelo no obliga a nadie a aprender un idioma robótico: toma la estructura natural del lenguaje y la convierte en tecnología consultable.

Hasta aquí hemos tratado cada hecho como una pieza solitaria. Pero cuando se amontonan por millones, emerge algo nuevo: un **espacio geométrico**. No una metáfora, sino una herramienta matemática real, donde consultar masivamente equivale a poner restricciones sobre coordenadas (como jugar a la batalla naval a escala planetaria). Ese espacio es el tema del próximo capítulo.

08

El espacio multidimensional

Las coordenadas no eran una figura retórica. Cuando un hecho se vuelve un punto y una consulta se vuelve un corte, lo que parecía una imagen amable resulta ser una geometría con leyes propias: parcial, multivaluada y tipada.

Abre una hoja de cálculo y reserva siete columnas, una por cada coordenada: **quién, qué, dónde, cuándo, cuánto, cuál** y la columna de enlaces que las amarra, **cómo**. Ahora pega, en filas sucesivas, hechos que no tienen nada que ver entre sí: una venta que se registró a media tarde, un gol anotado en el segundo tiempo, el estreno de un largometraje, una ordenanza municipal de micromovilidad, la llamada de una usuaria a un agente de inteligencia artificial. Cinco mundos ajenos, una sola rejilla. Y, fila tras fila, la mayoría de las celdas en blanco. Esa hoja medio vacía, lejos de ser un borrador descuidado, es la representación más honesta de lo que vamos a estudiar.

> **DE DÓNDE VENIMOS**
> En el capítulo anterior un hecho dejó de ser una fila para volverse un evento atómico con varias lecturas. Aquí damos el paso siguiente: si cada hecho es un punto, el conjunto de todos los hechos es un *espacio* con propiedades formales.

Esa rejilla, llevada a su conclusión natural, es lo que llamaremos el **espacio multidimensional** de WQuestions. Tiene un eje por cada coordenada de valor (seis dimensiones que fijan posición) y una urdimbre de enlaces que une las lecturas de cada hecho. Los sucesos del mundo son puntos dentro de ese espacio. Habrá regiones congestionadas, donde los hechos llenan casi todos sus ejes, y regiones casi desiertas. Pero todos, sin excepción, obedecen a la misma física geométrica. Este capítulo trata de esa física. No la estudiamos por afición a las matemáticas, sino porque pensar la base de datos como un espacio cambia, de raíz, cómo se escribe una consulta y cómo un sistema automático audita lo que una organización sabe de sí misma.

De la metáfora a la geometría

Conviene decirlo sin rodeos, porque es la tesis del capítulo: **las coordenadas no son una metáfora**. Cuando en los capítulos anteriores dijimos que un hecho «se ubica» en los ejes de las preguntas, no estábamos adornando una idea con vocabulario espacial prestado. Estábamos describiendo, con precisión, un espacio que admite definición formal y que se comporta como tal. Si nos pusiéramos estrictos, diríamos que el universo de valores (llamémoslo `V`) es la unión de todos los individuos que viven en los seis ejes de valor:

NOTACIÓN

```
V  =  Q ∪ O ∪ L ∪ T ∪ N ∪ K          (el universo de valores)
S  ⊆  Q × O × L × T × N × K          (una situación: un punto, posiblemente parcial)
M                                     (los enlaces que conectan cada lectura del punto)
```

Una *situación* es un punto de ese producto: una tupla con, a lo sumo, una lectura por eje. Y el «a lo sumo» es el primer detalle que separa este espacio de los que verás en un libro de cálculo. Pero antes de las diferencias, fijemos la intuición con un punto concreto. Toma la venta que el vendedor 17 registró a las cuatro y media de la tarde: no es una abstracción, es un punto con coordenadas legibles, una por eje, unidas por los enlaces de *cómo*.

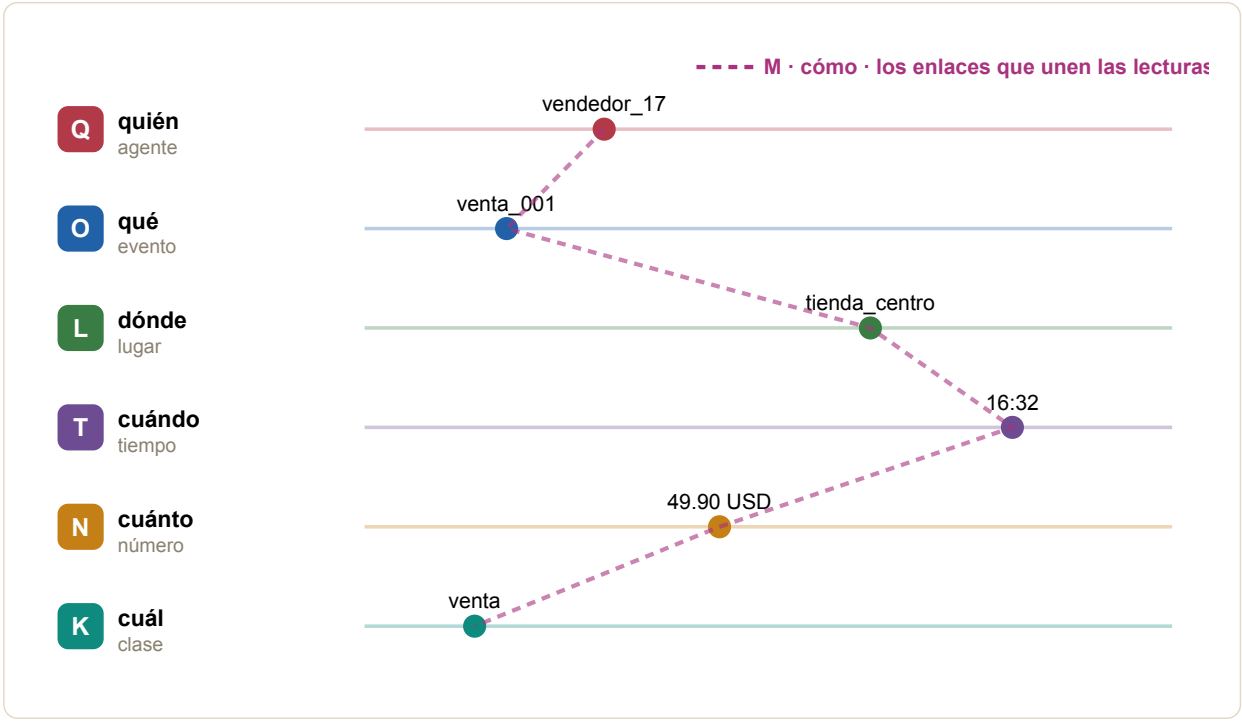


Figura 8.1. La situación `venta_001` leída como un punto en coordenadas paralelas: una lectura por eje de valor. La línea magenta no es un séptimo eje de posición, sino la urdimbre de enlaces (*cómo*) que ata cada coordenada al mismo hecho. Quitar un punto de un eje no rompe el dibujo: deja un eje sin lectura.

Ahora bien, en la inmensa mayoría de los puntos no habrá una lectura en cada eje. Y eso es lo esperable, no un defecto. El gol de Messi toca Q, O y T; la llamada al agente de IA, por ocurrir «en la nube», deja vacío el eje L. Por eso conviene pensar el espacio como una **estructura parcial**: cada hecho ocupa solo las dimensiones que le importan y deja en paz las demás. La promesa del modelo nunca fue que todo en la vida sea hipercomplejo. Fue que *cualquier* hecho, por excéntrico que sea, puede encontrar su lugar exacto en este mapa de coordenadas.

Tres propiedades formales

Si trabajas con datos, la palabra «espacio multidimensional» evocará territorios conocidos: el espacio vectorial $\mathbb{R}^n$ del álgebra, las tablas relacionales, los cubos OLAP del *business intelligence*. Comparten la idea de varios ejes. Pero el espacio de WQuestions difiere de todos ellos en tres propiedades que conviene nombrar con precisión, porque cada una desactiva una incomodidad clásica del modelado de datos.

1 PARCIAL

En $\mathbb{R}^n$ todo punto tiene un valor obligatorio en cada eje: no existe un punto «sin coordenada Y». Aquí, un eje sin lectura simplemente *no aplica* para ese hecho. No hace falta inventar un `NULL` ni un «desconocido» de relleno para que la estructura no se rompa: la ausencia es un estado legítimo.

2 MULTIVALUADA

En geometría clásica un punto tiene una sola coordenada X. Aquí, gracias a los enlaces de *cómo*, un mismo hecho puede proyectar *varias* lecturas sobre el mismo eje. Un partido tiene dos equipos; una llamada al agente pudo apoyarse en cinco fuentes. En $\mathbb{R}^n$ eso sería ilegal; en datos reales es obligatorio.

3 TIPADA

En un gráfico escolar todos los ejes son la misma recta de números. Aquí cada eje es de una especie distinta: Q guarda agentes, T instantes, N magnitudes, K conceptos. El espacio sabe que sumar una persona con una fecha es un sinsentido, y protege la base contra ese cruce.

La tercera propiedad merece una observación que el resto del libro aprovechará. Que el espacio sea **tipado** no es un detalle decorativo: es lo que permite que las consultas se validen antes de ejecutarse. Si alguien pide «hechos donde Q sea anterior a 2026», el motor puede rechazar la pregunta sin tocar un solo dato, porque sabe que en el eje de los agentes la relación «anterior a» carece de sentido. La geometría no solo almacena: además, defiende su propia coherencia.

IDEA CLAVE

Un espacio **parcial, multivaluado y tipado** es exactamente lo que el mundo necesitaba y lo que $\mathbb{R}^n$ se negaba a dar. La parcialidad admite la ignorancia honesta; la multivaluación admite la pluralidad real; el tipado admite la diferencia de naturalezas. Las tres juntas convierten una rejilla medio vacía en una herramienta de precisión.

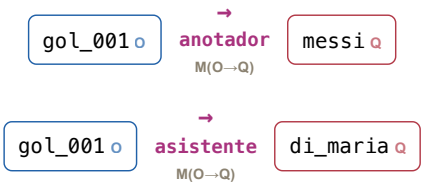
Las dimensiones son los roles, no los ejes

Aquí cabe deshacer una confusión que el lenguaje de «ejes» arrastra. Cuando decimos que el espacio tiene seis dimensiones de valor, podría parecer que la dimensionalidad de un hecho es, como mucho, seis. No es así. Lo que da dimensión a una situación no son los seis ejes, sino los **roles** que cada eje puede desplegar dentro de ese hecho. Un gol de fútbol no tiene «un quién»: tiene un anotador y un asistente, dos roles distintos sobre el mismo eje Q. Un largometraje no tiene «un autor»: tiene una directora y una guionista. Cada rol es una dimensión propia.

SUTILEZA

Esto es lo que la multivaluación significa en serio: no «varios valores apilados sin orden», sino *varios roles tipados* sobre un mismo eje, cada uno con su firma `M`. El eje aporta la especie; el rol, la posición concreta dentro del hecho.

Míralo en el gol de Messi. El eje Q aparece dos veces, pero no como duplicado: como dos coordenadas con roles distintos, cada una con su propio enlace de *cómo*.



Por eso la dimensionalidad real de un hecho es el número de roles que activa, no el número de ejes que existen. El espacio de seis ejes es, en rigor, un espacio de dimensión abierta: cada situación reificada puede abrir tantos roles como su naturaleza exija, y todos aterrizan, tipados, sobre los mismos seis ejes de valor. Los ejes son el alfabeto; los roles, las palabras.

La hoja dispersa, en acción

Volvamos a la rejilla del principio, ahora con datos del repertorio. Cinco situaciones de cinco dominios sin parentesco alguno, conviviendo en la misma estructura. Observa, sobre todo, las celdas en blanco: no son un descuido, son la parcialidad funcionando.

LA SITUACIÓN	Q QUIÉN	O QUÉ	L DÓNDE	T CUÁNDO	N CUÁNTO	K CUÁL
Venta registrada	`vendedor_17`	`venta_001`	`tienda_centro`	16:32	49.90 USD	`venta`
Gol marcado	`messi`	`gol_001`	`estadio_lima`	min. 87	1 gol	`gol`
Estreno de cine	`serra`	`pelicula_marea`	—	2026-09-12	—	`largometraje`
Ordenanza publicada	`alcalde_reyes`	`ordenanza_142`	`municipio_centro`	2026-05-14	—	`ordenanza_municipal`
Llamada a la IA	—	`sesion_ia_5521`	—	10:32	4180 tok.	`modelo_lumen_2026`

Figura 8.2. La hoja dispersa: cinco situaciones de cinco dominios ajenos en una sola estructura. El estreno no tiene lugar fijo ni magnitud; la llamada al agente no tiene agente humano ni lugar físico. Las celdas vacías son la norma, no la excepción: el espacio es parcial por diseño.

Lo notable no es que las cinco filas quepan en la misma tabla, sino que **la estructura sea idéntica para todas**. Cualquier analista (y cualquier modelo de lenguaje) puede leer esta rejilla y entender qué dice cada fila sin ser experto en ventas, en fútbol ni en derecho administrativo. Y esa uniformidad es lo que vuelve trivial la búsqueda. Si alguien pide «todo lo que ocurrió el 14 de mayo de 2026», basta filtrar el eje T: el sistema devolverá, en el mismo gesto, la ordenanza de la municipalidad. No hace falta tender un puente entre el servidor de la tienda, el de deportes y el legal. Todo vive en la misma geometría.

Consultar es restringir la geometría

Si los hechos son puntos en un espacio de seis ejes, entonces una consulta no es más que **imponer restricciones geométricas sobre las coordenadas**. Fijar un eje a un valor es trazar un hiperplano; pedir un rango es recortar una franja; quedarse con unos pocos ejes es una proyección. En la práctica, casi todo lo que una organización pregunta cae en uno de tres gestos.

Fijar un punto y soltar el resto. «Muéstrame todo lo que involucre al vendedor 17.» El motor clava una estaca en Q `= vendedor_17` y recoge todos los hechos que lo tocan, sin importar el día, la tienda ni el producto. Geométricamente: nos quedamos con el hiperplano que pasa por ese valor y es libre en todos los demás ejes.

Acotar un intervalo. «Dame los eventos entre el 1 y el 31 de mayo.» El motor apoya una regla de medición sobre el eje T y descarta cuanto caiga fuera de la franja, dejando libres a las personas y los lugares. Esto solo es posible porque el eje está tipado: «entre dos fechas» tiene sentido en T, no en Q.

Cruzar varios cortes. «Todos los goles que Messi anotó en el segundo tiempo.» El motor corta por Q `= messi`, luego por K `= gol`, luego por T `= segundo tiempo`. El resultado es el rincón donde los tres cortes se intersecan: una restricción compuesta.

La consulta de cruce, escrita como tripletas, deja ver su forma. No es una pregunta en lenguaje de tablas, sino un conjunto de restricciones sobre un mismo punto:

TRIPLETAS

```
(?evento, anotador,    messi)          ∈ M(O→Q)   ← corta el eje Q
(?evento, clase,       gol)            ∈ M(O→K)   ← corta el eje K
(?evento, tiempo,      segundo)        ∈ M(O→T)   ← corta el eje T
──────────────────────────────────────────────────
?evento  =  la intersección de los tres cortes
```

La belleza técnica está en que el motor ejecuta *siempre la misma operación* (un corte geométrico) sin que le importe si busca goles, diagnósticos o transacciones. Ahí se cumple la promesa del modelo: un único lenguaje para interrogar al mundo entero.

> *Una consulta deja de ser una orden en el dialecto de cada base de datos y pasa a ser lo que siempre quiso ser: una restricción sobre las coordenadas de un hecho.*

LA GEOMETRÍA DE LA PREGUNTA

Situaciones como nube de puntos

Cuando una organización acumula millones de estos puntos parciales, el espacio deja de ser una rejilla y se vuelve una nube. Y en esa nube, las situaciones se ordenan solas según cuántas dimensiones activan y qué magnitud despliegan. La figura siguiente sitúa cinco situaciones del repertorio en un plano de dos ejes elegidos a propósito: cuántas coordenadas llena cada hecho (su densidad) frente a la magnitud numérica que registra. No es una afinidad ni una distancia física (ya volveremos sobre eso); es una manera de *ver* que los puntos no se reparten al azar.

Figura 8.3. Cinco situaciones del repertorio como puntos en un plano de dispersión: en el eje horizontal, cuántas coordenadas llena cada hecho; en el vertical, la magnitud numérica que registra (en escala relativa). Pasa el cursor sobre cada punto para leer la situación. La venta, densa y muy descrita, ocupa la esquina superior derecha; el estreno, parcial y sin magnitud, queda abajo a la izquierda.

Esta es la **densidad emergente**: no una regla matemática, sino un fenómeno orgánico. A medida que el sistema absorbe datos, ciertas regiones del espacio se pueblan mucho más que otras. En un hospital, la zona donde se cruzan pacientes Q, episodios O, fechas T y diagnósticos K se vuelve un nodo brillante de información. Y cuando a ese mismo hospital se le instala un asistente nuevo (un agente que toma citas por mensajería), este empieza a poblar su propio rincón vacío del espacio (sesiones O, latencias N, modelos K) sin estorbar un solo dato del sistema clínico. El día en que una paciente pregunta por mensajería «¿cuándo me toca la consulta?», ambos sistemas se cruzan en un punto del mapa, sin necesidad de tender un túnel de integración entre servidores.

UNA DENSIDAD QUE SE PUEDE AUDITAR

La densidad emergente tiene un uso inesperado: revela puntos ciegos. Si llegas a una organización y su mapa muestra la región de «ventas» densamente poblada pero la de «reclamos» misteriosamente vacía, sabes de inmediato qué parte del negocio no se está registrando, sin leer una línea de documentación. La forma de la nube es un diagnóstico.

De la geometría a la tabla que ya conoces

Toda esta geometría deja a más de un lector con una incomodidad legítima: *está muy bien, pero yo necesito una tabla*. La buena noticia es que no hay que elegir. Una consulta recorta un subconjunto de puntos del espacio; entregar ese recorte a un humano (o a

una hoja de cálculo, o a un reporte) es, sencillamente, **proyectar los ejes a columnas**. La geometría no reemplaza a la tabla: la genera bajo demanda, y en más de una forma.

La primera ya la tienes delante. Esa hoja dispersa de la Figura 8.2 (cinco hechos de cinco dominios sobre los seis ejes) **es** la tabla universal de hechos: una lectura plana, con sus códigos crudos, donde conviven una venta y una ordenanza sin romper nada. Es la vista que querría una máquina. Pero una persona no quiere códigos: quiere nombres, fechas y un encabezado que entienda. Esa es la segunda vista.

Pídele al sistema «un reporte de los trámites de licencias». Por dentro, el motor hace dos cosas: **filtra** por el eje K (se queda con los hechos cuya clase es una licencia) y **resuelve** cada código a su etiqueta legible a través del lexicon: `juan` se vuelve «Juan», `tramite_juan_lic_04` se vuelve «Licencia de funcionamiento», `zona_centro` se vuelve «Centro». El resultado es exactamente la tabla clásica que esperabas:

CIUDADANO Q	TRÁMITE O	UBICACIÓN L	FECHA T	COSTO N	ESTADO M
Juan	Licencia de funcionamiento	Jr. Trujillo 450, Centro	22-06-2026	S/ 450,00	Solicitado
Carla	Licencia de micromovilidad	Av. Perú 1200, Norte	23-06-2026	S/ 300,00	En revisión
Marta	Remodelación de local	Jr. Lima 88, Centro	24-06-2026	S/ 520,00	Aprobada

Figura 8.4. La misma geometría, servida como reporte: se filtra el eje K por la familia de licencias y el lexicon resuelve cada coordenada a su etiqueta. Bajo «Juan» vive `juan`; bajo «Centro», `zona_centro`. La columna M «Estado» no es un eje de posición: es *un enlace* (el cable `estado`) proyectado a propósito como columna, porque es el que este reporte quería ver.

Repara en esa última columna, porque toca un punto fino del modelo. La tabla universal tiene seis columnas de valor, no siete: M es la urdimbre de enlaces, no una coordenada de posición. Pero al proyectar para un humano puedes *elegir* qué enlace subir a columna (aquí, el estado del trámite). Cada reporte decide qué cable de *cómo* le interesa mostrar; el modelo de fondo no cambia.

Y hay una tercera vista, la que más impresiona a quien manda. Si el alcalde pide «cuántos trámites de cada tipo se hicieron por zona», no hace falta un programa nuevo: cruzas dos ejes en una cuadrícula. Las filas son el eje K, las columnas el eje L, y cada celda cuenta los puntos que caen en esa intersección. En SQL eso pediría un `GROUP BY` con `PIVOT`; aquí es la misma operación geométrica de siempre (un corte), contada ahora sobre dos dimensiones:

TIPO DE TRÁMITE K \ ZONA L	CENTRO	NORTE	SUR
Licencias de construcción	45	12	8
Licencias de funcionamiento	120	54	30
Multas de fiscalización	15	42	10

Figura 8.5. Una tabla pivote es solo dos ejes pintados en una matriz: K contra L, con el conteo de puntos en cada cruce. Lo mismo daría cruzar T contra K (trámites por mes y tipo) o cualquier otro par. La rejilla no estaba preconstruida: se arma al vuelo, porque todo vive en la misma geometría.

Tres vistas, un solo almacén: la lectura plana para la máquina, la proyección con nombres para la persona, la pivote para quien decide. Ninguna se diseñó por separado; las tres son cortes y proyecciones del mismo espacio de puntos.

Conviene ponerle nombre a esa pivote, porque es el hogar de un mecanismo que reaparecerá en cada dominio del libro: el reporte. Pivotar es tomar muchas situaciones y agruparlas por dos coordenadas a la vez, y de ahí en adelante toda vez que un hospital pregunte por sus diagnósticos o un banco por su cartera estará volviendo aquí. El caso individual fue la entrada al espacio; el reporte sobre muchos es su salida natural.

PARA EL DESARROLLADOR

Servir esto a un usuario final es menos trabajo del que parece. Tu capa de software hace dos pasos: **(1)** lanza la consulta de coordenadas al grafo y recibe una lista de objetos; **(2)** los formatea como un `DataFrame` de pandas o una tabla HTML, usando los nombres de las preguntas como cabeceras. No hay un esquema de tablas que mantener: las columnas son siempre las mismas siete preguntas, y el contenido cambia según el corte. La vista es código de presentación; el dato vive una sola vez.

> *La tabla nunca fue el modelo. Era una de sus vistas —y el grafo sabe generarla cuando la pides.*

DEL ESPACIO A LA REJILLA

Frente a $\mathbb{R}^n$, las tablas y los cubos

Conviene ahora contrastar este espacio, de frente, con las tres tecnologías multidimensionales que más se le parecen. No para descartarlas (cada una resuelve su problema), sino para fijar el límite exacto donde WQuestions hace algo distinto.

EL ESPACIO VECTORIAL $\mathbb{R}^n$

Ejes homogéneos, puntos totales, distancias y sumas. Sirve para medir y promediar. Pero exige una coordenada en cada eje y trata a todos los ejes como la misma recta. WQuestions rompe ambas reglas: admite ejes vacíos y ejes de naturalezas distintas.

LAS TABLAS RELACIONALES

Una tabla por tipo de entidad, columnas fijas, claves foráneas que se tienden a mano entre tablas. El cruce entre dominios es un proyecto de integración. Aquí, en cambio, todo vive en un mismo espacio y el cruce es un corte más, no un *join* que alguien tuvo que prever.

LOS CUBOS OLAP

Un cubo por área (ventas, recursos humanos), con dimensiones cerradas en su mundo. WQuestions es *un solo cubo maestro* donde la «industria» es apenas un filtro más dentro del eje K. Mil cubos pequeños se vuelven un único universo integrado.

El contraste con los *embeddings* de la inteligencia artificial merece párrafo aparte, porque ahí la diferencia es de otra índole. Cuando un modelo «lee» una palabra, la convierte en un vector denso de cientos de números para medir su cercanía a otras. Ese espacio es enormemente útil (captura parecidos sutiles) pero es **opaco**: ningún humano lee esos millones de cifras para saber por qué el modelo unió dos ideas. El espacio de WQuestions es lo contrario: **transparente y auditable**, cada eje con una etiqueta legible. La IA usa los *embeddings* para pensar rápido en privado; usa WQuestions para guardar el resultado de un modo que una persona pueda inspeccionar y defender. No compiten: se complementan.

PRECEDENTE · LOS ESPACIOS CONCEPTUALES DE GÄRDENFORS (13)

El filósofo y científico cognitivo Peter Gärdenfors propuso, en *Conceptual Spaces* (2000), que la mente representa los conceptos no como listas de símbolos ni como pesos de una red, sino como regiones de un **espacio geométrico de dimensiones de calidad**: el color, por ejemplo, vive en un espacio de tono, saturación y brillo, y «rojo» es una región convexa dentro de él. Esa intuición (que el conocimiento tiene forma geométrica y que las categorías son zonas, no etiquetas) es el antecedente más directo de este capítulo.

La diferencia es de propósito, y las dos visiones se dan la mano. Gärdenfors modela cómo el cerebro *aprende y representa* un concepto general; WQuestions modela cómo archivar los *hechos concretos* que usan esos conceptos. Sus dimensiones de calidad son continuas y perceptuales; nuestros ejes son tipados y heterogéneos. Pero ambas comparten la apuesta de fondo: que pensar en términos de espacio, y no de tabla, es la forma correcta de organizar lo que sabemos.

Tres cosas que este espacio no es

Para que la metáfora geométrica no prometa de más, conviene cerrar el contorno con tres negaciones deliberadas. La geometría es real, pero no hace todo lo que una geometría podría sugerir.

NO MIDE DISTANCIAS DE AFINIDAD

Que dos hechos compartan fecha no los vuelve «parecidos». Este espacio cruza conexiones exactas; no calcula cuán semejante es un paciente a otro. Para eso están, precisamente, los *embeddings*.

NO EXIGE UNA BASE DE DATOS RARA

El espacio de seis ejes es una forma de pensar y de organizar el código. A la hora de persistir, sirve PostgreSQL, MongoDB o un grafo como Neo4j. La geometría es independiente del motor.

NO ES UNA PRISIÓN DE DATOS

Modelar Recursos Humanos hoy no te prohíbe modelar Contabilidad mañana. Lo que modelas aterriza en el espacio; lo que no modelas, sencillamente no se ve. El sistema es modular y abierto.

Un espacio hecho para agentes

Cerremos uniendo la geometría con quien va a recorrerla. Cuando un agente autónomo se conecta a los datos de una organización, lo que hace, en términos del espacio, es **navegar esta geometría como un dron**. Dar de alta un cliente es dibujar un punto nuevo. Investigar un fraude es recortar una región y leer lo que hay dentro. Responder una pregunta es imponer una restricción y devolver el corte. Las mismas tres operaciones de antes, ahora ejecutadas por una máquina.

Y aquí está el cierre del argumento: este mapa de coordenadas (quién, qué, dónde, cuándo, cuánto, cuál) es **el mismo armazón cognitivo que el modelo de lenguaje ya usaba** para entender el mundo descrito en palabras. Al presentarle los datos estructurados de esta forma, el agente no aterriza en un ecosistema extraño que debe aprender desde cero: se encuentra con un universo digital que obedece a las mismas leyes que organizan su propio entrenamiento. Esa compatibilidad entre la forma de la pregunta y la forma del dato es lo que vuelve rentable todo el esfuerzo de modelar.

Pero la vida real reserva un nudo. En el eje O habitan unos individuos especiales: las **situaciones reificadas** (un partido entero, una sesión completa, un proceso de varios actos) donde miles de hechos menores convergen porque hablan de lo mismo. Son los puntos que más roles abren y, por tanto, los de mayor dimensión real. Cómo el modelo desata esos nudos sin que la estructura se desborde (cómo distingue una situación de su contexto, y dónde entra la agencia) es justo el asunto del próximo capítulo.

09

Situaciones, contextos y agencia

El mundo no nos llega como una lista de hechos sueltos, sino como escenas: un partido, una sesión, una escena de cine. Aprenderás cuándo una acción merece volverse un sustantivo, quién puede protagonizarla y cómo el tiempo deja de borrar la historia.

Considera dos frases que parecen decir lo mismo y no lo dicen. La primera: «alguien corre». La segunda: «el correr de alguien, ayer, en el parque». La diferencia es minúscula en el idioma y enorme en la arquitectura. En la primera, *correr* es un verbo fugaz: pasa y se esfuma, no deja a qué agarrarse. En la segunda, *el correr* se ha vuelto un sustantivo (una **cosa**) y, de golpe, admite un dueño, un lugar, un ayer; y mañana admitirá un más todavía: que duró cuarenta minutos, que fue su récord, que lo interrumpió la lluvia.

Toda lengua humana sabe hacer ese truco. *Vender* se vuelve *la venta*; *operar*, *la operación*; *decidir*, *la decisión*. Al convertir un verbo en sustantivo no añadimos un adorno literario: le otorgamos **identidad**. Lo que era un acto pasajero pasa a ser una entidad de la que podemos hablar, que podemos contar, fechar, cancelar y corregir. Este capítulo trata de cuándo conviene hacer ese truco dentro del modelo, y de las tres decisiones de diseño que lo gobiernan.

> **ETIMOLOGÍA**
> A este truco (tomar un evento fugaz y volverlo un objeto de primera clase) el modelo lo llama **reificación**, del latín *res*, «cosa»: literalmente, «hacer cosa» un evento. La entidad que nace de ahí es una **situación reificada**.

El nombre técnico de ese truco es **reificación**, y la entidad que produce es una **situación reificada**. Estas situaciones viven siempre en el eje O qué. Son los nudos de la red: los conectores maestros donde decenas de hechos atómicos convergen porque todos hablan de lo mismo. Sin ellos, el modelo solo sabría registrar oraciones simples. Con ellos, puede guardar cualquier escena humana, por intrincada que sea, sin abandonar jamás la forma de la tripleta que fijamos en el capítulo 7.

No todo merece ser un sustantivo

La tentación, una vez que descubres la reificación, es aplicarla a todo. ¿Si un cliente es alto, debo crear «la situación de su altura»? La respuesta es un no rotundo. Reificar cuesta: cada situación es un objeto nuevo en la base de datos, con su identificador único, que el sistema tendrá que buscar y mantener. Reificar por costumbre es inflar la base con abstracción vacía y volverla lenta.

La estatura de una persona se queda perfectamente como un hecho atómico plano, `(cliente_1042, estatura_cm, 178)`: no tiene partes, no participa nadie más, nadie volverá a hablar de ese número, no caduca. Reificarla sería como guardar un termómetro dentro de una caja fuerte. La pregunta correcta, entonces, no es «¿puedo reificar esto?», sino «¿hay alguna razón de peso para hacerlo?». El modelo reconoce exactamente cuatro razones, y basta con que se cumpla una.

1 TIENE ATRIBUTOS PROPIOS

El hecho carga detalles que no caben en una sola frase: un lugar, un instrumento, un modo, un actor secundario. Un gol profesional tiene minuto, pierna y un pase previo; esos datos no tienen de dónde colgar si el gol no es una entidad.

2 PARTICIPAN MÁS DE DOS ACTORES

La tripleta solo conecta dos cosas a la vez. Una venta reúne vendedor, comprador, producto, precio y fecha. Reificarla es el modo estándar de amarrar cinco participantes sin romper la regla de las tres columnas.

3 SERÁ REFERENCIADO LUEGO

Si sabes que llegarán novedades sobre el hecho («el partido fue suspendido», «la sesión se reanudó»), ese hecho debe existir antes como objeto independiente para poder recibirlas.

4 SU VALOR CAMBIARÁ Y HAY HISTORIA

El precio de una suscripción de hoy cambiará el año que viene. Si reificamos con fecha de inicio y de fin, conservamos el historial intacto en vez de pisarlo. Es la regla de vigencia que verás más abajo.

Si un dato no cumple ninguna de las cuatro, déjalo como tripleta plana y el sistema seguirá rápido. Esta frontera es lo bastante importante como para escribirla en piedra: es la cuarta decisión de diseño del modelo.

D4 CUÁNDO REIFICAR UNA SITUACIÓN

Un evento o relación solo se transforma en una «situación reificada» (guardada en el eje O) si cumple al menos uno de cuatro requisitos: (1) tiene atributos propios como lugar o modo, (2) participan más de dos actores, (3) será referenciado por otros eventos en el futuro, o (4) su valor cambiará y necesitamos conservar un registro histórico. Si no cumple ninguna, usamos una tripleta simple.

Léela como una compuerta de eficiencia, no como un permiso. Reificar es poderoso justamente porque es selectivo: el modelo reserva sus nudos para los eventos que de verdad articulan información, y deja el resto como átomos baratos. La economía es deliberada.

Autopsia de una situación

¿Cómo se ve uno de estos nudos por dentro? Abramos uno. Tomemos un evento cotidiano pero con suficiente sustancia: la venta número 1, en la que el vendedor 17 le vendió una camiseta al cliente 1042 a las 16:32 de una tarde de mayo. Cumple D4 de sobra (reúne varios participantes, carga media docena de datos propios y volverá a aparecer cuando el cliente reclame o devuelva). Lo primero es «hacer cosa» el evento, creándolo en el eje O:

TRIPLETAS

```
(venta_001) ∈ O          // el nudo: a partir de aquí, todo cuelga de él
```

Ese identificador, `venta_001`, se vuelve el punto articulador. Todos los detalles de la operación se enganchan a él con hechos atómicos de tres partes, cada uno aportando una coordenada de un eje distinto:

TRIPLETAS

```
(venta_001, instancia_de,    venta)                    ∈ M(O, K)
(venta_001, agente,          vendedor_17)              ∈ M(O, Q)
(venta_001, cliente,         cliente_1042)             ∈ M(O, Q)
(venta_001, objeto,          camiseta_88)              ∈ M(O, O)
(venta_001, momento,         2026-05-14T16:32:00-05:00) ∈ M(O, T)
(venta_001, lugar_de,        tienda_centro)            ∈ M(O, L)
```

```
(venta_001, monto,           49.90)                  ∈ M(O, N)
(venta_001, impuesto_pct,    18)                     ∈ M(O, N)
(venta_001, estatus_factual, real)                   ∈ M(O, K)
```

Nueve hechos atómicos dibujan la venta completa: qué clase de operación es, quién la hizo, a quién, qué objeto, cuándo, dónde, por cuánto y en qué estado. Y aquí llega la recompensa de haberla reificado. Como `venta_001` ahora tiene identidad propia, otros eventos posteriores pueden apuntar directamente a ella:

TRIPLETAS

```
(valoracion_001, sobre_situacion, venta_001)   // el cliente la puntúa
(pago_tarjeta_9, parte_de,        venta_001)   // cómo se pagó
(devolucion_088, rectifica,       venta_001)   // "talla equivocada", se cambió
```

La valoración, el pago y la devolución son, cada una, **nuevas situaciones reificadas** que se conectan al nudo original. La red crece de forma orgánica, sin cambiar jamás su formato de tres columnas. Esto es lo que la siguiente figura hace visible: un evento en el centro y sus participantes irradiando hacia los ejes, cada cable con su color.

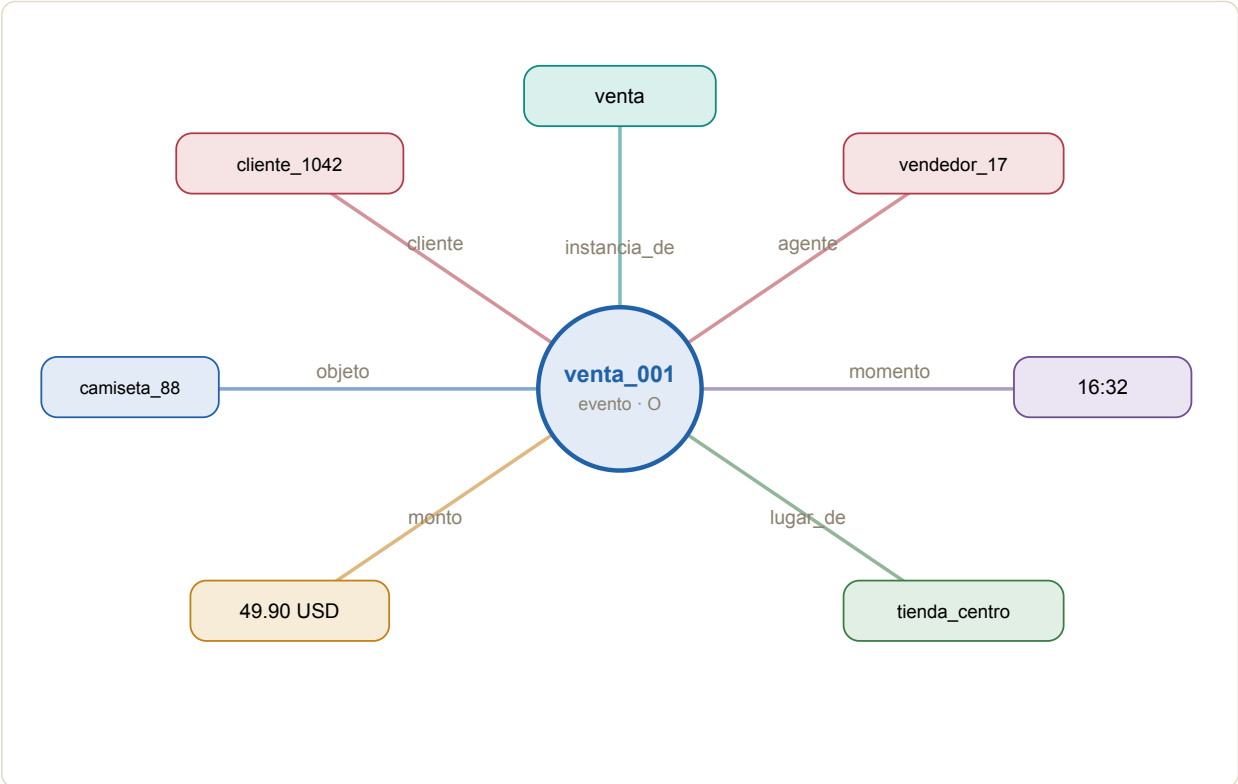


Figura 9.1. La venta del vendedor 17, reificada como el individuo `venta_001` en el eje O, con sus participantes colgados por roles canónicos. Cada cable lleva el color del eje al que apunta: el vendedor y el cliente en rojo (Q), la clase en verde azulado (K), el objeto vendido en azul (O), el momento en violeta (T), el lugar en verde (L), el monto en ámbar (N). La escena entera cabe sin abandonar la tripleta.

Quién puede ser agente

Detengámonos en uno de esos cables, porque oficializa algo que los ejemplos ya venían gritando. En la venta, el rol `agente` lo ocupó una persona de carne y hueso. Pero abre el abanico de dominios y el protagonista de una acción puede ser otra cosa: el algoritmo que asigna un conductor, el servidor bancario que cobra el mantenimiento, el modelo de lenguaje que redacta un borrador, el sensor que dispara una alarma. El modelo **no exige que el agente sea humano**.

Mira una sesión de un asistente de inteligencia artificial. El protagonista de la situación (quien ejecuta la respuesta, invoca las herramientas y consume los tokens) no es la usuaria que escribe, sino el modelo mismo:

TRIPLETAS

```
(sesion_ia_5521, instancia_de,    sesion_modelo_lenguaje)  ∈ M(O, K)
(sesion_ia_5521, agente,          modelo_lumen_2026)       ∈ M(O, Q)   ← un software es el agente
(sesion_ia_5521, usuaria,         paredes)                 ∈ M(O, Q)
(sesion_ia_5521, herramienta,     consulta_grafo)          ∈ M(O, K)
(sesion_ia_5521, tokens_salida,   920)                     ∈ M(O, N)
(sesion_ia_5521, latencia_ms,     2100)                    ∈ M(O, N)
```

El mismo molde sirve para un sensor. «El termopar registró 480 °C» pone a un dispositivo físico en el rol de agente de la situación *medir*, sin que ningún humano haya tocado nada. Esta universalidad del rol es la que nos deja auditar plantas industriales, plataformas de IA y sistemas automáticos con exactamente el mismo lenguaje que usamos para registrar ventas entre personas. Lo formalizamos así:

D5 AGENCIA CONTEXTUAL

El rol de `agente` puede ser ocupado por humanos, corporaciones, algoritmos de software o sensores físicos. Todo depende del verbo de la acción. Hay verbos que exigen un agente (como «vender») y otros que no necesitan a nadie («ocurrir», «llover»).

El rol, no la voluntad

Conviene fijar el límite, porque la pregunta es legítima: si un horno inteligente o un algoritmo de recomendación pueden «vestirse» de agente, ¿cuándo deja un objeto de ser una *herramienta* y pasa a ser un *actor*? La tentación es buscar la frontera en la **voluntad** o la **conciencia**, y por ahí no hay salida limpia: ni un algoritmo ni una empresa «quieren» nada en sentido fuerte.

El modelo esquiva la trampa filosófica. El criterio no es la voluntad: es el **rol que la entidad ocupa en una situación concreta**. Una misma cosa puede ser las dos cosas en momentos distintos. La cámara con la que se filma una película es un **instrumento** (un objeto en O) de la situación *filmar*; el agente es la directora. Pero «el dron sobrevoló el set y capturó el plano aéreo» pone al dron en el rol de **agente** de la situación *capturar*, donde no hay otro candidato a protagonista. El mismo aparato es herramienta o actor según qué verbo lo invoque y qué rol llene, no según una propiedad que lleve puesta.

La regla operativa, entonces, es simple: una entidad entra a Q como agente cuando **ocupa el rol `agente` de un verbo que lo exige** (vender, asignar, redactar, capturar); se queda como instrumento en O cuando es solo el medio a través del cual otro actúa. La agencia la determina la **gramática de la situación**, no la metafísica del objeto. Por eso el modelo puede auditar a un algoritmo que niega un crédito con el mismo rigor con que audita a un gerente que lo niega, sin tener que resolver primero si el algoritmo «tiene voluntad».

> *La agencia no es una propiedad que un objeto lleva puesta; es un papel que el verbo reparte. El robot que sutura es agente; el bisturí, jamás.*

LA REGLA DEL ROL

PRECEDENTE

Esta idea (que el sentido vive en la *situación*, no en la palabra aislada) es el corazón de la *semántica de situaciones* de Barwise y Perry[11] (1983). Y la noción de que un evento es un individuo del que se predican roles viene de Davidson[12] (1967): la reificación del modelo es, en el fondo, una forma operativa de su argumento eventual.

DOS RAÍCES FORMALES

Davidson propuso en 1967 tratar cada evento como un individuo lógico: «alguien corrió» se vuelve «existe un evento *e* tal que *e* es un correr, su agente es alguien, su lugar es el parque…». Reificar en el eje O es exactamente eso, llevado a una base de datos. Barwise y Perry, en su semántica de situaciones, añaden la pieza que cierra el capítulo: los participantes ocupan *roles* dentro de una escena, y es la escena —no la palabra— la que fija el significado.

Eventos que todavía no han pasado

Una situación no tiene por qué ser un hecho consumado. Los negocios necesitan guardar reuniones planeadas, rodajes que se cancelaron, diagnósticos que son hipótesis, compras que se devolvieron. Antes que inventar etiquetas duplicadas para cada caso (un cable `agente_real` y otro `agente_planeado`), la solución más limpia es obligar a toda situación a declarar abiertamente su **estatus factual**:

SIGNATURA

```
estatus_factual : de O hacia K
  Opciones permitidas: real | planeado | confirmado | hipotético | cancelado | rectificado
```

Una escena de cine programada para rodarse la semana próxima se guarda igual que una ya filmada, solo que con `estatus_factual: planeado`. Cuando se rueda, el sistema no borra el plan: crea una situación *nueva* con estatus `real` y la conecta a la antigua diciendo que la `cumple`. Si se cancela, se crea una situación de cancelación. El plan original nunca se reescribe.

TRIPLETAS

```
(escena_42) ∈ O
  instancia_de    : escena
  parte_de        : pelicula_marea
  directora       : serra
  momento         : 2026-06-04T07:00:00-05:00
  estatus_factual : planeado                    ← aún no se ha rodado
```

Este pequeño recurso asegura la **inmutabilidad** de la base: el pasado jamás se sobreescribe. Y a la vez deja mapear toda la red de expectativas, promesas y fracasos que compone el día a día de cualquier operación real. Aquí conviene fijar una convención que atraviesa todo el modelo.

CONVENCIÓN DE HECHOS INMUTABLES

Un hecho nunca se borra ni se edita: se *supera*. Para corregir el pasado no se modifica la situación vieja, sino que se crea una nueva y se la enlaza a la anterior con un cable de reacción (`cumple` , `cancela` , `rectifica`). La verdad del sistema no es un estado que se pisa, sino una sucesión de hechos que se acumula. Auditar deja de ser un esfuerzo: es leer la pila.

Datos que caducan: la vigencia

Llegamos a uno de los recursos de diseño más finos del modelo. Hay propiedades que sencillamente no duran para siempre. El precio de una camiseta cambia con cada temporada y cada rebaja. La dirección de un cliente cambia cuando se muda. El modelo de IA que corre en producción se reemplaza por una versión nueva cada pocas semanas. Una base de datos ingenua, al recibir un valor nuevo, pisa el viejo y destruye el historial para siempre.

El modelo evita ese desastre usando la reificación para domesticar el tiempo. En lugar de la barbaridad de reescribir un dato vivo (

TRIPLETAS

```
(sesion_ia, modelo_en_uso, modelo_lumen_2025)   ✗  // ¿y si se actualiza? ¿borrar y pisar?
```

) convertimos la propiedad cambiante en una sucesión de situaciones, cada una con su rango de vigencia. Esta es la sexta decisión de diseño.

D6 VIGENCIA TEMPORAL

Las propiedades que cambian con el paso del tiempo no se guardan directamente. Se reifican convirtiéndolas en situaciones, y se les añade una fecha de `inicio` y una de `fin` (su rango de vigencia).

Así, la pregunta «¿qué versión del modelo estaba en producción?» deja de tener una sola respuesta y pasa a tener una respuesta *por fecha*:

TRIPLETAS

```
(despliegue_001) ∈ O
  sujeto : servicio_inferencia
  modelo : modelo_lumen_2025
  inicio : 2025-09-01
  fin    : 2026-03-14

(despliegue_002) ∈ O
  sujeto : servicio_inferencia
  modelo : modelo_lumen_2026
  inicio : 2026-03-15
  fin    : null          ← vacío: el sistema sabe que esta es la versión actual
```

Cuando le preguntas al sistema «¿qué modelo corría en producción?», la máquina te pide una fecha de referencia. Si preguntas por el 20 de febrero de 2026, recorre los rangos y devuelve `modelo_lumen_2025`; si preguntas por hoy, devuelve `modelo_lumen_2026`. La siguiente figura muestra esa sucesión como una banda temporal: cada vigencia es un tramo, y una consulta fechada es una línea vertical que cae sobre el tramo correcto.

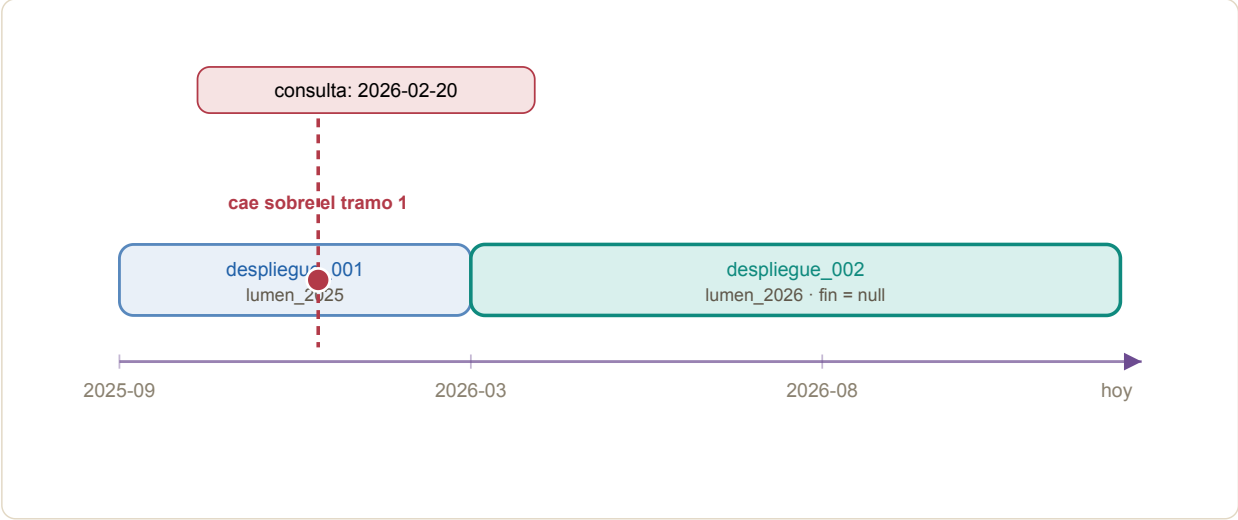


Figura 9.2. La propiedad «modelo en producción» modelada según D6 como una sucesión de situaciones, cada una con su rango de vigencia (azul, ya cerrado; verde azulado, abierto con `fin = null`). Una consulta fechada (línea roja, eje Q de quien pregunta) cae sobre el tramo válido en ese instante y devuelve la versión correcta. Recuperar el pasado deja de ser arqueología.

Esta técnica (la **bitemporalidad**) vuelve triviales consultas que antes enloquecían a los analistas: «¿qué versión exacta del modelo corría el día que se reportó el fallo el año pasado?». Sin D6, responder eso es casi imposible; con D6, es leer un rango.

Cómo se conectan las situaciones entre sí

Para que la base no degenere en una selva de eventos sueltos, el modelo define un puñado de cables oficiales (relaciones canónicas) que organizan cómo se relacionan las situaciones entre ellas. Son pocos, pero cubren todos los escenarios. Conviene verlos en dos columnas, porque caen en familias claras.

1 · Jerarquía (`parte_de` y su inverso `contiene`). Mete eventos pequeños dentro de eventos grandes. Un gol es `parte_de` un partido; una escena es `parte_de` una película. El todo y la parte coexisten como entidades de pleno derecho, sin que el todo aplaste a sus partes.

2 · Secuencia lógica (`precede` y su inverso `sigue_a`). Ordena el flujo de un proceso. No importa tanto la hora del reloj como qué paso va obligatoriamente antes que otro: la grabación del sonido `precede` a la mezcla, sin importar cuántos días medien.

3 · Acción y reacción (`cumple` , `cancela` , `modifica` , `rectifica`). Define cómo un evento nuevo impacta o anula a uno del pasado. El rodaje real `cumple` la escena planeada; una toma corregida `rectifica` la anterior. Aquí vive la inmutabilidad: se supera, no se borra.

4 · El grupo del «por qué» (`causado_por` , `motivado_por` , `justificado_por`). Los cables que explican los motivos (humanos o mecánicos) detrás de las situaciones. Son tan particulares que merecen su propio capítulo: el siguiente.

Mira cómo se aplican a un partido de fútbol, el ejemplo arquetípico de una situación que *contiene* otras. El partido es el nudo grande; cada jugada es un nudo pequeño que cuelga de él por `parte_de` , y entre jugadas hay secuencia y reacción:

TRIPLETAS

```
(partido_arg_per_2026) ∈ O
  instancia_de : partido
  estatus_factual : real

(gol_001, parte_de,     partido_arg_per_2026)   // la jugada cuelga del partido
(gol_001, instancia_de, gol_jugada_abierta)  ∈ M(O, K)
(gol_001, agente,       messi)               ∈ M(O, Q)
(gol_001, asistente,    di_maria)            ∈ M(O, Q)
(gol_001, minuto,       87)                  ∈ M(O, N)

(pase_001, precede, gol_001)    // ocurrió justo antes
(pase_001, agente,      messi)       ∈ M(O, Q)
(pase_001, beneficiario, di_maria)   ∈ M(O, Q)
```

El partido, el pase y el gol son situaciones distintas, todas en O, todas tejidas con cables canónicos. Borra cualquiera y las demás siguen teniendo sentido; añade una jugada nueva y nada se rompe. La estructura n-aria (agente, asistente, minuto y pierna en un solo gol) se conserva entera porque cada situación absorbe la complejidad sin deformar la tripleta.

El zoom de la reificación

Cierro con una aclaración vital para quien implemente. Reificar no es un interruptor de encendido y apagado: es un **zoom**. Un mismo evento admite varios niveles de detalle, y los tres son perfectamente legales. Tómalo con la venta de antes, pero ahora pensando en las ventas de la tienda a lo largo de un día.

1 BROCHAZO GRUESO

Una tripleta plana. Suficiente si solo quieres el total del día.

```
(tienda_centro,
ventas_dia_usd, 3420)
```

2 REIFICACIÓN NORMAL

«Hicimos cosa» el cierre. Necesario para auditar caja y turno.

```
(cierre_dia_14) ∈ O
  lugar  : tienda_centro
  total  : 3420 USD
  turno  : tarde_t2
```

3 REIFICACIÓN EXTREMA

Cada venta, una situación propia, ligada al día por `parte_de`.

```
(venta_001) ∈ O
  parte_de : cierre_dia_14
  monto    : 49.90
```

¿Cuál es el correcto? **Los tres.** Depende del presupuesto y del objetivo. Y lo más hermoso de esta arquitectura es que **saltar del nivel 1 al nivel 3 no rompe la base**: si mañana necesitas más detalle, inyectas situaciones nuevas y las cuelgas de las viejas con `parte_de`. El sistema nunca se detiene, porque nunca cambia su forma. El nivel de zoom es una decisión de negocio, no una atadura del modelo.

EN LA PRÁCTICA

Empieza siempre por el nivel más grueso que responda tus preguntas de hoy. Reificar de más es deuda técnica disfrazada de prolijidad; reificar de menos se corrige sin dolor más adelante, gracias a `parte_de`. La regla D4 es tu checklist: si un dato no cumple ninguno de sus cuatro requisitos, déjalo plano y duerme tranquilo.

Darle identidad a una es nombrar a todas

Reificar tiene una recompensa que se cobra más tarde. Al «hacer cosa» a `venta_001` parece que solo le damos identidad a un evento suelto para colgarle atributos: su monto, su cliente, su momento. Pero al darle un tipo —al decir que es una instancia de `venta`— la inscribimos, sin proponérnoslo, en un conjunto: el de todas las ventas. Esa es la otra cara del nudo. Un evento con identidad propia no solo recibe detalles; queda contado junto a sus pares, y «¿cuántas ventas hubo?» pasa a tener respuesta sin armar nada nuevo.

PYTHON

```python
# Cuántas ventas se reificaron — un tipo de situación, contado entero
count(u, Pattern(type_constraint=u.ind("venta")))
```

Las situaciones reificadas son las rotondas por donde circula el tráfico de la base: los puntos donde convergen los hechos y desde donde se ramifican los procesos. Pero para que el mapa esté completo falta pavimentar un tipo de conexión muy especial entre esas rotondas. Saber que ocurrió una venta o un fallo es útil; saber *qué lo causó*, *qué finalidad perseguía* o *qué norma lo justificó* es lo que eleva una base de datos a inteligencia de negocio. Esos cables del «por qué» son tan singulares que el próximo capítulo defiende una tesis incómoda: que **el porqué no es un eje**.

10

El "por qué" no es un eje

Cuatro veces preguntamos lo mismo y cuatro veces respondemos cosas que no se parecen en nada. La palabra es una sola; debajo hay cuatro mundos. Modelarlos como uno solo es el error que este capítulo desarma.

Imagina la sala de control de un estadio de fútbol, una noche cualquiera, con cuatro personas haciéndose preguntas al mismo tiempo. El técnico de mantenimiento mira un tablero y pregunta *¿por qué se apagó la pantalla del marcador?*; le responden que un cable se recalentó. La entrenadora pregunta *¿por qué Messi remató de zurda desde el borde del área?*; le dicen que le llegó el pase de Di María y la portería quedó abierta. El árbitro pregunta *¿por qué se detiene el juego ahora?*; le responden que el reglamento obliga a pararlo cuando hay un jugador lesionado. Y un periodista, abajo en la grada, pregunta *¿por qué la afición se puso de pie?*; le contestan que la jugada los emocionó. Cuatro «por qué» idénticos en la boca. Cuatro respuestas que no comparten ni la sustancia ni la dirección en el tiempo.

> **ADELANTO**
> Con este capítulo el grafo gana lo que le faltaba: la capacidad de *explicar*. Aún quedan dos piezas de la Parte III (la identidad de las entidades entre sistemas y los puentes con otros paradigmas) antes de que la Parte IV aborde cómo el lenguaje humano entra al modelo, empezando por el verbo como signatura.

El cerebro de quien responde hace, sin notarlo, un trabajo prodigioso: detecta que cada pregunta exige un tipo de explicación distinto y cambia de registro al vuelo. Al cable recalentado le da una explicación **física**: una cosa empujó a la otra. Al remate de zurda le da una explicación de **intención**: Messi quiso, decidió. Al juego detenido le da una explicación **normativa**: una regla lo manda. Y a la afición de pie le da, otra vez, una explicación de estado mental: una emoción la movió. Nadie enseña esto en la escuela. Lo hacemos desde niños.

Esa misma confusión afortunada del lenguaje (una palabra para cuatro cosas) es la que ha hecho tropezar a generaciones de ingenieros de datos. La tentación es obvia: si tengo un eje para el *quién*, otro para el *cuándo* y otro para el *cuánto*, ¿por qué no abrir uno para el *por qué* y guardar ahí las explicaciones? El capítulo entero es la respuesta a esa pregunta. Y la respuesta es no.

Cuatro respuestas, cuatro sustancias

Llevemos las cuatro preguntas de la sala de control a un terreno más frío, el de un acta de incidencias, y miremos no las palabras sino *qué tipo de cosa* es cada respuesta:

LAS CUATRO PREGUNTAS Y LA NATURALEZA DE SU RESPUESTA

¿Por qué se apagó el marcador? — Porque un cable se recalentó. La respuesta es **un hecho anterior**, un evento del pasado que empujó a otro. Pura física.

¿Por qué Messi remató de zurda? — Porque quería marcar y ganar el partido. La respuesta es **un estado mental**: un deseo en la cabeza de alguien.

¿Por qué el cirujano abrió esa vía? — Para alcanzar el lóbulo afectado. La respuesta es **un resultado futuro**, algo que *todavía no ocurre* y que la acción persigue.

¿Por qué se detuvo el juego? — Porque el reglamento lo exige ante un jugador lesionado. La respuesta es **una norma**: un documento, una regla con autoridad.

Detente en la columna invisible de la derecha: **un evento pasado, un estado mental, un resultado futuro y una norma**. No hay forma de meter esas cuatro cosas en la misma caja sin mentir. Un evento pasado y un resultado futuro apuntan en direcciones opuestas del tiempo. Una emoción vive en una mente; una norma vive en un boletín oficial. Si las obligáramos a convivir en un único eje «por qué», estaríamos guardando en la misma gaveta un cortocircuito, un deseo, un plan y un párrafo legal.

Y aquí está el problema arquitectónico en una sola frase: si abriéramos un eje exclusivo para el porqué, **¿qué clase de cosa viviría dentro?** La pregunta no tiene respuesta, porque el eje no tendría sustancia propia.

Por qué un eje "por qué" no puede existir

Conviene recordar qué hace que un eje sea un eje en este modelo. La caja Q guarda agentes: personas, organizaciones, software. La caja T guarda instantes y rangos de tiempo. La caja K guarda clases atemporales. Cada eje es **homogéneo**: todo lo que vive dentro está hecho de la misma materia y se ordena con las mismas reglas. Dos fechas se pueden comparar; dos números se pueden sumar; dos clases se pueden subordinar una a otra. La homogeneidad no es un lujo: es lo que permite que el sistema razone sobre un eje sin caso por caso.

Apliquemos esa exigencia al supuesto eje «por qué». Sus habitantes serían las respuestas que acabamos de listar, y ninguna se parece a la otra:

HETEROGENEIDAD

```
¿por qué se apagó el marcador?   → un EVENTO PASADO   (vive en O, ya ocurrió)
¿por qué remató Messi?           → un ESTADO MENTAL   (vive en O, en una mente)
¿por qué abrió la vía?           → un EVENTO FUTURO    (vive en O, aún no ocurre)
¿por qué se detuvo el juego?     → una NORMA           (vive en O, un documento)
```

Cuatro sustancias distintas pidiendo el mismo cajón. Si forzáramos al sistema a tratarlas por igual, lo condenaríamos a confundirlas. Un gerente que preguntara *¿qué causó esta falla?* podría recibir como respuesta una cláusula contractual; un auditor que buscara *qué norma amparó esta decisión* podría toparse con un cortocircuito. El eje sería un cajón de sastre, y un cajón de sastre no es una coordenada: es justo lo que este libro intenta abolir.

LA TRAMPA DEL CAJÓN DE "OBSERVACIONES"

La versión perezosa de este error es vieja y conocida: un campo de texto libre llamado *Observaciones* o *Justificación* donde cada quien escribe lo que quiere. Para la máquina, ese campo es un cementerio: no se puede auditar, no se puede recorrer, no se puede consultar. El supuesto eje «por qué» sería ese mismo cementerio, solo que con la pretensión de parecer estructura.

La decisión D7: cuatro cables, no un eje

La salida no es buscar un sustantivo común a las cuatro respuestas —no lo hay— sino aceptar que el porqué del lenguaje natural es, en realidad, cuatro relaciones distintas que el idioma comprime en una sola palabra. En lugar de un eje nuevo, el modelo tiende cuatro cables, cada uno con su signatura y su disciplina. Esta es la séptima decisión de diseño del libro, y conviene dejarla escrita en piedra.

D7 **EL PORQUÉ SE DIVIDE EN CUATRO CABLES**

El modelo no tiene un eje "por qué". En su lugar, el sistema toma el "porque" del lenguaje natural y lo divide en cuatro cables distintos (relaciones canónicas) que conectan a las situaciones entre sí. Esos cables son: `causado_por` (para la física), `motivado_por` (para la intención humana), `con_finalidad` (para el propósito futuro) y `justificado_por` (para las reglas y leyes).

En la práctica, esto significa que **no nace ninguna caja nueva**. Los cuatro cables son habitantes del eje **M** (el *cómo*, los predicados que aprendimos a tender en el capítulo 5), y todos comparten la misma signatura: conectan una situación del eje O con otra situación del eje O. La explicación deja de ser un atributo y pasa a ser un enlace entre hechos.

SIGNATURAS

```
causado_por      ∈ M(O, O)    // la física ciega de causa y efecto
motivado_por     ∈ M(O, O)    // la intención en la mente de un agente
con_finalidad    ∈ M(O, O)    // un propósito proyectado hacia el futuro
justificado_por  ∈ M(O, O)    // la autoridad de una norma o regla
```

La maniobra clave es *cuándo* se desambigua. No guardamos un porqué genérico para interpretarlo después: elegimos el cable correcto **antes** de escribir el hecho. En el momento de registrar, alguien (una persona, o una IA leyendo una frase) decide si lo que tiene entre manos es una causa, un motivo, una finalidad o una justificación. Esa decisión, tomada una vez al entrar, ahorra mil ambigüedades al salir.

CONEXIÓN

Los cuatro cables son `M(O, O)` porque ambos extremos son situaciones del eje O. Que un evento pueda ser sujeto y objeto de un enlace es precisamente lo que la reificación del capítulo 9 hizo posible.

Las cuatro relaciones se dibujan mejor que se describen. La figura siguiente las pone una al lado de la otra como cuatro cables de distinto color saliendo de un mismo hecho, cada uno hacia una sustancia diferente.

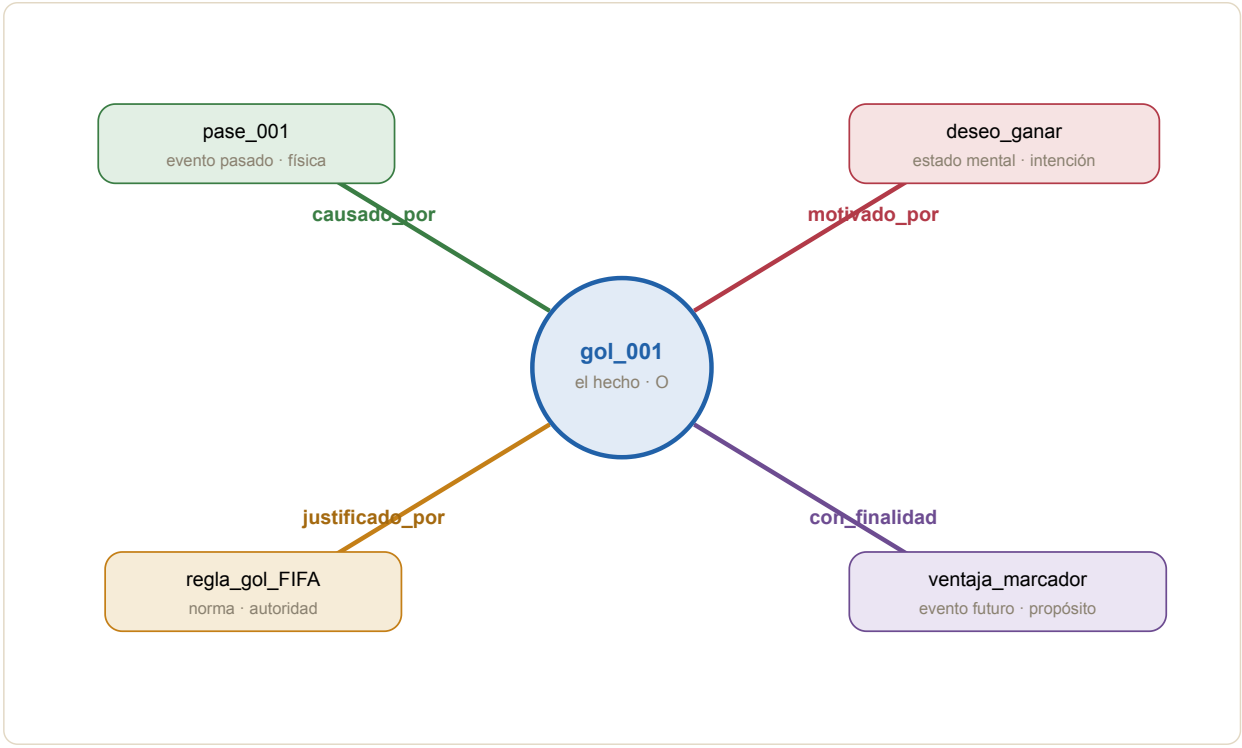


Figura 10.1. El mismo hecho explicado por cuatro cables que no se confunden: `causado_por` apunta a un evento pasado (la física), `motivado_por` a un estado mental (la intención), `con_finalidad` a un evento futuro (el propósito) y `justificado_por` a una norma (la autoridad). Cuatro relaciones precisas en lugar de un eje confuso.

Los cuatro cables, uno por uno

Cada cable tiene su carácter, su dominio favorito y una propiedad que lo distingue de sus hermanos. Vale la pena recorrerlos despacio, porque la diferencia entre ellos es justamente la riqueza que un eje único aplastaría.

1 · `causado_por` — la física, sin intención

Es el cable del dominó: un evento empuja a otro sin que medie deseo, plan ni permiso. Sirve para las conexiones mecánicas y ciegas, las que ocurrirían igual aunque no hubiera nadie mirando. Es el favorito de los diagnósticos y de las investigaciones de fallas.

TRIPLETAS

```
(gol_001,              causado_por, pase_001)                ∈ M(O, O)
(rotura_red_012,       causado_por, impacto_balon_993)       ∈ M(O, O)
(corte_imagen_transmi, causado_por, sobrecarga_generador_01) ∈ M(O, O)
```

Tiene una propiedad reveladora: **no necesita un agente humano**. Un balón golpea la red y la revienta sin que nadie lo desee; un generador se sobrecarga sin malicia. Por eso el modelo no exige que vincules una persona cuando usas este cable. La causa física es indiferente a las intenciones; ese es, precisamente, su signo distintivo.

2 · `motivado_por` — la chispa en una mente

Aquí cambia todo. Este cable explica una acción por el estado mental (deseo, miedo, creencia, ambición) del agente que la ejecutó. Responde no a *¿qué fuerza movió esto?* sino a *¿qué pasaba por la cabeza de quien lo hizo?*. Donde `causado_por` ignora a las personas, `motivado_por` las pone en el centro.

TRIPLETAS

```
(gol_001,          motivado_por, deseo_ganar_messi)          ∈ M(0, 0)
(cambio_tactico_04, motivado_por, temor_remontada_entrenador) ∈ M(0, 0)
(falta_tactica_06,  motivado_por, plan_frenar_contraataque)   ∈ M(0, 0)
```

La distinción con la causa física parece sutil, pero es vital. *El impacto reventó la red* y *la jugada motivó el cambio táctico* no son la misma clase de frase. La rotura no es algo que la red *quisiera*: simplemente ocurrió. El cambio táctico, en cambio, fue una decisión de alguien que leyó el partido. Las causas viven en el mundo físico; los motivos viven en mentes. Confundirlos es confundir un terremoto con una venganza.

> **LA LÍNEA QUE SEPARA LA CAUSA DEL MOTIVO**
>
> *El incendio causó el derrumbe* frente a *el incendio motivó la evacuación*. El derrumbe es física ciega: `causado_por`. La evacuación es una decisión humana ante el peligro: `motivado_por`. La misma chispa inicial, dos cables distintos, según haya o no una mente eligiendo en el medio.

3 · `con_finalidad` — el viaje hacia adelante

Es el cable más extraño de la familia, porque no conecta el hecho con algo del pasado sino con un **resultado que se busca en el futuro**. Es el cable del propósito, del objetivo, de la teleología. Responde a *¿para qué?*.

TRIPLETAS

```
(gol_001,            con_finalidad, ventaja_antes_del_final) ∈ M(0, 0)
(consulta_grafo_551, con_finalidad, respuesta_a_usuaria)     ∈ M(0, 0)
(rebaja_entradas_q3, con_finalidad, aumento_asistencia)      ∈ M(0, 0)
```

Surge de inmediato una objeción legítima: *¿cómo guardo en una base de datos un evento que todavía no ha sucedido?* La respuesta es elegante y reutiliza maquinaria que ya tenemos. El resultado futuro se guarda *hoy* como una situación más en el eje O, pero con una etiqueta de estado (la propiedad `estatus_factual` que vimos en el capítulo de situaciones) que dice `previsto`, `intencionado` o `hipotético`. El evento ya vive en el disco; lo único que apunta hacia adelante es su etiqueta de «aún no realizado».

TRIPLETAS

```
(ventaja_antes_del_final, instancia_de,    estado_marcador)  ∈ M(0, K)
(ventaja_antes_del_final, estatus_factual, intencionado)     ∈ M(0, K)
(gol_001,                 con_finalidad,   ventaja_antes_del_final) ∈ M(0, 0)
```

4 · `justificado_por` — el sello de la autoridad

Este cable aparece cada vez que una acción no se explica por un choque físico ni por un deseo, sino por una **regla, protocolo o ley que la autoriza**. Es indispensable para el derecho, los procedimientos regulados y las auditorías. Lo que está al otro extremo no es un evento del mundo natural: es una norma.

TRIPLETAS

```
(detencion_juego_088,    justificado_por, regla_lesion_FIFA)         ∈ M(0, 0)
(expulsion_jugador_03,   justificado_por, articulo_12_reglamento)    ∈ M(0, 0)
(anulacion_gol_120,      justificado_por, regla_fuera_de_juego)      ∈ M(0, 0)
```

Es el cable que vuelve auditable cualquier sistema regulado. Si un día alguien cuestiona por qué se anuló un gol, el grafo no responde con una opinión: señala el artículo exacto del reglamento que la respaldó, con su emisor y su fecha de vigencia. Volveremos sobre cómo se guarda una norma (porque una regla, en este modelo, también es un objeto) al final del capítulo.

Cuando la causa raíz es contar

Separar el porqué en cuatro cables rinde de verdad cuando dejas de mirar un hecho y miras el patrón. Si cada corte de transmisión cuelga de un `causado_por`, entonces «¿qué causa se repite más?» deja de ser una investigación y se vuelve un conteo: basta agrupar las situaciones por el otro extremo de ese cable. El análisis de causa raíz —que en otros sistemas es un proyecto con su analista y su hoja de cálculo— aquí es preguntar cuántos hechos comparten la misma causa, una causa por vez.

PYTHON

```
# Cortes atribuidos a una causa concreta — un corte; la causa raíz se compara entre causas
count(u, Pattern(fixed={"causado_por": u.ind("sobrecarga_generador_01")},
                 type_constraint=u.ind("corte_imagen")))
```

Cuatro cables en un solo hecho

Lo verdaderamente potente de tener cuatro cables bien separados es que un mismo evento puede llevarlos todos a la vez, y cada uno cuenta una parte distinta de la historia sin pisar a las demás. Volvamos a ese gol de Messi, marcado en el minuto 87, y preguntémosle las cuatro cosas:

TRIPLETAS

```
(gol_001, causado_por,     pase_001)               ← qué lo hizo posible: la asistencia de Di
María.
(gol_001, motivado_por,    deseo_ganar_messi)       ← qué pasaba por su mente: quería el triunfo.
(gol_001, con_finalidad,   ventaja_antes_del_final) ← qué resultado buscaba: ponerse por delante.
(gol_001, justificado_por, regla_fuera_de_juego)    ← qué lo hace válido: estaba habilitado.
```

Cuatro hechos atómicos, cuatro explicaciones de la misma jugada, ninguna intercambiable con otra. Si hubiéramos aplastado el porqué en un solo campo, esta riqueza se habría reducido a un murmullo: «se marcó porque sí». Cuatro cables la conservan entera. Y nótese que cada línea sigue siendo la tripleta de tres partes de siempre (la decisión D3 intacta); no hemos inventado ninguna estructura nueva.

Cadenas explicativas: el camino de migas de pan

En la vida real, los porqués rara vez vienen solos. Se enganchan unos con otros y forman una historia entera: cada respuesta abre la puerta a una nueva pregunta. A esa secuencia la llamamos una **cadena explicativa**, y es donde el modelo deja de archivar hechos y empieza a contar relatos navegables.

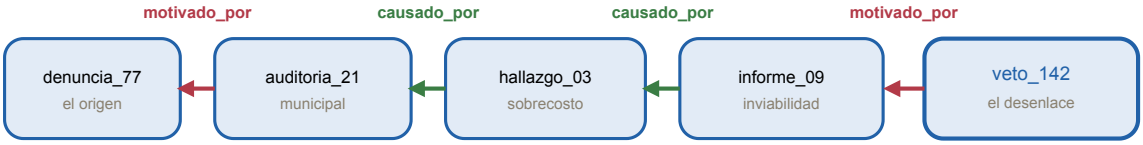
Tomemos un escándalo de gobierno municipal, de los que llenan portadas. Una alcaldesa veta una ordenanza. *¿Por qué?* Porque un informe técnico la declaró inviable. *¿Y por qué se emitió ese informe?* Porque una auditoría detectó un sobrecosto. *¿Y por qué se abrió la auditoría?* Porque un concejal presentó una denuncia. La historia, en el grafo, se ve como un camino de migas de pan perfectamente enlazado:

TRIPLETAS

```
(veto_ordenanza_142, motivado_por, informe_inviabilidad_09)   ∈ M(0, 0)
(informe_inviabilidad_09, causado_por, hallazgo_sobrecosto_03) ∈ M(0, 0)
```

```
(hallazgo_sobrecosto_03, causado_por, auditoria_municipal_21)  ∈ M(O, O)
(auditoria_municipal_21, motivado_por, denuncia_concejal_77)   ∈ M(O, O)
```

Esto no es prosa literaria: es un mapa que la computadora puede recorrer. Si le preguntas *«¿qué eventos contribuyeron al veto de la ordenanza?»*, el motor se aferra al primer hecho y empieza a trepar hacia atrás por los cables, hecho tras hecho, hasta agotar la cadena. Y como cada cable tiene tipo, puedes pedirle algo quirúrgico: *«recorre la historia hacia atrás, pero muéstrame solo las motivaciones humanas e ignora las causas mecánicas»*. La máquina filtra por `motivado_por` y descarta `causado_por` sin que tengas que escribir una línea de lógica nueva.



El motor trepa de derecha a izquierda; cada flecha es un cable tipado que puede filtrarse por separado.

Figura 10.2. Una cadena explicativa reconstruida hacia atrás, desde el desenlace (`veto_142`) hasta el origen lejano (`denuncia_77`). Los cables alternan entre `motivado_por` (intención, en rojo) y `causado_por` (física, en verde): por eso el grafo puede contar la historia completa o solo su hilo humano, según se le pida.

Reglas como objetos: el porqué normativo

El cable `justificado_por` nos dejó una deuda. Si su extremo apunta a una «regla», ¿cómo se guarda una regla en el grafo? La respuesta es coherente con todo lo demás del libro: **una regla es una situación reificada**, un objeto del eje O como cualquier otro, solo que de una clase especial. Toda norma bien formada tiene una condición, una consecuencia, una vigencia y una autoridad que la emitió.

TRIPLETAS

```
(regla_lesion_FIFA, instancia_de,   norma_reglamentaria)  ∈ M(O, K)
(regla_lesion_FIFA, emisor,         federacion_fifa)      ∈ M(O, Q)
(regla_lesion_FIFA, vigencia_desde, 2024-10-01)           ∈ M(O, T)
(regla_lesion_FIFA, condicion,      jugador_lesionado)    ∈ M(O, K)
(regla_lesion_FIFA, consecuente,    juego_detenido)       ∈ M(O, K)
```

Una vez que la norma vive como objeto, todo se vuelve posible: puedes citarla desde cien hechos distintos, modificarla mañana sin romper nada, compararla con otras normas o consultar desde cuándo rige. Cuando una jugada concreta se ampara en ella, el grafo simplemente tiende el cable:

`detencion_juego_088` o —justificado_por→ M(O→O) `regla_lesion_FIFA` o

Esto vuelve auditable a cualquier organización en el acto. Ante una controversia, el sistema no improvisa: indica exactamente qué norma autorizó la acción, quién la emitió y desde cuándo estaba vigente. Y si la federación inventa una regla nueva la próxima temporada, el grafo no se rompe: asimila un objeto más.

EL MODELO GUARDA; EL RAZONADOR DECIDE

Una aclaración: el grafo guarda la regla de forma impecable para que puedas *consultarla*, pero no es un autómata que tome decisiones por su cuenta. Para que el sistema lea la condición `jugador_lesionado` y aplique solo la consecuencia `juego_detenido`, hace falta conectarle por encima un evaluador o un modelo de IA. WQuestions provee la información perfecta; el razonamiento automático se construye sobre ella.

Cómo decidir qué cable usar

Cuando estés frente a un porqué y dudes de cuál de los cuatro cables tender, una sola pregunta suele bastar. La distinción no es académica: es la diferencia entre un grafo que explica y uno que confunde.

¿Habría ocurrido sin que nadie lo deseara? Si la respuesta es sí (un balón revienta la red, un cable se recalienta), es `causado_por`. La causa física no consulta a nadie.

¿La explicación está en la cabeza de alguien? Si lo que mueve la acción es un deseo, un miedo o una creencia de un agente, es `motivado_por`. El motivo necesita una mente.

¿Apunta a algo que todavía no ocurre? Si lo que explica la acción es el resultado que persigue, un estado futuro buscado, es `con_finalidad`. La finalidad mira hacia adelante.

¿La respuesta es una regla o una ley? Si lo que ampara la acción es una norma con emisor y vigencia, es `justificado_por`. La justificación se apoya en una autoridad, no en un hecho del mundo.

La alternativa perezosa sería un único cable genérico, un `por_que` para todo. Mejoraría algo frente al campo de texto libre, pero perderíamos la frontera entre la física, la mente, el futuro y la ley: un reporte mezclaría cortocircuitos con miedos y reglas como si fueran la misma sustancia. Los cuatro cables capturan justo el matiz que el cerebro humano usa para entender el mundo, y lo hacen sin abandonar la tripleta de tres partes ni la maquinaria de búsqueda que ya teníamos.

El mismo agente, humano o de silicio

Vale la pena un último giro, porque toca un dominio que llegó hace nada: los agentes de inteligencia artificial. Cuando el agente que ejecuta una acción no es una persona sino un modelo de lenguaje, los cuatro cables siguen funcionando sin un solo cambio (es la agencia contextual que ya vimos: el rol de agente lo puede ocupar un software). Mira la telemetría de una sesión:

TRIPLETAS

```
(consulta_grafo_551, agente,         modelo_lumen_2026)        ∈ M(O, Q)
(consulta_grafo_551, motivado_por,   peticion_usuaria_paredes) ∈ M(O, O)
(consulta_grafo_551, con_finalidad,  respuesta_a_usuaria)      ∈ M(O, O)
(consulta_grafo_551, justificado_por, politica_acceso_lectura) ∈ M(O, O)
```

El agente de IA actúa *motivado por* la petición de la usuaria Paredes, *con la finalidad* de devolverle una respuesta, y su acción está *justificada por* una política de acceso de lectura. Tres de los cuatro porqués, aplicados a un agente que no tiene cuerpo. Lo notable es que no inventamos vocabulario nuevo para la IA: el mismo modelo que explica la jugada de un delantero de fútbol explica la consulta de un modelo de lenguaje. Y ese hecho (que un agente pueda registrar *por qué* hizo lo que hizo, en un formato auditable) es una de las piezas que el capítulo 26 convierte en argumento.

Un precedente filosófico viejísimo

LAS CUATRO CAUSAS DE ARISTÓTELES (1)

Que el porqué se rompa en varias preguntas no es una ocurrencia moderna. Hace más de dos mil años, Aristóteles distinguió cuatro *causas* al explicar por qué existe algo: la material, la formal, la eficiente y la final. Nuestros cables no calcan esa lista (el modelo nace de la práctica, no de la metafísica), pero la coincidencia de fondo es elocuente: `causado_por` recoge su causa eficiente y `con_finalidad` su causa final. Veinticuatro siglos después, al exigirle precisión a una base de datos, volvemos a tropezar con la misma intuición: «por qué» nunca fue una sola pregunta.

> *El lenguaje comprime cuatro preguntas en una palabra; la ingeniería rigurosa vuelve a separarlas antes de guardarlas.*

LA TESIS DEL CAPÍTULO

El corazón arquitectónico, armado

Con esto cerramos el corazón arquitectónico del libro. En estos cuatro capítulos vimos cómo un hecho se reduce a una tripleta tipada, cómo millones de tripletas forman un espacio geométrico consultable, cómo las situaciones reificadas amarran eventos con muchos participantes, y —ahora— cómo cuatro cables le dan al grafo algo que ninguna tabla tuvo jamás: la capacidad de **explicarse a sí mismo**. La maquinaria conceptual está armada y engrasada.

Pero antes de salir de la Parte III quedan dos piezas que el modelo todavía necesita: cómo se reconoce que una misma entidad reaparece en sistemas distintos (la identidad) y cómo dialoga este modelo con los paradigmas que lo rodean (los puentes). Y más allá falta una pieza decisiva: nadie en una empresa va a sentarse a teclear tripletas a mano; la gente quiere hablar y escribir con naturalidad. La **Parte IV** resolverá ese obstáculo (cómo se construye el puente entre el lenguaje humano, ambiguo, vivo, desordenado, y la rigidez precisa de los hechos atómicos), empezando por la pieza que decide la forma de cada hecho: el verbo como signatura.

11

La identidad a través de los sistemas

Tres sistemas le dan tres nombres distintos. Pero es una sola persona. Antes de que el grafo pueda cruzar la tienda, la clínica y la municipalidad, tiene que resolver una pregunta que parece trivial y casi nunca lo es: ¿son el mismo?

Un martes de mayo, **Juan Vega** compra una camiseta en la tienda del centro. Para la caja registradora, Juan no es Juan: es `cliente_1042`, el comprador de la `camiseta_88`. Tres semanas después, ese mismo Juan llega a urgencias con una arritmia. Para el sistema de la clínica no existe ningún `cliente_1042`; existe `paciente_vega`, con su episodio y su historia. Y cuando, al mes, Juan paga sus arbitrios, la municipalidad lo conoce por un tercer nombre todavía: `contribuyente_77-3389`. Tres sistemas, tres identificadores, una sola persona de carne y hueso que ninguno de los tres sabe que comparte con los demás.

Esa es la grieta que este capítulo cierra. El libro prometió, desde la introducción, que las preguntas servirían de *vocabulario común* para que sistemas que hoy no se hablan pudieran responder juntos. Pero un vocabulario común no basta. Aunque la tienda y la clínica coincidieran en llamar `agente` al cable que apunta a una persona, seguirían sin saber que `cliente_1042` y `paciente_vega` son *la misma* persona. Responder a esa pregunta —¿quién es quién, de verdad, a través de los sistemas?— es lo que llamaremos **resolución de identidad**, y es el último eslabón que hace posible la promesa cross-dominio.

> **DÓNDE ENCAJA**
> Este capítulo cierra la Parte III. Da por sabidos el hecho atómico, la reificación y los cables del eje K. El siguiente conecta estas ideas con los objetos de la POO, los bits, los grafos y las cadenas de bloques.

Identidad no es lo mismo que identificador

Conviene separar dos palabras que el lenguaje cotidiano confunde y que la arquitectura obliga a distinguir. La **identidad** es la persona real: Juan Vega, un individuo único en el mundo, que existía antes de comprar nada y existirá después de que la camiseta se decolore. La identidad vive en el eje Q quién: es una entidad del modelo, no una cadena de texto. El **identificador**, en cambio, es el rótulo que un sistema concreto le pega encima para poder referirse a él: `cliente_1042` es el identificador que *la tienda* eligió, y solo tiene sentido dentro de la tienda.

La distinción importa porque los identificadores son *locales y arbitrarios*. Cada sistema acuña los suyos sin consultar a nadie: la tienda numera a sus clientes en orden de llegada, la clínica usa el apellido, la municipalidad concatena un código tributario. Ninguno es «más correcto» que otro; simplemente son tres nombres privados para una misma persona pública. El error clásico (y la causa de media torre de Babel) es confundir el rótulo con lo rotulado: tratar `cliente_1042` como si *fuera* la persona, en vez de como una flecha que apunta a ella.

IDENTIDAD VS. IDENTIFICADOR

Identidad: el individuo real, único en el mundo, una entidad del eje Q. Es lo rotulado.

Identificador (UID): el rótulo que *un* sistema le asigna para poder apuntarlo. Es local, arbitrario y prescindible. Es el rótulo. Una misma identidad puede llevar tantos identificadores como sistemas la conozcan.

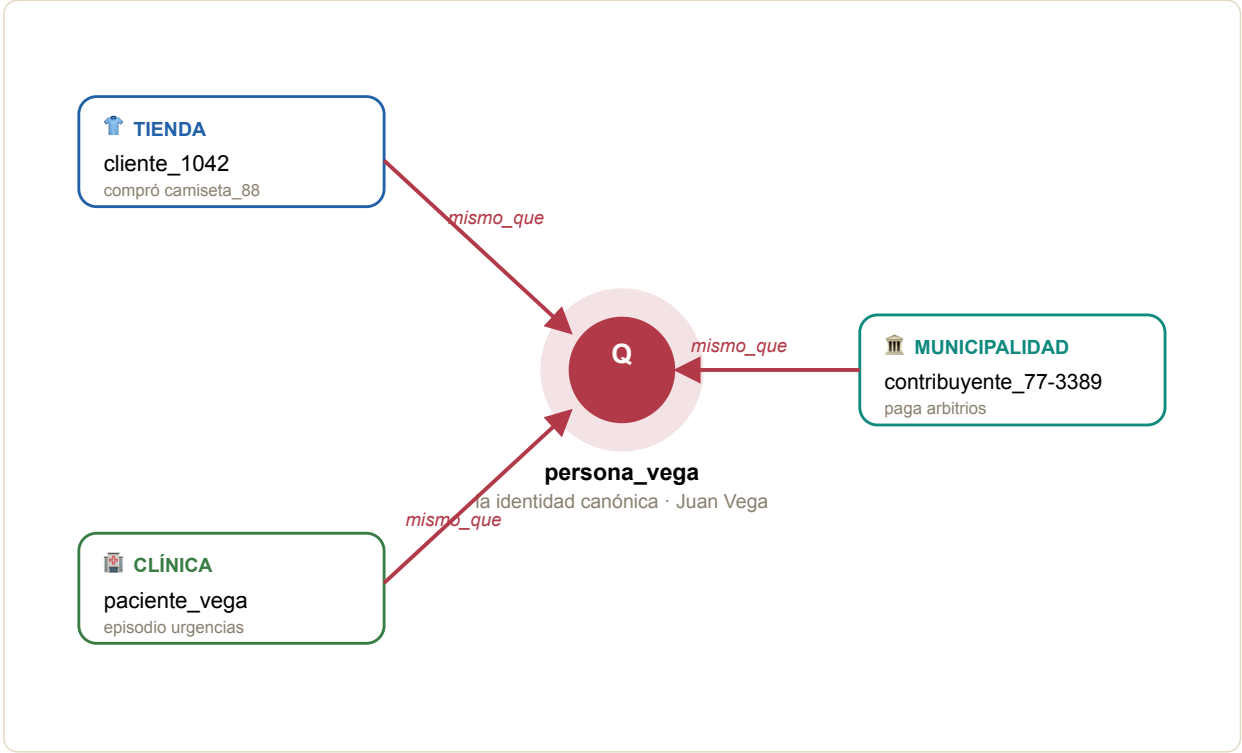


Figura 11.1. Tres sistemas conocen a Juan Vega con tres identificadores distintos. Cada rótulo local apunta, mediante el cable `mismo_que`, a una **identidad canónica** en el eje Q. Resolver la identidad es, literalmente, dibujar esas tres flechas.

La analogía que lo aclara todo: punteros

Quien haya programado en un lenguaje de bajo nivel reconocerá esta situación de inmediato, porque es exactamente la de los **punteros**. Un puntero no es un objeto: es una *dirección*, un papelito que dice «el objeto que buscas está allá». Varios papelitos pueden señalar el mismo rincón de la memoria. El objeto es uno; las direcciones que lo apuntan, muchas. Esa asimetría —un referente, varios referentes— es la columna vertebral de este capítulo.

Traduzcamos el modelo entero a ese vocabulario, porque la correspondencia es asombrosamente limpia:

Cada identificador es un puntero. `cliente_1042` no contiene a Juan: apunta a él. Es la dirección que usa la tienda para llegar a la persona.

Reificar es «tomar la dirección». Cuando en el capítulo 9 convertimos un evento fugaz (una venta, un gol) en un objeto con identidad propia, lo que hicimos fue darle una dirección estable para poder apuntarlo después. Reificar es el operador `&` del modelo: pasar de «algo ocurrió» a «este algo, al que ahora puedo señalar».

`mismo_que` **es *aliasing* de punteros.** Dos direcciones, un mismo objeto. Declarar `(cliente_1042, mismo_que, persona_vega)` es decirle al sistema: «estos dos papelitos llevan al mismo sitio; trátalos como sinónimos».

El grafo entero es una estructura de punteros. Cada tripleta enlaza nodos por referencia, no por copia. El conocimiento no es una pila de tablas planas: es un tejido de direcciones que se apuntan unas a otras, como un montón de objetos en un *heap*.

HACIA EL CAP. 12

Esta misma idea reaparece en la programación orientada a objetos: una variable no «contiene» un objeto, guarda una *referencia* a él, y dos variables pueden referirse al mismo. El capítulo 12 lleva la analogía hasta el fondo.

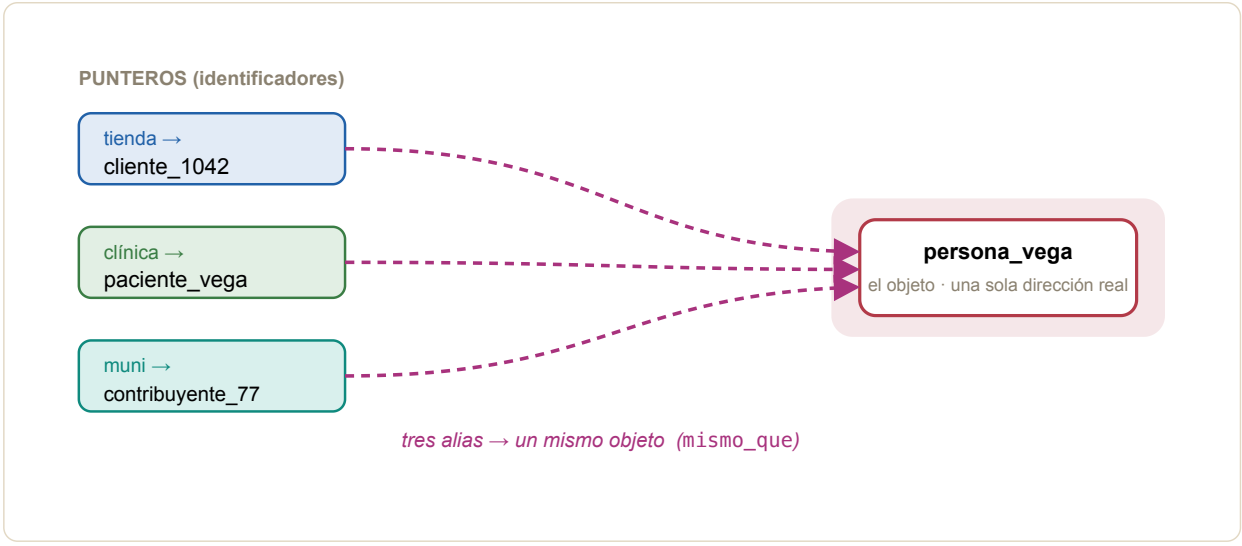


Figura 11.2. Los tres identificadores son punteros: tres direcciones distintas que, vía `mismo_que`, refieren al mismo objeto `persona_vega`. Esto es *aliasing*: nombres distintos para una sola cosa. El grafo no copia a Juan tres veces; lo apunta tres veces.

Cómo se resuelve la identidad, en la práctica

Saber que hace falta resolver la identidad es la mitad fácil. La otra mitad es *cómo* el sistema decide que dos punteros llevan al mismo sitio. Hay tres vías, ordenadas de la más fiable a la más resbaladiza.

Vía 1 · Claves naturales con autoridad externa

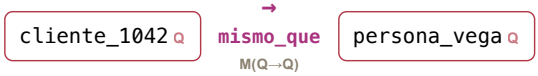
La situación ideal: existe un identificador que *no* es local, sino que lo emite una autoridad reconocida por todos. El DNI de una persona, el RUC de una empresa, un correo verificado, el ISBN de un libro, el IBAN de una cuenta. Cuando dos sistemas registran el mismo DNI, la identidad se resuelve sola: no hay nada que adivinar, porque ambos están citando la misma fuente de verdad externa. La clave natural es, en términos de la analogía, una dirección *canónica* que todo el mundo acordó usar.

TRIPLETAS

```
(cliente_1042,  documento_dni, "20148833")   ∈ M(Q, K)   // la tienda lo registró
(paciente_vega, documento_dni, "20148833")   ∈ M(Q, K)   // la clínica también
// misma clave natural ⇒ misma identidad: la resolución es deductiva, no probable
```

Vía 2 · `mismo_que` declarado como un hecho más

A veces no hay clave común, pero alguien (un humano, un proceso de conciliación, una migración) *sabe* que dos identificadores coinciden y quiere dejarlo escrito. El modelo no necesita maquinaria nueva para eso: la identidad compartida se guarda como cualquier otro hecho, una tripleta con el cable `mismo_que`. Es la elegancia del enfoque: una afirmación de identidad es un dato de primera clase, fechable, atribuible y revisable como cualquier otro.



Porque es un hecho como los demás, puede reificarse cuando importa *quién* lo afirmó y *cuándo* (exactamente las razones de reificación del capítulo 9). Una fusión de identidades en un banco no es trivial: conviene saber qué operador la aprobó, con qué evidencia y en qué fecha, por si mañana hay que deshacerla.

`OWL:SAMEAS` Y LA RESOLUCIÓN DE ENTIDADES (22)

Esta idea no es nueva ni exclusiva del modelo. La Web Semántica la formalizó hace dos décadas con `owl:sameAs`, el predicado del estándar OWL que declara que dos URIs denotan el mismo individuo; es el motor que permite a un dato de DBpedia[33] enlazarse con su gemelo en Wikidata[32]. Nuestro `mismo_que` es, deliberadamente, ese mismo gesto traducido al vocabulario de las preguntas.

El problema más amplio (decidir cuándo dos registros son la misma cosa) tiene nombre propio en la literatura: *entity resolution* (también *record linkage* o *deduplication*), estudiado desde el trabajo seminal de Fellegi y Sunter en 1969. El modelo no lo reinventa; le da un lugar limpio donde aterrizar.

Vía 3 · *Matching* probabilístico

El caso incómodo, y el más frecuente en la vida real: no hay clave común y nadie declaró nada. La tienda guardó «Juan Vega», la clínica «J. Vega Ramos», la municipalidad «Vega, Juan A.». ¿Son el mismo? Probablemente. Pero «probablemente» es una palabra peligrosa en una base de datos. Aquí la identidad se *infiere* comparando rasgos (nombre, fecha de nacimiento, dirección, teléfono) y calculando una probabilidad de coincidencia. Por encima de cierto umbral, el sistema propone que son la misma persona; por debajo, los deja separados.

EL *MATCHING* PROBABILÍSTICO ES UN RETO DE TOOLING, NO DE MODELO

El modelo te da *dónde* escribir la conclusión (una tripleta `mismo_que`, idealmente reificada con su grado de confianza) pero no resuelve por ti el cálculo. Emparejar a ciegas funde a dos personas distintas; ser demasiado cauto deja la historia partida en pedazos. Es un problema abierto de herramientas e infraestructura, y por eso lo retomamos sin rodeos en el capítulo 30, «Qué falta», entre las piezas que aún hacen falta para que esto sea de uso diario.

El modelo: una persona canónica con identidades colgando

Con las tres vías sobre la mesa, el patrón de modelado es directo. Existe una **identidad canónica** (un único nodo en Q, `persona_vega`) y, a su alrededor, los identificadores locales de cada sistema, unidos a ella por `mismo_que`. Cada sistema sigue usando su rótulo de siempre, sin renunciar a nada; el grafo se encarga de que todos esos rótulos converjan en una sola persona.

TRIPLETAS

```
(persona_vega,  instancia_de, persona)      ∈ M(Q, K)   // la identidad canónica
(persona_vega,  documento_dni, "20148833")  ∈ M(Q, K)   // su clave natural

(cliente_1042,        mismo_que, persona_vega)  ∈ M(Q, Q)   // alias de la tienda
(paciente_vega,       mismo_que, persona_vega)  ∈ M(Q, Q)   // alias de la clínica
(contribuyente_77_3389, mismo_que, persona_vega) ∈ M(Q, Q)  // alias de la municipalidad

// y los hechos de cada sistema cuelgan de su propio puntero, como siempre:
(venta_001,  comprador, cliente_1042)   ∈ M(O, Q)   // la tienda vendió la camiseta_88
(urgencias_2026_071, paciente, paciente_vega) ∈ M(O, Q)   // la clínica abrió el episodio
```

Observa el reparto de trabajo. Cada sistema escribe sus hechos contra *su* puntero, como lo ha hecho siempre, sin coordinarse con nadie: la tienda no necesita saber que la clínica existe. La única tripleta que cruza fronteras es la afirmación de identidad. Resolver un grafo de muchos sistemas no exige reescribir los datos de cada uno: exige tender los hilos `mismo_que` que faltan. El resto del tejido ya estaba.

Reconciliar que el Juan Vega de la tienda, el de la clínica y el de la municipalidad son uno solo no sirve únicamente para ese Juan. Hecho a escala, es lo que permite el padrón único: cruzar los registros de todas las personas repartidas entre sistemas y contarlas

una sola vez, sin duplicados.

DIRECCIÓN DEL CABLE

Por convención, el alias apunta *hacia* la identidad canónica (`cliente_1042 → persona_vega`), no al revés. Así el nodo canónico no necesita enumerar a sus alias: para encontrarlos basta seguir las flechas hacia atrás, igual que se rastrean las referencias a un objeto.

No es una decisión numerada: es una convención de fondo

Vale la pena ser explícito sobre el estatus de esto. La resolución de identidad no introduce una regla de diseño nueva al modo de las que hemos ido numerando (no hay aquí ninguna maquinaria que no existiera ya). Es, más bien, una **convención fuerte** que se desprende de piezas que el lector ya conoce: la entidad del eje Q como referente, el cable como referencia, la reificación como dirección estable. La identidad emerge de juntarlas bien, no de añadir nada.

PRINCIPIO DE IDENTIDAD CANÓNICA

Cada individuo del mundo se modela como **una sola identidad canónica** en el eje Q. Los identificadores que cada sistema le asigna son punteros locales que apuntan a ella mediante `mismo_que`. Un sistema escribe sus hechos contra su propio puntero; el grafo los reconcilia siguiendo los alias hasta la identidad canónica. Resolver la identidad *no* es fundir datos: es declarar referencias.

Por qué esto sostiene toda la promesa del libro

Detente a imaginar la consulta que da sentido a este libro entero: *«muéstrame todo lo que el sistema sabe de Juan Vega»*. Sus compras en la tienda, su episodio en urgencias, sus arbitrios en la municipalidad (tres dominios que nunca fueron diseñados para hablarse) reunidos en una sola respuesta coherente. Esa consulta es la prueba viviente de la interoperabilidad que la introducción prometió. Y es *imposible* sin resolución de identidad: sin las flechas `mismo_que`, la pregunta «¿qué sabemos de Juan?» se rompe en tres preguntas que ningún sistema puede juntar, porque ninguno reconoce al Juan del otro.

De aquí brota un principio que reaparecerá con fuerza cuando lleguemos a la municipalidad: el principio de **«una sola vez»**. Si la clínica ya verificó la dirección de Juan, la municipalidad no debería volver a pedírsela; si la tienda ya tiene su correo, ningún tercer sistema tendría por qué exigirlo de nuevo. El dato vive una vez, en el individuo canónico, y todos los sistemas que lo conocen lo consultan allí.

Ese principio («no vuelvas a pedir lo que otro sistema ya sabe») es la cara amable de la resolución de identidad para el ciudadano, el paciente, el cliente. Lo que para el ingeniero es un puntero compartido, para la persona es no tener que llenar el mismo formulario tres veces. La identidad bien resuelta no es una sutileza técnica: es lo que convierte un montón de sistemas aislados en algo que, por fin, se comporta como si supiera quién eres.

> *Un identificador es un papelito que dice dónde mirar; la identidad es lo que encuentras al mirar. El error de toda torre de Babel es confundir el papelito con la persona.*

LA TESIS DEL CAPÍTULO

EN LA PRÁCTICA

Cuando integres un sistema nuevo al grafo, no intentes renumerar sus identificadores: déjalos como están y tiende los `mismo_que`. Prioriza siempre la *vía 1* (busca una clave natural con autoridad externa antes que cualquier otra cosa), porque convierte la resolución en una deducción y no en una apuesta.

Reserva el *matching* probabilístico para cuando no quede alternativa, y cuando lo uses, **guarda la confianza**: reifica el `mismo_que` con quién lo afirmó, cuándo y con qué grado de certeza. Una fusión de identidades que no puedes auditar es una fusión que no puedes deshacer.

Con esto, la Parte III queda completa: tenemos el hecho atómico, el espacio donde vive, las situaciones que lo enriquecen, la causalidad que lo enlaza y, ahora, la identidad que permite que todo eso cruce de un sistema a otro sin perderse. Lo que sigue es tender la mano a los mundos vecinos (los objetos de la programación, los bits, los grafos, las cadenas de bloques) y mostrar que el modelo no es una isla, sino un puente.

12

Puentes: objetos, bits, grafos y cadenas

Nada de esto es magia, y nada es del todo nuevo. WQuestions se asienta sobre cuatro ideas que el lector técnico ya domina (la programación orientada a objetos, el bit, el grafo y la cadena de bloques). Este capítulo tiende los puentes, y marca dónde el puente sostiene y dónde se rompe.

Imagina a un programador con quince años de oficio leyendo este libro por encima del hombro de un colega. Hacia el capítulo 7 levanta una ceja. «¿Tripletas? —dice—. Eso ya existe. Yo modelo con clases y objetos desde que aprendí a programar; mis datos viven en tablas que se consultan en milisegundos; y eso de los hechos que no se pueden borrar suena a blockchain, una moda que ya pasó.» Tiene razón en cada observación, y por eso vale la pena tomárselo en serio. WQuestions no se inventó en un vacío: respira el mismo aire que la programación orientada a objetos, los bits, las bases de grafos y los libros mayores inmutables. Este capítulo se dedica a contestarle, una objeción a la vez.

El método será siempre el mismo. Para cada tecnología familiar tenderemos un **puente** (un mapa que muestra qué pieza de WQuestions corresponde a qué pieza que el lector ya conoce) y luego marcaremos el punto exacto donde el puente deja de sostener peso. Porque la analogía sirve para entrar, pero engaña si se la lleva demasiado lejos. Cuatro puentes, cuatro secciones.

> **PARA QUIÉN**
> Si no programas, puedes leer este capítulo en superficie: cada sección abre con la idea intuitiva. Si vienes del oficio, las tablas y los bloques de código están escritos para ti, y conviene que los discutas con tu propio escepticismo a cuestas.

Puente uno · Objetos: la programación orientada a objetos

La primera reacción de cualquier ingeniero es la más natural: «esto ya lo hago con clases». Y en parte lleva razón. La programación orientada a objetos (POO) lleva medio siglo organizando el software alrededor de la misma intuición que WQuestions (que el mundo se describe con *cosas* que pertenecen a *categorías* y que se *relacionan* entre sí). El parecido es tan profundo que conviene hacerlo explícito antes de señalar la grieta.

Tres conceptos de la POO tienen un gemelo casi perfecto en el modelo. Cuando un programador escribe `v = Venta(...)`, está *instanciando*: fabricando un individuo a partir de un molde. Nosotros decimos exactamente lo mismo con el cable `instancia_de`, que ancla un sujeto del eje O en su clase del eje K. Cuando una clase `VentaOnline` *hereda* de `Venta`, declara que todo lo que vale para una venta vale para una venta en línea; nosotros lo escribimos con `subtipo_de` entre dos conceptos del eje K. Y los *atributos* y *asociaciones* de un objeto (su `monto`, su `vendedor`) no son otra cosa que nuestros predicados del eje M.

TRIPLETAS

```
(venta_online,  subtipo_de,    venta)          ∈ M(K, K)   // herencia
(venta_001,     instancia_de,  venta_online)   ∈ M(O, K)   // instanciación
(venta_001,     vendedor,      vendedor_17)     ∈ M(O, Q)   // asociación
(venta_001,     monto_usd,     49.90)           ∈ M(O, N)   // atributo
```

Para fijar el paralelo, traduzcamos la misma entidad (nuestra venta canónica) a una clase de Python y, justo al lado, a las tripletas equivalentes. Mira primero el código que el programador escéptico reconoce a primera vista:

PYTHON

```python
class Venta:
    def __init__(self, id, vendedor, cliente, articulo, monto, momento):
        self.id       = id
        self.vendedor = vendedor      # asociación  → Vendedor
        self.cliente  = cliente       # asociación  → Cliente
        self.articulo = articulo      # asociación  → Articulo
        self.monto    = monto         # atributo    (float)
        self.momento  = momento       # atributo    (datetime)

    def aplicar_descuento(self, pct):     # comportamiento, acoplado al dato
        self.monto *= (1 - pct / 100)

v = Venta("venta_001", vendedor_17, cliente_1042,
          camiseta_88, 49.90, "2026-05-14T16:32")
```

Y ahora la misma venta como hechos atómicos. Fíjate en que *no hay método*: el dato está separado de lo que se le hace.

TRIPLETAS

```
(venta_001, instancia_de, venta)          ∈ M(O, K)
(venta_001, vendedor,     vendedor_17)    ∈ M(O, Q)
(venta_001, cliente,      cliente_1042)   ∈ M(O, Q)
(venta_001, objeto,       camiseta_88)    ∈ M(O, O)
(venta_001, monto_usd,    49.90)          ∈ M(O, N)
(venta_001, momento,      2026-05-14T16:32) ∈ M(O, T)
```

Línea por línea, los atributos de la clase reaparecen como tripletas. La figura siguiente pone las dos representaciones frente a frente: a la izquierda, el objeto como caja cerrada con sus casillas; a la derecha, el mismo hecho desplegado como estrella de nodos en el grafo.

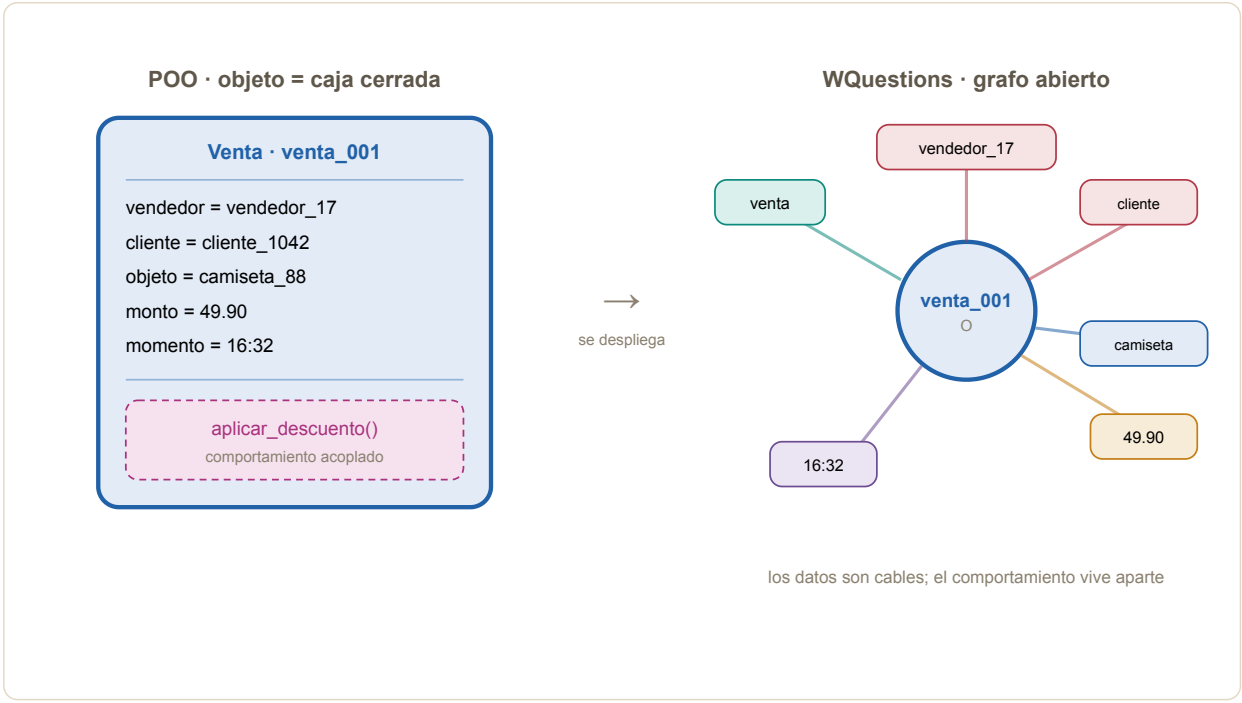


Figura 12.1. La misma venta, dos veces. A la izquierda, el objeto de la POO: una caja que encierra datos *y* el método que los manipula, con su esquema fijado en el código. A la derecha, la venta desplegada como grafo: cada atributo es un cable tipado y el comportamiento ya no vive dentro del dato.

Dónde se rompe el puente

La analogía es buena para entrar, y traicionera si se la lleva más allá. Hay dos diferencias que no son de grado, sino de naturaleza, y conviene nombrarlas con precisión.

La primera: la POO acopla datos y comportamiento; WQuestions los desacopla. Un objeto `Venta` guarda el monto *y* sabe aplicarse un descuento: el dato y el código que lo toca viven en la misma caja. Eso es elegante para programar, pero ata la información a un lenguaje y a una versión del software. Nuestras tripletas no saben hacer nada: son afirmaciones inertes sobre el mundo. El comportamiento (cómo calcular un descuento, cómo validar una venta) vive en otra capa, separado del hecho. Un dato que no carga lógica es un dato que cualquier sistema puede leer sin importar en qué lenguaje fue escrito.

La segunda: la POO fija el esquema en el código (mundo cerrado); WQuestions es de mundo abierto. En cuanto compilas la clase `Venta`, sus campos quedan tallados en piedra. Si mañana el negocio quiere registrar el lote de inventario del artículo, alguien debe abrir el archivo, añadir un atributo, recompilar y migrar los datos viejos. Añadir un cable nuevo a un sujeto del grafo, en cambio, es escribir una tripleta más; no hay esquema que recompilar porque el esquema nunca estuvo cerrado. Esta es, en una frase, la diferencia entre un mundo que solo admite lo previsto y uno que admite lo que la realidad traiga.

> **IDEA CLAVE**
>
> La POO encierra datos y comportamiento en una caja cuyo esquema se fija al compilar (**mundo cerrado**). WQuestions deja los datos como tripletas inertes y abiertas, y manda el comportamiento a otra capa (**mundo abierto**). No es que una sea correcta y la otra no: sirven a propósitos distintos. La POO optimiza la ejecución de un programa; WQuestions optimiza que muchos programas distintos compartan los mismos hechos.

Un último hilo conecta esta sección con el capítulo anterior. En la POO, dos variables que apuntan al mismo objeto comparten su *referencia* (ese puntero invisible que distingue «el mismo objeto» de «un objeto igual»). Esa referencia es, ni más ni menos, lo que en el capítulo 11 llamamos el identificador único: el ancla que permite a un hecho hablar de `venta_001` y a otro, en otro sistema, hablar de la misma venta sin confundirla con su gemela. El puntero del objeto y el UID del grafo cumplen la misma función: dar identidad estable a una cosa.

Resumamos el puente entero en una sola tabla. A la izquierda, el vocabulario que el ingeniero ya domina; a la derecha, su correspondiente en el modelo.

PROGRAMACIÓN ORIENTADA A OBJETOS	WQUESTIONS	COMENTARIO
Clase	Concepto del eje K	El molde, la categoría atemporal.
Instanciación (`new Venta()`)	Cable `instancia_de`	Fabricar un individuo del molde.
Herencia (`extends`)	Cable `subtipo_de`	Especializar una categoría.
Atributo / campo	Predicado funcional del eje M	Un valor por sujeto.
Asociación / referencia	Predicado del eje M a otra entidad	Un cable hacia otro nodo.
Referencia de objeto (puntero)	Identificador único (UID)	Misma función: identidad estable.
Método (comportamiento)	— (vive en otra capa)	El modelo desacopla el dato del código.
Esquema fijado al compilar	Esquema abierto y extensible	Mundo cerrado vs. mundo abierto.

Puente dos · Bits

Pasada la primera objeción, el programador afina la segunda, y es más visceral. «Vale —dice—, entiendo el mapa con la POO. Pero descomponer cada venta en seis, diez, veinte tripletas parece un caos. Donde yo guardaba una fila ahora tengo veinte líneas. Eso es muchísimo más almacenamiento, y leer veinte líneas para entender una venta es agotador.» La objeción es legítima, y la respuesta exige cambiar de escala mental.

> *Las tripletas son al conocimiento lo que los bits son a todo lo digital: atómicas, uniformes, abrumadoras de a una, e imprescindibles en masa.*

EL PRINCIPIO DEL BIT

Detengámonos en esa frase, porque es la clave de la sección. Una fotografía, vista por dentro, son millones de unos y ceros. A ningún ingeniero se le ocurriría quejarse de que «un bit no dice nada» o de que «son demasiados para leerlos». El bit no se diseñó para que un humano lo lea de a uno: se diseñó para ser la unidad *uniforme* y *atómica* sobre la que se construye todo lo digital, precisamente porque su simplicidad permite operar billones de ellos a velocidad de máquina. La tripleta aspira a ser exactamente eso para el conocimiento: el átomo uniforme de la digitalización del *significado*.

Figura 12.2. Una foto se digitaliza en millones de bits; una escena se digitaliza en miles de tripletas. En ambos casos, la unidad es minúscula y abrumadora de a una, pero es justo esa atomicidad uniforme la que vuelve la información operable a escala de máquina.

Queda el miedo al almacenamiento, que es el más concreto. ¿No ocupa una barbaridad guardar cada hecho como una tripleta de texto larga, con su sujeto y su predicado repetidos miles de veces? La respuesta es que *no*, y la razón merece detallarse, porque deshace un malentendido común.

Las bases especializadas en tripletas (los *triple stores*) jamás guardan el texto literal que tú escribes. Aplican **codificación por diccionario**: cada cadena que se repite (un predicado como `monto_usd`, un valor como `vendedor_17`) se almacena *una sola vez* en una tabla y, de ahí en adelante, se referencia por un número entero corto. Donde tú ves `vendedor_17` mil veces, la máquina ve un entero mil veces (y los enteros se comprimen casi a la nada cuando se repiten). A eso se suma el **almacenamiento columnar**, que guarda juntos todos los valores de una misma posición, donde la redundancia es máxima y la compresión, brutal.

> **EN LA PRÁCTICA**
>
> El predicado `instancia_de` puede aparecer en cien millones de tripletas. En disco no son cien millones de copias de la palabra: es *un* entero, repetido en una columna que el compresor reduce a casi nada. La máquina nunca «lee» la tripleta como texto humano; opera sobre enteros. Lo que para ti es prosa, para ella son índices. El temor al tamaño nace de imaginar el grafo como un archivo de texto gigante, y esa imagen es falsa.

El programador escéptico empieza a aflojar. Las tripletas no son un derroche: son la versión del bit aplicada al significado, y la industria del almacenamiento ya resolvió hace décadas el problema de comprimir cosas atómicas y repetidas. Lo cual nos lleva, naturalmente, a las bases de datos que hacen justo esto.

Puente tres · Grafos: bases ya no relacionales, sino dirigidas al dato

«Está bien —concede el ingeniero—, pero todo esto, ¿en qué motor lo corro? Yo tengo PostgreSQL.» Aquí la respuesta es la más alentadora de todo el capítulo: no hace falta inventar nada. La industria lleva veinte años construyendo bases de datos que abandonaron el modelo de tablas para abrazar el modelo de la **red de nodos y enlaces** (el grafo). WQuestions no compite con ellas; encuentra en ellas su hogar natural. El contraste con el modelo relacional ya lo desarrollamos en el capítulo 8: aquí miramos las alternativas que sí encajan.

Esas tecnologías se agrupan en tres familias, y conviene conocerlas por su nombre, porque la afinidad con WQuestions varía mucho de una a otra.

Triple / RDF[8] stores. Son las más cercanas en espíritu: guardan el mundo, literalmente, como tripletas `(sujeto, predicado, objeto)`, igual que nosotros. Nacieron del estándar RDF de la web semántica. En esta familia están **Apache Jena** (con su servidor Fuseki), **GraphDB**, **Blazegraph**, **Virtuoso**, **Amazon Neptune** y **Stardog**. Si WQuestions tuviera que elegir un motor de fábrica, sería uno de estos: la unidad de almacenamiento *ya es* la tripleta.

Property graphs (grafos de propiedades). En vez de tripletas puras, modelan el mundo como nodos y aristas a los que se les cuelgan propiedades. Son extraordinariamente rápidos para recorrer relaciones («amigos de amigos», rutas, cadenas de suministro). Los referentes son **Neo4j**, **Memgraph** y **TigerGraph**. Encajan bien, aunque su modelo de propiedades en la arista exige una traducción menor respecto de nuestra tripleta canónica.

Datalog, inmutables y temporales. Aquí ocurre la coincidencia más asombrosa del capítulo. **Datomic** y **XTDB** guardan la información como *datoms*: hechos de la forma `[entidad, atributo, valor, tiempo-de-transacción]`. Léelo de nuevo. Es nuestra tripleta (entidad, predicado, valor) más una marca temporal. Casi exactamente lo que el modelo propone. Y XTDB es **bitemporal de fábrica**: distingue de raíz entre cuándo ocurrió un hecho en el mundo y cuándo el sistema se enteró, que es justo la distinción de la vigencia (la regla D6 del capítulo 9).

> **CONVERGENCIA**
>
> Que dos comunidades independientes (la de RDF y la de Datomic) hayan llegado, por caminos distintos, a una unidad `(entidad, atributo, valor [, tiempo])` es la mejor señal de que la tripleta no es un capricho del modelo, sino una forma natural a la que el problema empuja por sí solo.

El *datom* de Datomic merece quedar escrito al lado de nuestra tripleta, porque el parecido es difícil de exagerar:

TRIPLETAS

```
// WQuestions — tripleta tipada
(venta_001, monto_usd, 49.90)                        ∈ M(O, N)

// Datomic / XTDB — datom: tripleta + tiempo de transacción
[venta_001  :monto_usd  49.90   tx-2026-05-14T16:32]

// la única diferencia: ellos añaden el "cuándo lo supo el sistema" de fábrica
```

Pongamos las tres familias en una tabla, marcando con claridad cuáles son las más afines al modelo. La afinidad alta significa que adoptar WQuestions sobre ese motor casi no exige traducción.

FAMILIA	UNIDAD DE DATO	EJEMPLOS	AFINIDAD CON WQUESTIONS
Triple / RDF stores	Tripleta `(s, p, o)`	Jena/Fuseki, GraphDB, Blazegraph, Virtuoso, Amazon Neptune, Stardog	**Muy alta** · la unidad ya es la tripleta
Datalog · inmutables · temporales	Datom `[e, a, v, t]`	Datomic, XTDB (bitemporal de fábrica)	**Máxima** · tripleta + tiempo, casi idéntica
Property graphs	Nodo + arista con propiedades	Neo4j, Memgraph, TigerGraph	Alta · recorrido veloz, traducción menor
Relacional (referencia, ver cap. 8)	Fila en una tabla de esquema fijo	PostgreSQL, MySQL, Oracle	Posible, pero forzada · tablas rígidas

PRECEDENTE

El modelo de datos de **Datomic**, ideado por Rich Hickey, y su sucesor de código abierto **XTDB** son los precedentes industriales más directos de WQuestions en lo que toca al almacenamiento. Comparten tres convicciones profundas: el dato es una tripleta (allí, un *datom*); los hechos no se sobrescriben sino que se acumulan; y el tiempo es un eje de primera clase, no una columna más. Donde WQuestions añade valor sobre ellos es en la *semántica*: tipar cada predicado por los ejes de las preguntas, de modo que el grafo no solo sea consultable, sino interoperable entre dominios y entendible por una IA sin manual.

Puente cuatro · Cadenas: lo que tomamos de la blockchain

Queda la objeción que el programador soltó al principio, casi con desdén: «esto de los hechos que no se borran suena a blockchain, y eso fue una burbuja». Aquí conviene separar la tecnología de moda de la *idea* que la hizo valiosa, porque la idea es excelente y WQuestions la hereda casi de oficio.

Lo notable es que el modelo ya tiene, sin haberlo buscado, los dos cimientos de un libro mayor al estilo blockchain. El primero: los **hechos son inmutables**. En WQuestions nunca se borra ni se sobrescribe nada; corregir el mundo significa *afirmar un hecho nuevo* que deja en evidencia al anterior, no pisarlo. El segundo: la **bitemporalidad** de la regla D6 (capítulo 9), que fecha cada situación con su rango de vigencia. Junta esas dos piezas y obtienes, sin agregar línea alguna, un **libro mayor de solo-anexar** (*append-only*) y auditable: la historia completa de cómo el sistema fue creyendo lo que cree, conservada intacta.

IDEA CLAVE

Inmutabilidad de los hechos + bitemporalidad (D6) = un libro mayor *append-only* y auditable, **gratis**. No es algo que WQuestions deba construir: es lo que ya es. La blockchain hizo célebre la idea de un registro que solo crece y nunca se reescribe; el modelo la tenía desde su primera decisión de diseño.

Sobre esa base, importar el resto de la maquinaria de la blockchain es directo, y son tres piezas que se agregan *encima* de lo que ya existe.

Procedencia por hecho. Cada tripleta puede llevar pegada su propia credencial: quién la afirmó, en qué momento, y una firma criptográfica que lo prueba. No es metadato cosmético —es la diferencia entre un dato y un dato *de quién responde por él*. Esto se modela, como todo, reificando: el acto de afirmar se vuelve una situación con su agente, su instante y su firma.

JSON

```
{
  "hecho":      ["venta_001", "monto_usd", 49.90],
  "afirmado_por": "vendedor_17",
  "momento":      "2026-05-14T16:32:00-05:00",
  "firma":        "0x9f2a…c4",
  "hash_previo":  "0x71be…08",
  "hash":         "0xa3d0…11"
}
```

Hash-encadenado (Merkle). Si cada hecho guarda el *hash* del anterior (como en el campo `hash_previo` de arriba), la cadena entera se vuelve evidencia de manipulación: alterar un hecho viejo rompe todos los hashes que vinieron después, y la falsificación salta a la vista al instante. Es la misma garantía criptográfica que sostiene una blockchain, aplicada a nuestro libro de hechos.

Consenso y federación. Cuando el grafo deja de pertenecer a una sola empresa y pasa a ser compartido entre varias (el sueño de la interoperabilidad con el que abrió este libro), hace falta un protocolo que decida qué hechos entran al registro común y quién puede afirmar qué. Ahí entran las ideas de consenso y federación. Pero eso ya es terreno de la seguridad del grafo compartido, que el capítulo 29 trata a fondo.

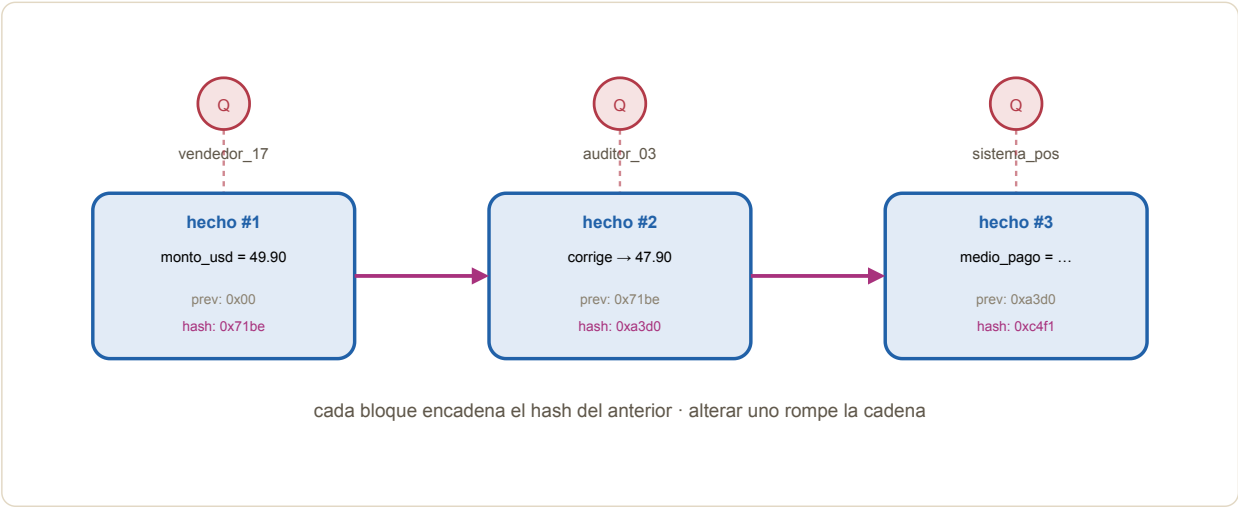


Figura 12.3. Tres hechos sucesivos sobre una misma venta, cada uno firmado por su agente (arriba) y encadenado por el hash del anterior. El hecho #2 corrige el monto sin borrar el #1; el #3 añade información. Manipular cualquiera de ellos rompería el encadenamiento, y la alteración quedaría a la vista.

Nótese el detalle del hecho #2 en la figura: corrige el monto a 47,90 *sin borrar* el 49,90 original. Eso es exactamente la inmutabilidad del modelo en acción, anudada con la vigencia del capítulo 9: el hecho viejo cierra su rango, el nuevo abre el suyo, y

la cadena de hashes deja constancia firmada de quién hizo la corrección y cuándo. Un auditor que llegue dentro de cinco años podrá reconstruir la verdad completa.

El escéptico, convencido

Volvamos al programador del comienzo, que ya no levanta la ceja. Las cuatro objeciones tenían fundamento, y las cuatro tienen respuesta. WQuestions *se parece* a la POO en cómo piensa las cosas y las categorías, pero la supera al desacoplar el dato del comportamiento y abrir el esquema. *Se parece* a una avalancha de bits, y por la misma razón que los bits funcionan: la atomicidad uniforme es lo que vuelve la información operable y comprimible a escala. *Vive* en las bases de grafos que la industria ya construyó (y en Datomic y XTDB casi sin traducción). Y *hereda* lo mejor de la blockchain (el registro inmutable y auditable) sin pagar el precio de la moda.

> *WQuestions no pide tirar lo que ya sabes. Pide reconocer que las preguntas estaban debajo de todo eso desde el principio.*

EL SENTIDO DE LOS CUATRO PUENTES

Ninguno de estos puentes es decorativo. Cada uno responde a un lector que llega con su propia herramienta favorita y la pregunta legítima de «¿esto qué me aporta sobre lo que ya tengo?». La respuesta, repetida cuatro veces, es la misma: WQuestions no reemplaza tu pila tecnológica; le pone debajo una capa de significado (los ejes de las preguntas) que hace que tus objetos, tus bits, tus grafos y tus cadenas por fin se entiendan entre sí. Con los cuatro puentes tendidos, estamos listos para la última pieza del lenguaje a los hechos: cómo el verbo de una frase se convierte en la signatura tipada de un predicado, que es el asunto del capítulo siguiente.

13

El verbo como signatura

Antes de inventar un esquema de datos, el idioma ya traía uno. Cada verbo es una declaración de función disfrazada de palabra: dice qué situación crea, qué roles exige y de qué caja debe venir cada uno.

Corre el minuto 87 y la selección pierde por un gol. Un relator de radio describe la jugada para miles de personas que no pueden ver la cancha: *«Messi recibe de Di María en la frontal del área, recorta sobre la marca y* **marca** *el gol con la zurda»*. En el instante en que pronuncia ese verbo (*marca*) ocurre algo que damos por sentado y que, sin embargo, es casi milagroso: cada oyente reconstruye en su cabeza el mismo evento. Sabe que *hubo* un remate, que *alguien* lo ejecutó, que *algo* entró en el arco, que pasó *en* un sitio y *en* un momento. El verbo no transmitió una imagen; transmitió un molde, y cada cerebro lo rellenó.

> **HILO DEL LIBRO**
> Este es el capítulo bisagra de la Parte IV. En el capítulo 7 fijamos el átomo (la tripleta tipada, D3); en el capítulo 9, cómo un evento se reifica como situación con roles. Aquí mostramos por qué el lenguaje ya venía organizado de esa misma forma.

Ese molde es el tema de este capítulo. La tesis es sencilla de enunciar y tiene consecuencias enormes: **cada verbo es una firma de función**. No describe una acción suelta, sino un *tipo* de situación, y trae de fábrica una declaración de qué participantes admite y de qué eje debe venir cada uno. Si en los capítulos anteriores construimos un modelo de datos sobre las preguntas, ahora descubriremos que el idioma (sin que nadie se lo pidiera) ya estaba estructurado sobre esos mismos ejes. El verbo es la costura natural entre la lengua que hablamos y los datos que guardamos.

El argumento fantasma de Davidson

Para entender por qué un verbo es una *firma* y no una simple etiqueta, conviene empezar por un problema que parecía trivial y resultó ser un abismo. Tomemos un verbo de cine y una oración mínima:

> *Serra dirigió la película.*
>
> UNA ORACIÓN DE DOS ARGUMENTOS… O ESO PARECE

La tentación del ingeniero clásico es tratar el verbo como una función de dos casillas: `dirigir(serra, pelicula_marea)`. Limpio y cerrado. Pero el lenguaje no se queda quieto, y la realidad tampoco. Apenas añadimos las circunstancias que cualquier ficha técnica registra, la oración crece:

> *Serra dirigió la película en los estudios del norte, durante el otoño de 2025, con un presupuesto ajustado, para retratar el duelo de su madre.*

¿Qué hacemos con esto? ¿Convertimos `dirigir` en una función de seis casillas? ¿Y mañana, cuando alguien diga «con dos cámaras» o «bajo presión del estudio», la ampliamos a ocho? Bajo esa lógica, el verbo tendría que reescribir su definición interna cada vez que a un hablante se le ocurre un detalle nuevo. Es absurdo: una función cuya aridad depende del humor del narrador no es una función, es un caos.

PRECEDENTE · EL ARGUMENTO EVENTIVO DE DAVIDSON (12) **(1967)**

En *«The Logical Form of Action Sentences»*, el filósofo **Donald Davidson** propuso una salida elegante. En toda oración de acción existe un argumento que no pronunciamos pero que está presente: **el evento mismo**. La oración no afirma sólo que Serra y la película están en cierta relación; afirma que *existió un evento e de tipo dirigir*, y que ese *e* tuvo a Serra como quien lo realiza, a la película como lo realizado, a los estudios como lugar, al otoño como tiempo. En notación lógica, «dirigió … en … durante …» se vuelve `∃e [ dirigir(e) ∧ agente(e, serra) ∧ tema(e, pelicula) ∧ lugar(e, estudios) ∧ … ]`.

La consecuencia es decisiva: cada circunstancia es una *conjunción independiente* colgada del mismo evento. Añadir «para retratar el duelo» no cambia el verbo: agrega una cláusula más. El verbo conserva siempre su forma; los detalles se enchufan.

Una década más tarde, lingüistas como Terence Parsons llevaron la idea a su forma actual, la que hoy llamamos **neodavidsoniana**: incluso el sujeto y el objeto directo dejan de ser argumentos «del verbo» y pasan a engancharse al evento mediante roles explícitos (*agente*, *tema*, *lugar*). El verbo deja de ser un predicado con casillas fijas y se convierte en algo más limpio: un predicado de un solo argumento, el evento, alrededor del cual orbitan los participantes.

Quien haya leído el capítulo 9 reconocerá el dibujo de inmediato. Lo que la filosofía del lenguaje llamó «argumento eventivo» es exactamente lo que nosotros llamamos **reificación**: convertir la acción en un individuo del eje O y colgarle sus participantes como hechos atómicos. No es una analogía forzada. Es la misma operación, descrita dos veces: una por los lógicos en 1967, otra por nosotros desde la ingeniería de datos. Cuando dos disciplinas que no se hablan llegan a la misma estructura, conviene prestar atención.

El verbo no es una acción: es un molde

De aquí sale el primer concepto que conviene fijar. Un verbo como *marcar*, *dirigir* o *llover* no nombra una acción que ocurrió una vez en el mundo. Nombra un **molde**: un tipo, una categoría de situación. Por eso los verbos no viven en el eje de las cosas que pasan (O), sino en el eje de los conceptos (K), tal como fijamos en la primera decisión de diseño del libro: en K viven las plantillas atemporales; en O, las instancias situadas.

Cuando alguien habla y emplea un verbo en una oración concreta, ocurren dos cosas, casi instantáneas y rigurosamente mecánicas:

1 SE FABRICA UNA INSTANCIA

El sistema crea una situación nueva y fresca en la caja O, con su propio identificador único. Es el evento *e* de Davidson, ahora con código de barras.

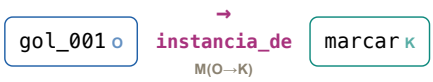
2 SE LA CLASIFICA

El sistema conecta esa instancia con su molde en K para registrar de qué *tipo* de situación se trata. Ese cable se llama `instancia_de`.

TRIPLETAS

```
(gol_001)                          ∈ O          // la instancia fresca
(gol_001, instancia_de, marcar)    ∈ M(O, K)    // su molde vive en K
```

Ese segundo hecho (el cable que une la instancia con su molde) es la primera tripleta de toda situación, y conviene verla con la forma insignia del modelo:



A quien programe, esto le sonará idéntico a la relación entre una clase y un objeto. El verbo *marcar* es la función (o la clase) que vive en la biblioteca; cada vez que un hablante la usa, el sistema la *invoca* y crea una ejecución concreta en memoria, anotando

qué participantes recibió. El verbo permanece; las instancias se multiplican.

La signatura: un contrato de tipos

Si el verbo es una función, entonces tiene una **signatura**: la declaración formal de qué argumentos acepta, cuáles son obligatorios, cuáles opcionales y —esto es lo decisivo— de qué *tipo* es cada uno. En un lenguaje de programación moderno escribiríamos algo como esto, y nadie se sorprendería:

FIRMA DEL VERBO · MARCAR

```
def marcar(agente: Q, tema: O, *, lugar: L = None,
           momento: T = None, asistido_por: Q = None) -> Situacion:
    """Crea una situación de tipo 'marcar' y le engancha sus participantes."""
```

Cada parámetro lleva su anotación de tipo, y ese tipo es uno de nuestros ejes. El *agente* debe venir de la caja de los que actúan (Q); el *tema*, de la caja de las cosas y eventos (O); el *lugar*, de la caja de las locaciones (L). Los parámetros con valor por defecto son los opcionales. Esto no es una metáfora suelta: es, literalmente, un **contrato de tipos** entre la oración y el almacén de datos.

DEFINICIÓN · SIGNATURA DE UN VERBO

La **signatura** (o *firma tipada*) de un verbo es la declaración del conjunto de roles que admite, cuáles son obligatorios y cuáles opcionales, y de qué eje debe provenir el valor de cada uno. Activar un verbo significa crear una instancia de su molde y enganchar valores a esos roles, respetando los tipos. Es el mismo contrato que valida una llamada a función: argumentos faltantes o del tipo equivocado se rechazan en la puerta.

Cada verbo del idioma tiene la suya. Algunas son austeras y solitarias. El verbo *llover*, por ejemplo, no exige agente (nadie «hace» la lluvia) y deja todo lo demás en opcional:

FIRMAS · DE LA MÁS SIMPLE A LA MÁS COMPLEJA

```
llover  ( momento?: T, lugar?: L, intensidad?: N )                      → Situacion
soñar   ( experimentador: Q, tema: O, momento?: T )                     → Situacion
marcar  ( agente: Q, tema: O, lugar?: L, momento?: T, asistido_por?: Q )  → Situacion
dirigir ( agente: Q, tema: O, lugar?: L, momento?: T, con_finalidad?: O )  → Situacion
```

El signo de interrogación marca lo opcional. Nótese que *soñar* no pide un *agente* sino un *experimentador*: quien sueña no actúa sobre el mundo, vive un estado mental. La signatura captura esa diferencia fina con un rol distinto, y veremos en un momento de dónde sale ese catálogo de roles.

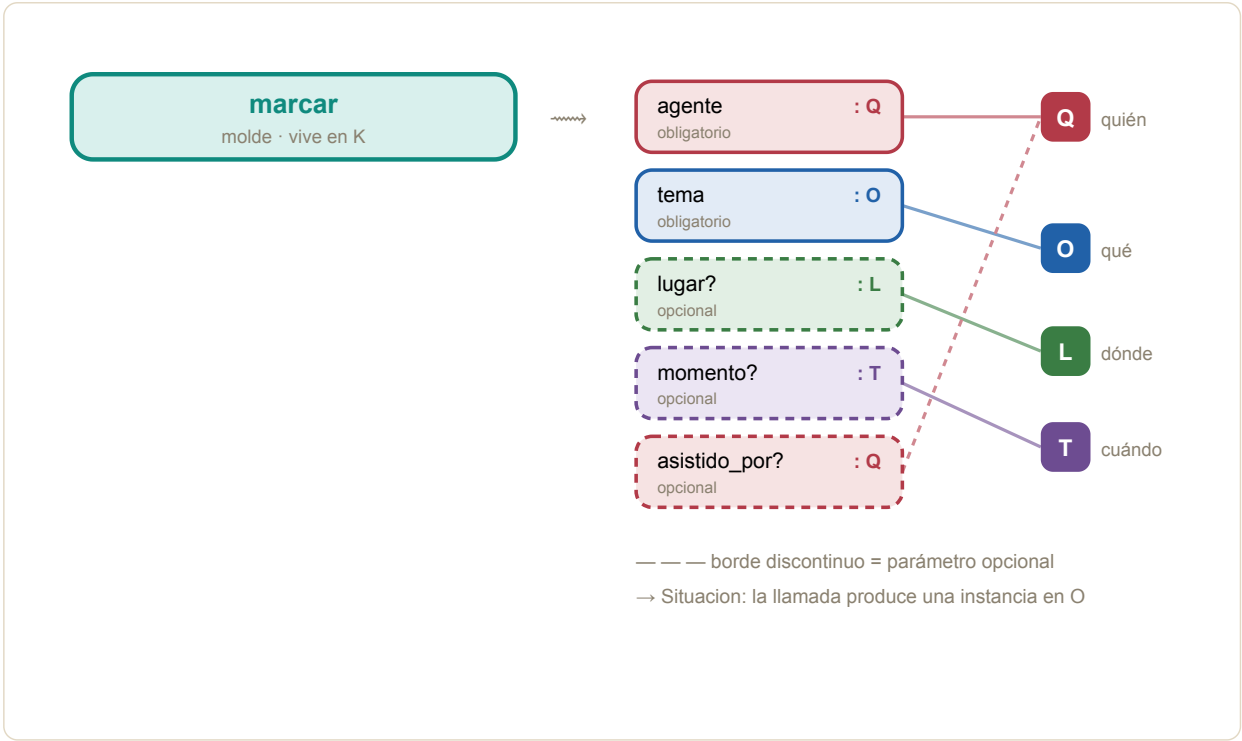


Figura 13.1. El verbo `marcar` como firma de función. La caja izquierda es el molde en K; al invocarlo despliega sus ranuras tipadas. Cada ranura declara su eje de destino (el *agente* sólo admite valores de Q, el *lugar* de L). El borde discontinuo marca los roles opcionales.

Hay una consecuencia que no salta a la vista en un caso suelto. Si cada verbo activa un tipo de situación con sus roles, entonces el verbo es también la llave de un agregado: «todas las situaciones que activó *marcar*» es, literalmente, el conjunto de todos esos goles registrados, listo para contarse o filtrarse. Elegir bien el verbo no solo escribe el hecho de hoy; define de antemano el conjunto sobre el que algún día se reportará.

Tres superpoderes que regala el contrato

Tratar el verbo como un contrato de tipos (y no como una palabra de diccionario) no es un adorno conceptual. Concede tres garantías que, en una base de datos tradicional, costarían cientos de líneas de validación escritas a mano.

🛡️

RECHAZO DE SINSENTIDOS

Si alguien intenta registrar *«la cancha marcó el gol»*, el sistema aborta antes de guardar nada. La firma de *marcar* exige que el *agente* venga de Q, y una cancha vive en L. No hace falta saber de fútbol: la matemática del tipo lo frena en seco.

🌊

TOLERANCIA A LO INCOMPLETO

Decir sólo *«llovió»* es una oración perfecta. No obligamos a nadie a rellenar *dónde* y *cuándo* si no le interesan: la firma los declara opcionales. La base de datos traga información parcial sin romperse.

🔌

MODIFICADORES CON BORDE

Marcar admite un *asistido_por*; *llover* admite una *intensidad*. Si alguien escribe *«marcó con una intensidad de tres milímetros por hora»*, el sistema lo rechaza. La firma decide qué modificadores son legales, sin reglas escritas caso por caso.

Estas tres propiedades son, en el fondo, la misma propiedad mirada desde tres ángulos: el tipo del argumento contiene la verdad sobre lo que es admisible. El verbo, al declarar sus tipos, hace de aduana. Y como la aduana es declarativa (vive en la firma, no en el código de la aplicación), vale para cualquier dominio sin que nadie reescriba el motor.

De dónde salen los roles: un catálogo cerrado

Habrás notado que las palabras `agente`, `tema`, `experimentador`, `lugar` no aparecen al azar: se repiten de verbo en verbo. No las inventamos en cada ejemplo. Provienen de un **catálogo cerrado y deliberadamente pequeño** de roles (los *cables conectores* del eje M), heredado casi sin cambios de lo que la lingüística lleva medio siglo refinando bajo el nombre de *roles temáticos*.

> **PRECEDENTE · FRAMENET (14) Y VERBNET (15) : EL VERBO TRAE SUS PAPELES**
>
> La intuición de que un verbo predefine sus participantes no es nuestra. El proyecto **FrameNet** (Charles Fillmore[24], Universidad de California en Berkeley) parte de una idea hermosa: comprender una palabra es invocar un *marco* (*frame*), una escena estereotipada con sus papeles. El marco *Commerce_buy* trae de fábrica sus huecos (comprador, vendedor, mercancía, dinero) y verbos como *comprar*, *adquirir* o *pagar* son distintas maneras de iluminar la misma escena.
>
> **VerbNet** (Karin Kipper-Schuler) lo lleva al plano operativo: clasifica miles de verbos del inglés en clases que comparten roles temáticos y restricciones de selección (exactamente lo que aquí llamamos *tipo del argumento*). Nuestro catálogo es un primo cercano y minimalista de estos dos: en lugar de miles de marcos, un núcleo de unas pocas decenas de roles que se reutilizan en todos los dominios.

¿Cuántos roles hacen falta para modelar el mundo? Menos de los que uno imagina. El núcleo canónico ronda las cuatro decenas de roles, y con ellos alcanza para describir fútbol, cine, medicina, banca, química o derecho. La razón es la misma que hace universales a las preguntas: el cerebro humano, sea cual sea la profesión de su dueño, procesa la realidad con un repertorio reducido y estable de papeles. He aquí las familias principales, sin pretensión de listarlas todas:

Participantes. El corazón del catálogo, salido directo de la lingüística. `agente` (quien actúa con intención), `paciente` (quien recibe o sufre la acción), `tema` (la cosa o sub-situación sobre la que recae), `beneficiario` (el destinatario), `experimentador` (quien vive un estado mental) e `instrumento` (lo que media la acción). El agente apunta a Q; el tema, a O.

Circunstancias. Los ejes de posición se enchufan con roles propios: `lugar`, `origen` y `destino` apuntan a L; `momento`, `inicio` y `fin` apuntan a T; `monto`, `cantidad` y `unidad` recogen el *cuánto* en N (con la unidad apuntando a una categoría en K).

Modales y del «por qué». Roles como `estatus_factual` o `polaridad` distinguen lo real de lo planeado, lo afirmado de lo negado. Y como vimos en el capítulo 10, el «por qué» se reparte en cuatro cables (`causado_por`, `motivado_por`, `con_finalidad`, `justificado_por`, D7), todos del tipo O→O.

> **ADELANTO**
>
> ¿Y si un dominio necesita un rol que el catálogo no trae: un químico que pide `reactivo`, un banco que pide `cuenta_origen`? El sistema lo admite sin pedir permiso. El catálogo es un núcleo estable, no una camisa de fuerza; el capítulo 14 detalla esa política de extensión.

La virtud de cerrar el catálogo es la interoperabilidad: si *todos* los verbos de *todos* los dominios reparten sus participantes entre el mismo puñado de roles canónicos, entonces «el agente de una venta» y «el agente de un diagnóstico» son la misma pregunta, respondida en industrias distintas. Esa uniformidad es lo que permite que un agente de inteligencia artificial aprenda los roles una sola vez y luego entienda cualquier base de datos del mundo.

El mapeo mecánico: una oración, paso a paso

Ya tenemos las piezas. Veamos ahora la maquinaria completa en marcha, desarmando una oración fresca y cargada de detalle (del mundo del cine) con un procedimiento tan simple que raya en lo tonto:

> *"Serra filmó la escena del faro al amanecer en la costa para cerrar el segundo acto.*

LA ORACIÓN QUE VAMOS A COMPILAR

EL PROCEDIMIENTO, EN CINCO PASOS

1. **Detectar el molde.** El verbo principal es *filmó* (de *filmar*). El sistema busca su firma en el lexicon.
2. **Reificar la situación.** Crea una instancia vacía en O y le asigna un identificador: `filmacion_042`.
3. **Anclar el molde.** Registra de qué trata el evento con `instancia_de` hacia K.
4. **Enchufar los roles.** Reparte cada fragmento de la oración en la ranura que la firma dejó abierta, respetando los tipos.
5. **Descender a los subeventos.** Si un fragmento esconde otro verbo («para cerrar…»), repite el procedimiento y engánchalo.

El paso 4 es el corazón mecánico. El sistema lee la oración y asigna cada constituyente a su rol y su eje, guiándose por la firma del verbo:

REPARTO DE ROLES · PASO 4

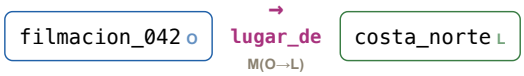
```
Serra                → quien filma      → agente         → eje Q
la escena del faro   → lo filmado       → tema           → eje O
al amanecer          → el tiempo        → momento        → eje T
en la costa          → el sitio         → lugar          → eje L
para cerrar el acto  → el objetivo      → con_finalidad  → eje O (¡otro verbo!)
```

Cada línea de ese reparto se convierte, sin intervención humana, en un hecho atómico (una tripleta tipada) colgado del evento central. La oración entera se vuelve una pila de átomos sobre un mismo sujeto:

TRIPLETAS · LA ORACIÓN COMPILADA

```
(filmacion_042, instancia_de,  accion_filmar)        ∈ M(O, K)
(filmacion_042, agente,        serra)                ∈ M(O, Q)
(filmacion_042, tema,          escena_42)            ∈ M(O, O)
(filmacion_042, momento,       2025-10-03T05:40:00)  ∈ M(O, T)
(filmacion_042, lugar_de,      costa_norte)          ∈ M(O, L)
(filmacion_042, con_finalidad, cierre_acto_001)      ∈ M(O, O)
(filmacion_042, estatus_factual, real)               ∈ M(O, K)
```

Dos de esas líneas, leídas como tripletas insignia, dejan ver el contrato en acción: el rol obliga al valor a venir de un eje concreto, y el sistema lo verifica al afirmar.



Y el paso 5 resuelve la cláusula final. «Para cerrar el segundo acto» no es una cosa: es otra situación, con su propio verbo (*cerrar*). El sistema no se inmuta, porque para él todas las situaciones son individuos iguales del eje O. Repite el procedimiento y crea un subevento:

TRIPLETAS · EL SUBEVENTO DE FINALIDAD

```
(cierre_acto_001, instancia_de, accion_cerrar)       ∈ M(O, K)
(cierre_acto_001, agente,       serra)               ∈ M(O, Q)
```

```
(cierre_acto_001, tema,         segundo_acto_marea)  ∈ M(O, O)
(cierre_acto_001, estatus_factual, intencionada)     ∈ M(O, K)
```

Una oración del lenguaje natural se ha transformado en once hechos validados, sin perder un matiz, sin un solo campo de «texto libre» y sin que ningún programador haya tenido que diseñar una tabla a la medida del cine. El verbo determinó el tipo; el sujeto se volvió agente; los complementos cayeron en sus roles canónicos; la subordinada generó una subsituación. Mecánico de principio a fin.

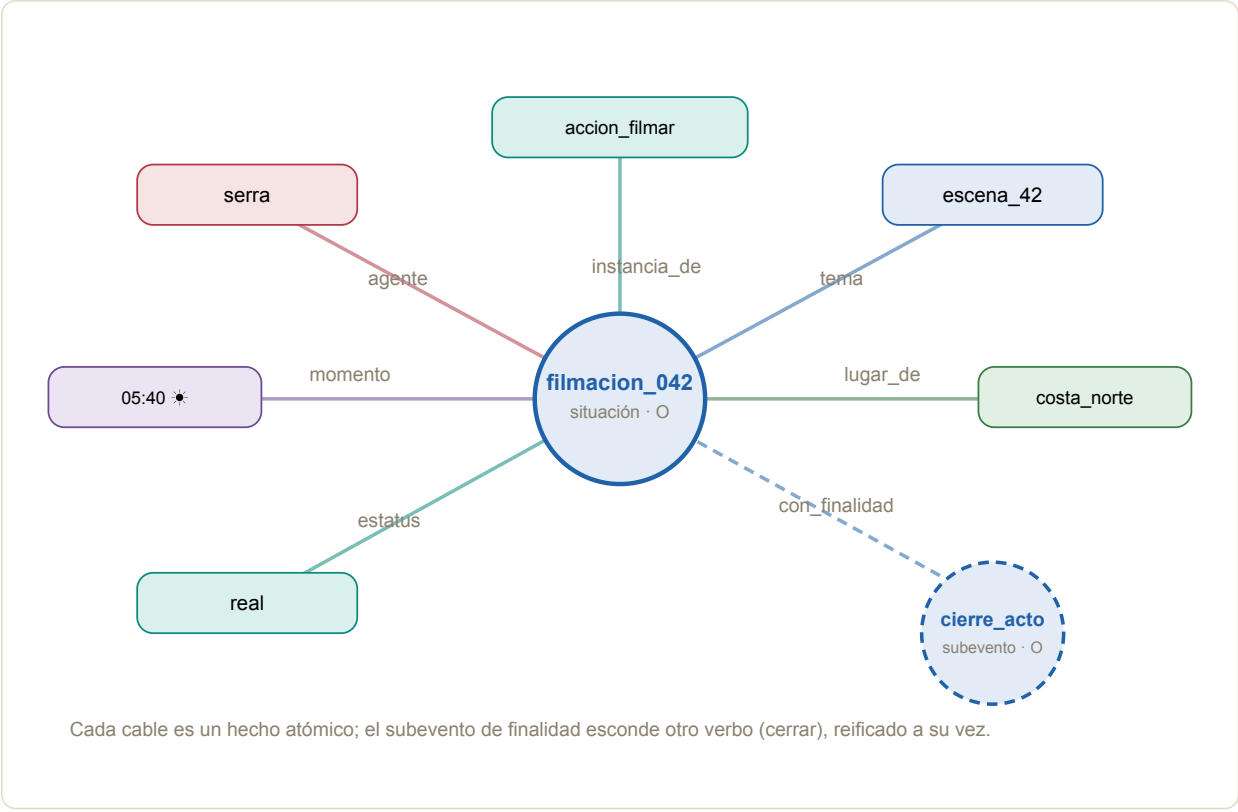


Figura 13.2. La oración «Serra filmó la escena del faro…» convertida en situación. El verbo se reifica como `filmacion_042` en O; el sujeto se vuelve *agente* en Q; cada complemento cae en su rol y su eje. La cláusula de finalidad genera un **subevento** (otra situación en O) enganchado por `con_finalidad`.

Lo notable es que el procedimiento no sabe nada de cine. Apliquemos exactamente la misma tubería a un verbo de un dominio sin parentesco, para comprobarlo. Una transmisión de radio deportiva:

TRIPLETAS · «MESSI LE PASÓ EL BALÓN A DI MARÍA EN EL MINUTO 87 CON LA ZURDA»

```
(pase_001, instancia_de, accion_pasar)          ∈ M(O, K)
(pase_001, agente,       messi)                 ∈ M(O, Q)
(pase_001, beneficiario, di_maria)              ∈ M(O, Q)
(pase_001, objeto_pase,  balon_partido_001)     ∈ M(O, O)
(pase_001, instrumento,  pierna_izquierda)      ∈ M(O, O)
(pase_001, parte_de,     partido_arg_per_2026)  ∈ M(O, O)
(pase_001, momento,      minuto_87)             ∈ M(O, T)
```

El verbo *pasar* no se parece en nada a *filmar* (pide un *beneficiario* donde el otro pedía una *finalidad*). Pero la tubería que los procesó fue idéntica: detectar el molde, reificar, anclar, enchufar. Esa es la prueba de que el motor no necesita reescribirse para cada negocio. Lo que cambia entre la clínica, el banco y el restaurante son las *firmas*, no la maquinaria que las aplica.

Las preguntas-WH son ranuras vacías

El cierre elegante de todo esto aparece cuando, en vez de *afirmar* un hecho, *preguntamos*. Las palabras interrogativas que abren este libro (*quién, qué, dónde, cuándo*) no son magia. Son, simplemente, una ranura de la firma a la que le hemos borrado el valor y dejado una incógnita. Preguntar es invocar el verbo con un hueco.

Tomemos el evento `filmacion_042` ya guardado. Cada pregunta humana se traduce a un patrón donde un rol queda abierto, y el eje de ese rol le dice al sistema dónde buscar la respuesta:

PREGUNTAS COMO PATRONES CON UN HUECO

```
¿Quién filmó la escena?    { tipo: accion_filmar, tema: escena_42, agente: ? }   → busca en Q
¿Qué filmó Serra?          { tipo: accion_filmar, agente: serra,  tema: ? }      → busca en O
¿Cuándo la filmó?          { tipo: accion_filmar, tema: escena_42, momento: ? }  → busca en T
¿Para qué la filmó?        { tipo: accion_filmar, tema: escena_42, con_finalidad: ? } → busca en O
```

> *Guardar un hecho y preguntar por él son la misma operación vista por sus dos caras: en una se completan las ranuras de la firma; en la otra, se deja una vacía y se busca con qué llenarla.*

LA SIMETRÍA ENTRE AFIRMAR Y PREGUNTAR

Esa simetría no es casual: es la consecuencia directa de tratar el verbo como un contrato de tipos. La pregunta *«¿quién?»* es legible para la máquina precisamente porque el rol `agente` declara que su valor vive en **Q**; el sistema sabe en qué caja rebuscar sin que nadie se lo explique. Las preguntas con las que un niño desarma el mundo y las consultas con las que un programa interroga una base de datos resultan ser, una vez más, la misma cosa.

El efecto matrioshka: oraciones sobre oraciones

Queda un nivel de dificultad que derriba a la mayoría de los modelos de datos: ¿qué hacemos cuando una oración habla *sobre otra oración*? Es el caso del rumor, la cita, la creencia, el reporte. Un ejemplo del cine, en plena rueda de prensa:

> *La guionista afirmó que el estudio había recortado el final.*

El verbo principal es *afirmar*, y la «cosa afirmada» no es un objeto que se pueda tocar: es **otra situación completa** (un recorte hecho por el estudio). Para nuestro modelo esto no representa ninguna dificultad nueva, porque todas las situaciones son individuos del mismo eje **O**. El *tema* de *afirmar* apunta, sin más, a otro evento:

TRIPLETAS · UNA SITUACIÓN DENTRO DE OTRA

```
(afirmacion_201, instancia_de,  accion_afirmar)        ∈ M(O, K)
(afirmacion_201, agente,        haddad)                ∈ M(O, Q)
(afirmacion_201, tema,          recorte_final_001)     ∈ M(O, O)   // ¡el tema es otro evento!

(recorte_final_001, instancia_de,   accion_recortar)   ∈ M(O, K)
(recorte_final_001, agente,         estudio_norte)     ∈ M(O, Q)
(recorte_final_001, tema,           final_pelicula)    ∈ M(O, O)
(recorte_final_001, estatus_factual, reportado)        ∈ M(O, K)   // ← clave
```

El último cable es oro puro. `estatus_factual: reportado` le dice a la base de datos que el sistema *no tiene evidencia* de que el estudio recortara el final; guarda el hecho de que *alguien lo afirmó*, sin comprometerse con su verdad. Gracias a esa marca, podemos representar oraciones cada vez más profundas («la productora negó que la guionista afirmara que el estudio recortó el final») apilando situaciones como muñecas rusas, sin programar una sola regla nueva. La recursión sale gratis, porque el eje O no distingue entre un evento simple y un evento que contiene otros.

> **IDEA CLAVE**
>
> Como todo verbo se reifica en un individuo de O, y el rol `tema` puede apuntar a cualquier individuo de O, las situaciones se contienen unas a otras sin límite. La composición del lenguaje (oraciones dentro de oraciones) se hereda sin coste: es la misma estructura recursiva, ahora en los datos.

El verbo es el contrato maestro

Cerremos fijando la idea en piedra. Cuando guardamos un verbo en nuestra arquitectura no estamos archivando una palabra de diccionario: estamos depositando un **contrato de tipos**. El verbo declara qué participantes son obligatorios, cuáles opcionales y de qué caja (Q, O, L, T…) debe venir cada dato. Si entra basura, la frena en la puerta; si entran datos limpios, ejecuta su ensamblaje en cinco pasos y los archiva sin pérdida.

Y aquí asoma la conexión más profunda del libro. Sesenta años de filosofía del lenguaje (Davidson, Parsons) descubrieron que el cerebro reifica eventos. Décadas de lingüística computacional (FrameNet, VerbNet) catalogaron que cada verbo trae sus papeles tipados. Nosotros, desde la ingeniería de datos, llegamos a la única forma de base de datos universal que no colapsa: reificar, tipar, apilar. Que tres caminos independientes desemboquen en la misma estructura no es coincidencia. Es la señal de que la forma de la lengua y la forma de los datos, cuando ambas se diseñan bien, son la misma forma.

Pero para que esta aduana funcione hace falta un documento donde estén escritos, uno por uno, los contratos de todos los verbos: sus firmas, sus roles, sus tipos, y —crucial— las mil formas en que un humano puede nombrar el mismo verbo. Ese documento maestro no es un apéndice gramatical aburrido: es el motor traductor que permite a una inteligencia artificial hablar con los discos duros de una empresa usando lenguaje natural. Se llama el **lexicon**, y es el tema del próximo capítulo.

De la firma del verbo, al compilador que la aplica.

14

El lexicon como compilador

El catálogo de tipos es exacto pero ilegible; el lenguaje de las personas es fluido pero ambiguo. Entre ambos hace falta un traductor. Ese traductor es el lexicon: la única cara que el modelo le muestra al mundo.

Un vendedor de turno escribe en la tablet del local una sola palabra para cerrar la operación: *tomar*. Lo hace cien veces al día y nunca duda. «El cliente **tomó** la camiseta visitante talla M.» La frase es transparente para cualquier hispanohablante. El problema empieza cuando esa misma palabra viaja por el resto del edificio. En la sala de reuniones del primer piso, la gerente anota que el comité «**tomó** una decisión sobre el nuevo proveedor». En la cancha del torneo que la tienda patrocina, el relator narra que el delantero «**tomó** el balón en el área». Y en el municipio de la esquina, un funcionario certifica que el alcalde le «**tomó** juramento» al nuevo regidor. Cuatro veces el mismo verbo; cuatro hechos que no tienen nada en común salvo seis letras.

> **POLISEMIA**
> Que un signo cargue varios significados emparentados no es un defecto del español: es la economía natural de cualquier lengua viva. El reto no es eliminarla, sino resolverla en el lugar correcto.

Si el sistema del capítulo anterior tomara el verbo al pie de la letra, haría lo que hace un parser ingenuo: leería *tomar*, dispararía una única firma (digamos `tomar(agente, tema)`) y guardaría las cuatro frases como instancias del mismo tipo de evento. El resultado sería una base de datos que cree que entregar una camiseta, deliberar en comité, recibir un balón y administrar un juramento son la misma clase de cosa. Las consultas posteriores heredarían el desastre: preguntar «¿qué decisiones se tomaron este mes?» devolvería también las ventas y las jugadas. El error no se nota al escribir; se cobra al leer.

El capítulo 13 nos dejó con el verbo entendido como **signatura**: cada verbo declara qué roles exige y de qué eje sale cada uno. Pero acabamos de ver que un verbo no basta como llave. *Tomar* no es *una* signatura: es una *familia* de signaturas que comparten ortografía. Para elegir la correcta hace falta mirar también qué acompaña al verbo. Y ese trabajo (el de pasar del lenguaje a la firma exacta) no lo hace el verbo solo. Lo hace una pieza dedicada, que es el verdadero protagonista de este capítulo.

Dos lenguajes que no deberían tocarse

Conviene nombrar con precisión los dos mundos que el capítulo 13 puso frente a frente. Por un lado está el **catálogo canónico**: el conjunto cerrado y versionado de roles (`agente`, `tema`, `beneficiario`, `experimentador`, `causado_por`…) con sus firmas, sus ejes de origen y sus reglas de obligatoriedad. Es exacto, estable y completamente ilegible para quien no diseñó el modelo. Ningún recepcionista, ningún médico, ningún relator deportivo va a escribir `experimentador` para registrar que a un cliente le encantó el postre.

Por el otro lado está el **lenguaje del usuario**: «vendedor», «comprador», «paciente», «socio», «el que pierde». Vocabulario vivo, distinto en cada gremio, lleno de sinónimos y de polisemia. Es lo único que la gente sabe escribir, y es lo único que un modelo de lenguaje recibe cuando lee una frase suelta. Si forzáramos a estos dos mundos a tocarse (si el catálogo canónico fuera la interfaz) habríamos construido una pieza de ingeniería impecable y socialmente inútil. La adopción se moriría en la primera capacitación.

La solución es interponer una capa de traducción entre ambos. Una capa que por fuera hable el idioma del usuario y por dentro emita tipos canónicos. A esa capa la llamamos **lexicon**, y su diseño descansa sobre dos decisiones que conviene enunciar de frente.

D8 **EL CATÁLOGO ES INVISIBLE; EL LEXICON ES LA INTERFAZ**

El catálogo canónico de roles existe, es exacto y es la base sobre la que se valida cada hecho —pero **nunca se expone directamente**. La única superficie con la que el mundo exterior interactúa es el **lexicon**. Por dentro, los ingenieros pueden renombrar un rol, fusionar dos o versionar el catálogo entero; por fuera, nada cambia, porque la cara visible del sistema es el diccionario, no el catálogo.

Si D8 fija *qué pieza es la interfaz*, su consecuencia natural es una promesa hacia el usuario final, que merece su propio número porque otras partes del libro la invocan:

D9 **EL USUARIO NUNCA TOCA ETIQUETAS CANÓNICAS**

Un humano **jamás** debe verse obligado a escribir `agente`, `beneficiario` o `experimentador` para que el sistema funcione. El usuario emplea su propio vocabulario («vendedor», «comprador», «cliente», «el que anota») y el lexicon lo traduce en silencio a la etiqueta canónica correspondiente. La complejidad del idioma se resuelve en el diccionario, no en la cabeza de quien escribe.

ENCAPSULAMIENTO

D8 es, en el fondo, el viejo principio de ocultar la implementación detrás de una interfaz estable. Lo que cambia es *qué* se encapsula: aquí no es una clase de código, sino el vocabulario entero de un dominio.

El par D8–D9 reparte responsabilidades con nitidez. D8 mira hacia adentro: garantiza a los ingenieros que pueden mover los engranajes (agregar roles, cambiar códigos internos, reorganizar el catálogo) sin romperle nada a nadie, porque la fachada permanece. D9 mira hacia afuera: garantiza a las personas que pueden seguir hablando como hablan. El lexicon es el contrato que sostiene ambas promesas a la vez: **estable hacia afuera, flexible hacia adentro**.

> *El catálogo canónico es el corazón del sistema; el lexicon es su rostro. Y al mundo solo se le muestra el rostro.*
>
> EL CONTRATO DE D8

El lexicon en capas

Vale la pena ver la arquitectura completa de un vistazo antes de bajar al detalle de una entrada. La frase del usuario entra por arriba con su vocabulario natural; el lexicon la desambigua y la traduce a roles canónicos; el almacenamiento, ya en el fondo, solo trabaja con identificadores internos. Tres capas, y cada una puede cambiar sin perturbar a las otras.

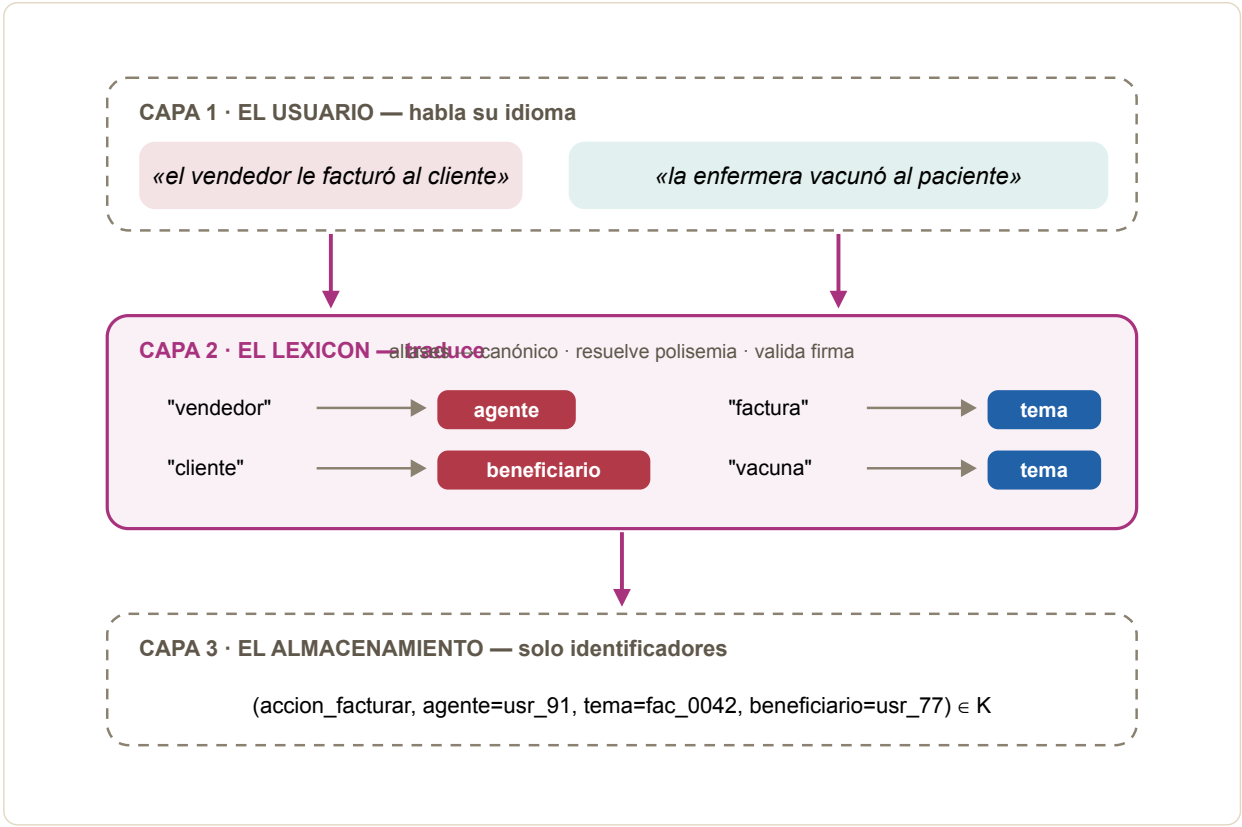


Figura 14.1. Las tres capas del lexicon. Arriba, el usuario escribe con el vocabulario de su gremio. En el centro, el lexicon traduce cada alias al rol canónico (en magenta, el eje **M**: la traducción *es* trabajo de predicados) y valida la firma. Abajo, el almacenamiento solo ve tipos e identificadores internos. Cualquiera de las tres capas puede cambiar sin tocar a las otras: eso es D8 hecho geometría.

Autopsia de una entrada

Bajemos al detalle. Una entrada del lexicon es una «página» del diccionario, y cada pieza tiene un trabajo concreto. Tomemos un verbo del mundo de la tienda (el local de la escena inicial despacha camisetas al público) y desarmemos su entrada de `vender`:

LEXICON

```
{
  "verbo": "vender",
  "tipo_situacion": "accion_vender",
  "roles": {
    "agente":       { "canonico": "agente",       "aliases": ["vendedor", "el que vende", "el que
despacha"] },
    "tema":         { "canonico": "tema",         "aliases": ["producto", "lo vendido", "camiseta"]
},
    "beneficiario": { "canonico": "beneficiario", "aliases": ["comprador", "cliente", "el que
compra"] },
    "por_cuanto":   { "canonico": "por_cuanto",   "aliases": ["precio", "monto", "importe"] }
  },
  "obligatorios": ["agente", "tema", "beneficiario", "por_cuanto"],
  "opcionales":   ["momento", "lugar_de", "moneda", "instrumento"],
  "ejemplo": "El vendedor le vendió una camiseta al cliente por 49.90 dólares"
}
```

Son seis piezas, y conviene leerlas como lo que son: la declaración de una función con su manual de uso adjunto.

1. `verbo`. La llave que activa la entrada. Puede ser una palabra suelta (`vender`) o un patrón compuesto (`tomar [decisión]`), como veremos al resolver la polisemia.

2. `tipo_situacion`. El código interno exacto con el que el evento se ancla en la caja **K**. Es el motor oculto; el usuario nunca lo ve ni lo escribe.
3. `roles` **y sus** `aliases`. Aquí vive D9. El rol canónico es `agente`, pero la lista de aliases le enseña al sistema que «vendedor», «el que vende» y «el que despacha» significan lo mismo. Gracias a eso, «el vendedor le vendió una camiseta al cliente» y «el agente transfirió un producto al beneficiario» producen el *idéntico* registro.
4. `obligatorios` **y** `opcionales`. Las reglas de seguridad de la firma. Dictan qué roles no pueden faltar para que el hecho se acepte. Una venta sin precio se rechaza; una venta sin lugar, no.
5. `ejemplo`. Una frase humana real. No es adorno: el motor de pruebas (y, cada vez más, el modelo de lenguaje) lee este ejemplo para aprender cómo se usa el verbo en la práctica.

Visto así, el lexicon deja de parecerse a un glosario y empieza a parecerse a otra cosa: a un **compilador**. Un compilador toma código fuente legible para humanos y lo traduce a instrucciones exactas que la máquina ejecuta, verificando de paso que los tipos cuadren. El lexicon hace exactamente eso con el lenguaje: toma una frase, la traduce a una tripleta tipada y comprueba que la firma se cumpla. El «código fuente» es el español; el «binario» es el hecho atómico del capítulo 7.

El idioma de los modelos: function calling

Aquí ocurre una de esas convergencias que parecen casualidad y no lo son. Los modelos de lenguaje de uso corporativo (entre ellos Claude, de Anthropic) se conectan al software de la empresa mediante un mecanismo estándar: el *function calling*, o uso de herramientas. El patrón es simple. La aplicación le entrega al modelo un catálogo en formato JSON que describe cada herramienta disponible, sus parámetros y cuáles son obligatorios. Cuando un usuario pide algo en lenguaje natural, el modelo elige la herramienta adecuada y emite una llamada estructurada, con cada parámetro en su sitio.

> **CONVERGENCIA**
> Que la lingüística formal de los años noventa y el *function calling* de los modelos actuales hayan llegado a la misma forma (nombre, descripción, parámetros tipados, obligatorios) no es coincidencia: ambos resuelven el mismo problema, mapear lenguaje a acción verificable.

Lo notable es que ese formato JSON es **estructuralmente idéntico** a una entrada del lexicon. Tomemos la misma entrada de `vender` y reescribámosla en el esquema que un modelo espera recibir:

TOOL SCHEMA (FUNCTION CALLING)

```
{
  "name": "accion_vender",
  "description": "Registrar una venta: transferencia de un bien con compensación monetaria",
  "input_schema": {
    "type": "object",
    "properties": {
      "agente":       { "type": "string", "eje": "Q", "description": "vendedor, el que vende, el que
despacha" },
      "tema":         { "type": "string", "eje": "O", "description": "producto, camiseta, lo
vendido" },
      "beneficiario": { "type": "string", "eje": "Q", "description": "comprador, cliente, el que
compra" },
      "por_cuanto":   { "type": "number", "eje": "N", "description": "precio, monto, importe" },
      "momento":      { "type": "string", "eje": "T", "description": "cuándo ocurrió" },
      "lugar_de":     { "type": "string", "eje": "L", "description": "dónde ocurrió" }
    },
    "required": ["agente", "tema", "beneficiario", "por_cuanto"]
  }
}
```

Las correspondencias son uno a uno, y vale la pena nombrarlas para que no quede como una impresión vaga:

El `tipo_situacion` del lexicon se vuelve el `name` de la herramienta. Los `aliases` de cada rol se vuelven la `description`, que

es justamente la pista que el modelo usa para saber qué significa el parámetro. El eje de origen (Q, O, N…) determina el `type` y aporta una validación semántica extra.

Y la lista de `obligatorios` es, literalmente, el campo `required` del esquema. No hay que traducir nada a mano ni mantener dos artefactos en paralelo: el lexicon *es* el catálogo de herramientas. Exponerlo a un modelo es serializarlo a JSON y entregarlo. Lo que en otras arquitecturas es un proyecto de integración, aquí es una función de exportación.

El efecto práctico es que una frase suelta se convierte en un hecho estructurado sin que nadie escriba un parser. El *prompt* que recibe el modelo es casi todo catálogo:

PROMPT

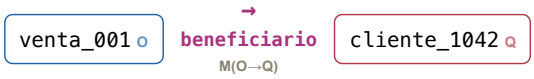
```
SISTEMA. Eres un extractor de hechos WQuestions. Dada una frase en español,
devuelve SOLO un JSON con la situación y sus roles canónicos.

Catálogo (verbo → situación · roles obligatorios):
  vender → accion_vender · [agente, tema, beneficiario, por_cuanto]
  tomar  → accion_consumir · [agente, tema]

Frase del usuario: "El vendedor le vendió una camiseta al cliente por 49.90 dólares"
```

Y la respuesta llega lista para asentarse en el grafo, ya traducida a roles canónicos:

RESPUESTA DEL MODELO

```
{
  "tipo_situacion": "accion_vender",
  "agente": "vendedor_17",
  "tema": "camiseta_88",
  "beneficiario": "cliente_1042",
  "por_cuanto": 49.90
}
```

Esa salida es exactamente el conjunto de tripletas tipadas del capítulo 7. Cada par rol–valor se vuelve un cable; por ejemplo, el `beneficiario` se lee como una tripleta del eje M que sale del evento y aterriza en una persona:



Y una vez asentada, la situación completa se descompone en los cables que ya conocemos:

TRIPLETAS

```
(venta_001, instancia_de,  accion_vender)   ∈ M(O, K)
(venta_001, agente,        vendedor_17)     ∈ M(O, Q)
(venta_001, tema,          camiseta_88)     ∈ M(O, O)
(venta_001, beneficiario,  cliente_1042)    ∈ M(O, Q)
(venta_001, por_cuanto,    49.90)           ∈ M(O, N)
```

El modelo nunca vio la palabra `por_cuanto` hasta que el lexicon se la nombró; el usuario nunca la verá en absoluto. Esa es la promesa de D9 cumplida de extremo a extremo: el español entra por arriba, la tripleta canónica sale por abajo, y la etiqueta interna jamás se le exige a una persona.

Y esa promesa no caduca al escribir el hecho: vale también al preguntar por muchos. El gerente pide «el total vendido por cada vendedor este mes» con sus palabras, y el lexicon resuelve «vendedor» contra `agente`, «total» contra `por_cuanto`, sin que él vea

jamás una etiqueta canónica. El reporte recorre miles de ventas, pero se pide y se devuelve en el idioma de la tienda, no en el del catálogo.

Resolver la polisemia: tomar, otra vez

Volvamos a la escena del principio. Tenemos cuatro frases que comparten el verbo *tomar* y no comparten nada más. ¿Cómo evita el lexicon confundirlas? La respuesta es decepcionantemente directa, y por eso funciona: **una entrada distinta para cada significado**. El lexicon no busca una regla mágica que cubra todos los usos de *tomar*; declara una lista de patrones, cada uno apuntando a su propio tipo de situación, y deja que el complemento decida.

LEXICON · POLISEMIA DE «TOMAR»

```
{
  "tomar [decisión | acuerdo | resolución]": {
    "tipo_situacion": "accion_decidir",
    "obligatorios": ["agente", "tema"],
    "ejemplo": "El comité tomó una decisión sobre el nuevo proveedor"
  },
  "tomar [el balón | la pelota | el pase]": {
    "tipo_situacion": "accion_recuperar_posesion",
    "obligatorios": ["agente", "tema"],
    "ejemplo": "El delantero tomó el balón en el área"
  },
  "tomar [juramento]": {
    "tipo_situacion": "acto_juramentacion",
    "obligatorios": ["agente", "paciente"],
    "ejemplo": "El alcalde le tomó juramento al nuevo regidor"
  },
  "tomar": {
    "tipo_situacion": "accion_consumir",
    "obligatorios": ["agente", "tema"],
    "ejemplo": "El cliente tomó un refresco en la cafetería del estadio"
  }
}
```

El procedimiento de resolución es, otra vez, el de un compilador: el sistema intenta encajar la frase **desde el patrón más específico hacia el más general**. Si aparece «tomar el balón», gana la entrada de fútbol. Si aparece «tomar juramento», gana el acto de juramentación. Y solo cuando ningún patrón con complemento coincide («tomar un refresco») cae la regla genérica de consumir. El orden importa: lo específico siempre primero, lo genérico como red de seguridad. La figura siguiente lo muestra con los colores de cada eje para que se vea adónde va a parar cada hecho.

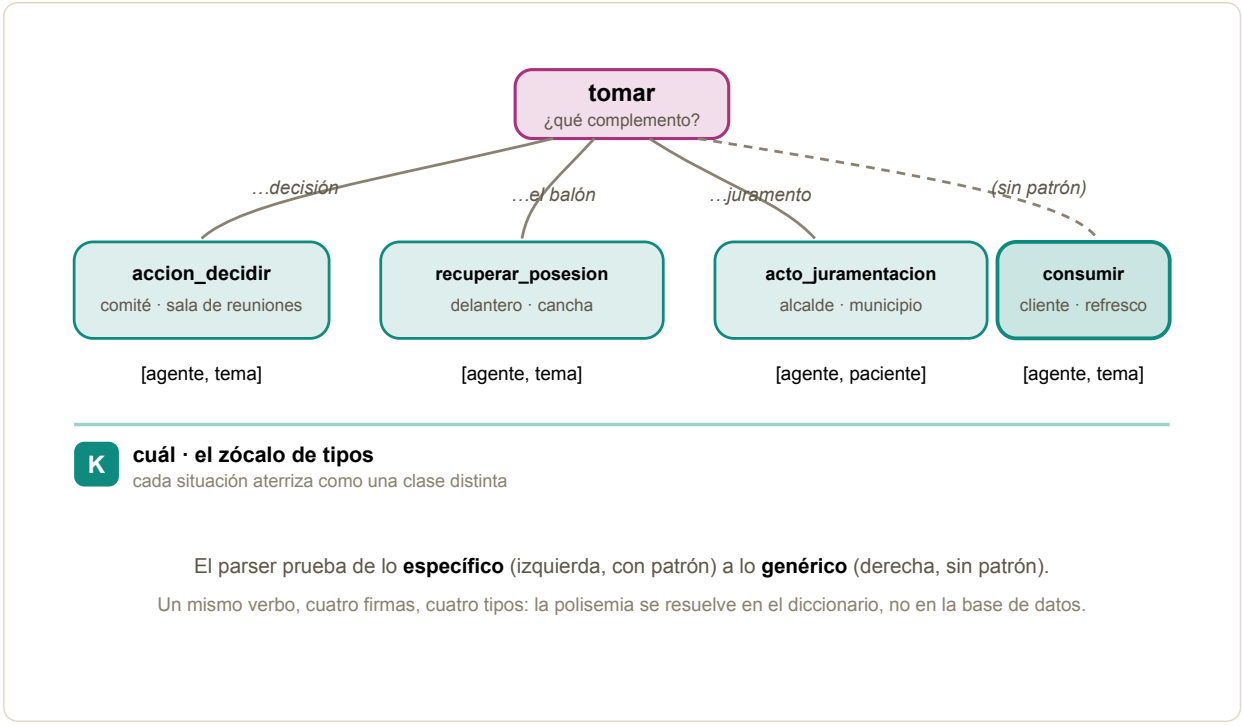


Figura 14.2. Resolución de la polisemia de *tomar*. El complemento (decisión, balón, juramento, o ninguno) desambigua entre cuatro unidades léxicas que conducen a cuatro tipos de situación distintos en el eje K, cada uno con su propia firma de roles. El patrón genérico (a trazos) actúa solo como red de seguridad.

La consecuencia arquitectónica es la que da título al capítulo. La complejidad del lenguaje no se elimina (sería imposible) sino que se **traslada al lugar correcto**: al diccionario. Un modelador de datos puede registrar un nuevo significado de *tomar* («tomar un préstamo», por ejemplo) agregando una entrada de texto, sin tocar una sola línea del núcleo. La base de datos sigue siendo ignorante sobre el español; el lexicon hace todo el trabajo lingüístico.

Dos clases de alias: por rol y por dominio

Hasta aquí los aliases han sido **por rol**: dentro de la entrada de `vender`, «vendedor» equivale a `agente`. Pero hay un segundo nivel de traducción, indispensable para que el modelo funcione de verdad en una empresa. Hay palabras que aplican *transversalmente* a todo un dominio, sin importar qué verbo se use. Una tienda siempre habla de «vendedor», «boleta» y «caja»; una clínica siempre habla de «paciente», «historia» y «diagnóstico». Repetir esos aliases en cada entrada sería absurdo. Para eso el lexicon admite **dialectos de dominio**: un pequeño archivo de traducción que se carga encima del catálogo base.

DIALECTO DE DOMINIO

```
{
  "dominio": "tienda_central",
  "aliases_de_dominio": {
    "vendedor":      "agente",
    "boleta":        "comprobante",
    "caja":          "lugar_de",
    "catálogo":      "catalogo",
    "despachar":     "verbo_vender",
    "talla":         "atributo_medida"
  }
}
```

Con este dialecto cargado, un vendedor puede escribir «despaché una camiseta talla M en la caja dos» y el sistema traduce «despaché» al verbo `vender`, «caja» al rol `lugar_de` y «talla» al atributo de medida: todo sin que ninguna entrada de verbo

mencione la palabra «caja». El núcleo permanece intacto, pero la tienda, la clínica y el municipio sienten, cada uno, que el software fue hecho a su medida. Es el mismo motor universal hablando tres jergas distintas.

CAPAS DE ALIAS

Los aliases por rol son *locales* a un verbo; los de dominio son *globales* a un gremio. El lexicon consulta primero el dialecto cargado y luego la entrada del verbo: lo específico del dominio puede afinar lo genérico del catálogo.

A hombros de gigantes

Sería ingenuo —y arrogante— presentar el lexicon como una invención sin ancestros. La lingüística computacional lleva tres décadas construyendo catálogos inmensos con esta misma lógica de verbos, roles y firmas. Cualquier implementación seria del lexicon debería alimentarse de ellos en lugar de empezar de cero.

PRECEDENTE · FRAMENET, VERBNET Y PROPBANK

FrameNet[14] (Universidad de Berkeley, años noventa) clasifica el idioma en *escenarios conceptuales* o *frames*. Su escenario de «Comercio» define roles predefinidos como *Buyer*, *Seller* y *Goods*: casi calcados de nuestra entrada de `vender`. Sus más de mil doscientos escenarios son una mina para poblar la caja **K** con tipos de situación ya pensados por lingüistas.

VerbNet[15] (Universidad de Pensilvania) agrupa miles de verbos del inglés que comparten una misma estructura lógica (verbos de transferencia, verbos de encuentro, verbos de cambio de estado) de modo que un verbo nuevo hereda la firma de su clase en lugar de definirse a mano.

PropBank prioriza la cobertura: etiqueta masivamente textos reales con roles genéricos (`Arg0` a `Arg5`). Es menos elegante, pero su escala lo convierte en la base preferida para *entrenar* modelos modernos a extraer estructura argumental.

WQuestions no compite con estas obras: actúa como una **capa de agregación** por encima de ellas. Se pueden extraer las signaturas de FrameNet, alinear las clases de VerbNet y usar los datos de PropBank para entrenar el extractor. El camino, en buena medida, ya está pavimentado; lo que aporta el lexicon es atarlo todo a un catálogo único de roles tipados por eje, y exponerlo con una sola interfaz.

Un vistazo al catálogo en acción

Para que esto no quede en abstracto, vale la pena ver varias entradas juntas, de dominios sin relación entre sí. Mientras las recorres, fíjate en tres cosas: que los roles se *reciclan*, que los aliases cambian con el gremio aunque el esqueleto no, y que los casos «raros» (verbos sin agente, verbos invertidos) encajan sin forzar nada.

LEXICON · CUATRO DOMINIOS

```
{
  "vacunar": {
    "tipo_situacion": "accion_vacunar",
    "roles": {
      "agente":      ["enfermera", "personal sanitario"],
      "paciente":    ["paciente", "vacunado"],
      "tema":        ["vacuna", "antígeno"]
    },
    "obligatorios": ["agente", "paciente", "tema"],
    "ejemplo": "La enfermera vacunó a 47 niños con la triple viral el sábado"
  },

  "multar": {
```

```
    "tipo_situacion": "accion_multar",
    "roles": {
      "agente":          ["autoridad", "ente regulador"],
      "paciente":        ["multado", "infractor"],
      "monto":           ["importe de la multa"],
      "justificado_por": ["norma aplicada", "artículo"]
    },
    "obligatorios": ["agente", "paciente", "monto", "justificado_por"],
    "ejemplo": "La municipalidad multó al local por ocupar la vereda"
  },

  "invocar": {
    "tipo_situacion": "accion_invocar_funcion",
    "roles": {
      "agente":      ["llamador", "cliente API", "el modelo"],
      "tema":        ["función invocada", "herramienta"]
    },
    "obligatorios": ["agente", "tema"],
    "ejemplo": "El modelo invocó la función accion_vender con los parámetros extraídos"
  },

  "llover": {
    "tipo_situacion": "evento_meteorologico",
    "roles": {
      "lugar_de":    ["región", "ciudad"],
      "intensidad":  ["leve", "moderada", "torrencial"]
    },
    "obligatorios": [],
    "notas": "Verbo impersonal: no exige agente ni ningún rol obligatorio.",
    "ejemplo": "Llovió torrencialmente sobre la ciudad durante seis horas"
  }
}
```

El catálogo entero, mirado en perspectiva, revela tres cosas que valen oro para quien implementa.

1 LOS MISMOS CABLES, SIEMPRE

Vacunar, multar, invocar y vender comparten un puñado de roles (`agente`, `tema`, `paciente`, `lugar_de`). El catálogo base de roles basta para dominios que no se parecen en nada.

2 LOS ALIASES SON EL PODER

Lo que distingue salud de banca no son los códigos internos, sino las palabras naturales. «Enfermera» y «autoridad» mapean ambos a `agente`; «vacuna» y «factura», ambos a `tema`. La base de datos ignora el negocio.

3 LO RARO TAMBIÉN ENCAJA

`llover` no tiene agente y se modela sin forzar nada: la lista de obligatorios queda vacía. Un agente de IA como `invocar` ocupa el mismo rol `agente` que un humano, tal como anticipó la regla de agencia contextual.

Un sistema empresarial maduro puede tener un lexicon de varios miles de entradas. Las que acabas de ver son apenas el cimiento, pero ya muestran el patrón: el conocimiento del dominio vive en texto declarativo, no en código, y crece agregando entradas, no reescribiendo el motor.

Qué gana quien lo implementa

Conviene cerrar nombrando con claridad la diferencia entre tener un lexicon y no tenerlo.

EL LEXICON EN PRODUCCIÓN

Habla el idioma real de tu gente. Sin lexicon, habría que enseñarle a cada empleado los campos robóticos del sistema. Con él, un vendedor escribe «el cliente tomó la camiseta visitante» y el hecho queda registrado en la jerga de la tienda, sin capacitación.

Es la puerta de entrada de la IA. El lexicon ya tiene la forma de un catálogo de *function calling*. Un modelo conectado a él empieza a redactar y auditar hechos sin que nadie reprograme el sistema: basta serializarlo y entregarlo.

Desambigua antes de escribir en disco. La polisemia (el caos de *tomar*) se resuelve en el diccionario. La base de datos recibe siempre hechos limpios, ya etiquetados con su tipo exacto en **K**.

Escala agregando texto. Abrir un dominio nuevo (logística, por decir) no exige crear cuarenta tablas: exige añadir un bloque de verbos al lexicon. El motor central hereda ese conocimiento de inmediato.

La genialidad de los siete ejes es el corazón del modelo, sí. Pero su **adopción** (el que una clínica, un banco o una tienda lo usen sin sentir que cambiaron de idioma) depende entera y exclusivamente de este traductor. El catálogo es el motor; el lexicon es el volante. Y nadie conduce agarrando el motor.

> *Un compilador no le pide al programador que escriba en binario. El lexicon no le pide al usuario que escriba en canónico. Esa es toda la diferencia entre un modelo elegante y un modelo que la gente usa.*

LA TESIS DEL CAPÍTULO

Nos queda un frente abierto. El lexicon resuelve la polisemia y la jerga, pero el lenguaje guarda trampas más finas: la negación, la modalidad, el tiempo verbal que no coincide con el tiempo del hecho, las metáforas que ningún diccionario anticipa. Es hora de poner el modelo completo (ejes, hecho atómico, lexicon) a prueba contra los casos que de verdad duelen. Eso es el próximo capítulo.

15

El modelo bajo presión

El verbo y el lexicon ya tienden el puente entre lo que decimos y lo que el grafo guarda. Ahora toca cruzarlo cargando las tres frases que suelen hundir cualquier puente.

Lee estas tres oraciones en voz alta y fíjate en lo poco que te cuesta entenderlas: *«El estreno del documental obligó a un cambio de cartelera»*; *«El estudio debía estrenar en marzo, pero no podía pagar la sala todavía»*; *«La distribuidora le tomó el pelo al cineasta novato con esa fecha»*. No has tropezado ni una vez. Y, sin embargo, las tres harían humear a un analizador ingenuo, de esos que buscan «el verbo de acción» y reparten roles alrededor. Este capítulo trata de por qué tropieza la máquina donde tú no, y de las tres convenciones con que WQuestions absorbe el golpe sin inventar un solo eje nuevo.

> **DÓNDE ESTAMOS**
> En el capítulo 13 vimos que el verbo es la signatura que reparte los roles; en el capítulo 14, que el lexicon compila el habla humana hacia el catálogo canónico. Aquí lo sometemos a las tres pruebas que más lo aprietan.

Conviene desmontar la ilusión de inmediato, porque es la raíz de los tres problemas. Un parser ingenuo cree que cada oración esconde *un* verbo de acción rodeado de participantes, y que su tarea es encontrarlo y colgar a cada quien en su clavija. Esa heurística funciona con *«la directora rodó la escena»* y se desmorona con casi todo lo demás. La primera frase esconde sus acciones bajo dos sustantivos (*estreno*, *cambio*); la segunda apila dos verbos de intención (*debía*, *podía*) sobre los hechos reales; la tercera dice *tomar el pelo* y no habla de cabello en absoluto. Tres trampas distintas, un mismo origen: el lenguaje no marca en la superficie lo que el modelo necesita por debajo.

> *El modelo no entiende español. Lo compila. Y compilar bien empieza por reconocer los tres disfraces que el español usa para esconder un hecho.*

LA TESIS DEL CAPÍTULO

Adelanto la conclusión para que leas con red: ninguna de las tres trampas obliga a ampliar la geometría de siete ejes. Las tres se resuelven en la capa de traducción (el lexicon de la decisión D9, donde el usuario nunca toca una etiqueta canónica) y todas terminan en la misma forma de siempre: tripletas tipadas `(sujeto, cable, objeto)`, el átomo de la decisión D3. El límite real del modelo, lo veremos al cerrar, está en otra parte.

Primera presión: la nominalización

El español tiene un talento casi tramposo para convertir acciones en cosas. *Estrenar* se vuelve *el estreno*; *rodar*, *el rodaje*; *cancelar*, *la cancelación*. A esa mutación los lingüistas la llaman **nominalización**: un verbo se disfraza de sustantivo y se cuela en la oración como si fuera un objeto más. Para ti, decir *«el documental se estrenó»* o *«el estreno del documental»* es lo mismo. Para el modelo, también debe serlo.

NOMINALIZACIÓN

Recurso por el cual un verbo adopta forma de sustantivo (*estrenar* → *el estreno*) y puede funcionar como sujeto u objeto de otra oración, ocultando que en el fondo nombra un *evento*, no una cosa.

La regla de arquitectura es de una sola línea, y vale la pena grabársela: **una nominalización no es un evento nuevo; es la misma situación reificada con otro empaque gramatical**. *El estreno* no pide un eje nuevo ni un tipo de dato exótico; pide solo que el lexicon sea lo bastante despierto para saber que *estreno* es un alias que dispara la misma entrada que *estrenar*. La reificación de eventos como situaciones en el eje O ya la justificamos en el capítulo 9 (la decisión D4); aquí no inventamos nada, solo le abrimos una segunda puerta de entrada al mismo evento.

Desarmemos la primera oración problemática:

> *El estreno del documental obligó a un cambio de cartelera.*

TRIPLETAS

```
(estreno_204,  instancia_de,    accion_estrenar)     ∈ M(O, K)
(estreno_204,  tema,            documental_orillas)  ∈ M(O, O)
(estreno_204,  estatus_factual, real)                ∈ M(O, K)

(cambio_071,   instancia_de,    accion_modificar)    ∈ M(O, K)
(cambio_071,   tema,            cartelera_sala_norte) ∈ M(O, O)
(cambio_071,   estatus_factual, real)                ∈ M(O, K)

(cambio_071,   motivado_por,    estreno_204)         ∈ M(O, O)   ← el conector del eje M
```

Mira lo que acaba de ocurrir. Dos sustantivos (*estreno* y *cambio*) se convirtieron en dos situaciones de pleno derecho en el eje O. Y el verbo de la superficie, *obligó a*, que un parser ingenuo habría elevado a evento propio, nosotros lo degradamos a lo que de verdad es: el cable `motivado_por` del capítulo 10, el enlace que ata un hecho a su razón. La acción aparente resultó ser pegamento causal entre dos eventos reificados. El resultado es un puñado de átomos limpios, sin un solo nodo fantasma.

MATIZ

¿`causado_por` o `motivado_por`? El capítulo 10 reparte el «por qué» en cuatro cables. Un estreno no *causa* físicamente un cambio de cartelera como una chispa causa fuego: lo *motiva*. Elegir bien el cable es trabajo del lexicon, no de un eje extra.

¿Cómo logra el sistema esto por sí solo? No con magia: con una entrada de lexicon que declara, junto al verbo, sus formas nominales. El compilador del capítulo 14 ya hace el resto.

LEXICON · JSON

```
{
  "verbo": "estrenar",
  "tipo_situacion": "accion_estrenar",
  "formas_nominales": ["estreno", "premier", "lanzamiento"],
  "roles": {
    "agente": "M(O,Q)",
    "tema":   "M(O,O)"
  },
  "obligatorios": ["tema"]
}
```

El campo `formas_nominales` es todo el truco. Con esa línea, al modelo le da exactamente igual que un periodista escriba *«el estudio estrenó el documental»* o *«el estreno del documental»*: la maquinaria extrae los mismos átomos y descarta la forma de la frase como descarta el espacio en blanco. Lo mismo aplica en cualquier dominio: da igual *«el banco aprobó el crédito»* que *«la*

aprobación del crédito»; da igual *«el agente consultó el grafo»* que *«la consulta al grafo»*. La poesía del redactor se evapora; el hecho permanece.

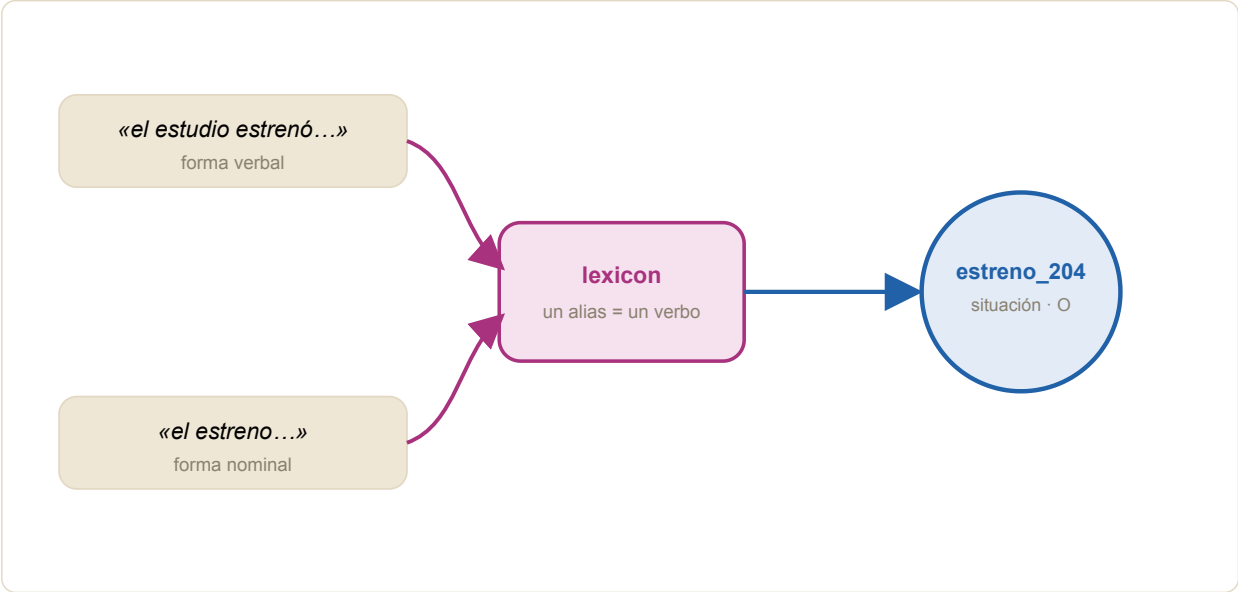


Figura 15.1. El verbo *estrenar* y el sustantivo *estreno* entran por puertas distintas, pero el lexicon los reconoce como el mismo disparador y ambos convergen en una única situación reificada, `estreno_204`, en el eje O. La nominalización no duplica el evento: solo le abre una segunda entrada.

Segunda presión: los modales

Los verbos modales (*querer, deber, poder, soler, parecer*) son una emboscada para cualquier base de datos. Tienen aspecto de verbo principal, conjugan como tal, pero en realidad son **modificadores** de un segundo verbo. Cuando un estudio dice *«queremos estrenar en marzo»*, no ocurren dos cosas en el mundo (un *querer* por un lado y un *estrenar* por otro). Ocurre *una* sola situación posible, *estrenar*, envuelta en una etiqueta de intención que avisa: esto todavía no pasó.

LA TRAMPA DEL NODO FANTASMA

Si el modelo reificara cada *querer*, *deber* y *poder* como un evento autónomo en el eje O, el grafo se llenaría de situaciones que no ocurren en ninguna parte del mundo. Multiplicaríamos los nodos por cuatro y, peor aún, mezclaríamos intenciones con hechos: el sistema ya no sabría qué *pasó* y qué solo se *quería*.

La salida es tratar a los modales como lo que son: **decoradores**. No crean una situación; envuelven una situación existente con propiedades que ajustan su modo y su factualidad. El núcleo (*estrenar*, *pagar*) sigue siendo un solo evento en O; el modal aporta dos cables de matiz: la `modalidad` (¿es un deseo, una obligación, una capacidad?) y el `estatus_factual` (¿ocurrió, se intenta, es imposible?). Desarmemos la segunda oración:

> *El estudio debía estrenar en marzo, pero no podía pagar la sala todavía.*

TRIPLETAS

```
(estrenar_309, instancia_de,    accion_estrenar)     ∈ M(O, K)
(estrenar_309, agente,          estudio_lumbre)      ∈ M(O, Q)
(estrenar_309, momento,         2026-03)             ∈ M(O, T)
(estrenar_309, modalidad,       deontica)            ∈ M(O, K)   ← un deber, no un hecho
(estrenar_309, estatus_factual, obligado)            ∈ M(O, K)

(pagar_309,    instancia_de,    accion_pagar)        ∈ M(O, K)
(pagar_309,    agente,          estudio_lumbre)      ∈ M(O, Q)
```

```
(pagar_309,    tema,           alquiler_sala_norte) ∈ M(0, O)
(pagar_309,    modalidad,      aletica)             ∈ M(0, K)   ← una capacidad
(pagar_309,    polaridad,      negativa)            ∈ M(0, K)   ← que no se tiene
(pagar_309,    estatus_factual, no_factible)        ∈ M(0, K)

(pagar_309,    contrasta_con,  estrenar_309)        ∈ M(0, O)   ← el conector del «pero»
```

Generamos **dos** situaciones, no cuatro: `estrenar_309` y `pagar_309`. El *debía* se guardó como `modalidad: deontica` (el modo de la obligación); el *no podía*, como `modalidad: aletica` (el modo de lo posible) teñido de `polaridad: negativa`. Y el adversativo *pero*, que un parser tomaría por ruido, se volvió un cable honesto, `contrasta_con`, que ata las dos situaciones. Ahorramos nodos, evitamos los fantasmas y, sobre todo, sostuvimos la regla de oro: **una situación en el mundo equivale a una situación en el sistema**.

VOCABULARIO

Deóntica: lo que debe ser (obligación, permiso). *Alética*: lo que puede ser (posibilidad, capacidad). *Volitiva*: lo que se desea. Tres sabores de modalidad que la lógica modal distingue desde hace décadas; el modelo solo los etiqueta.

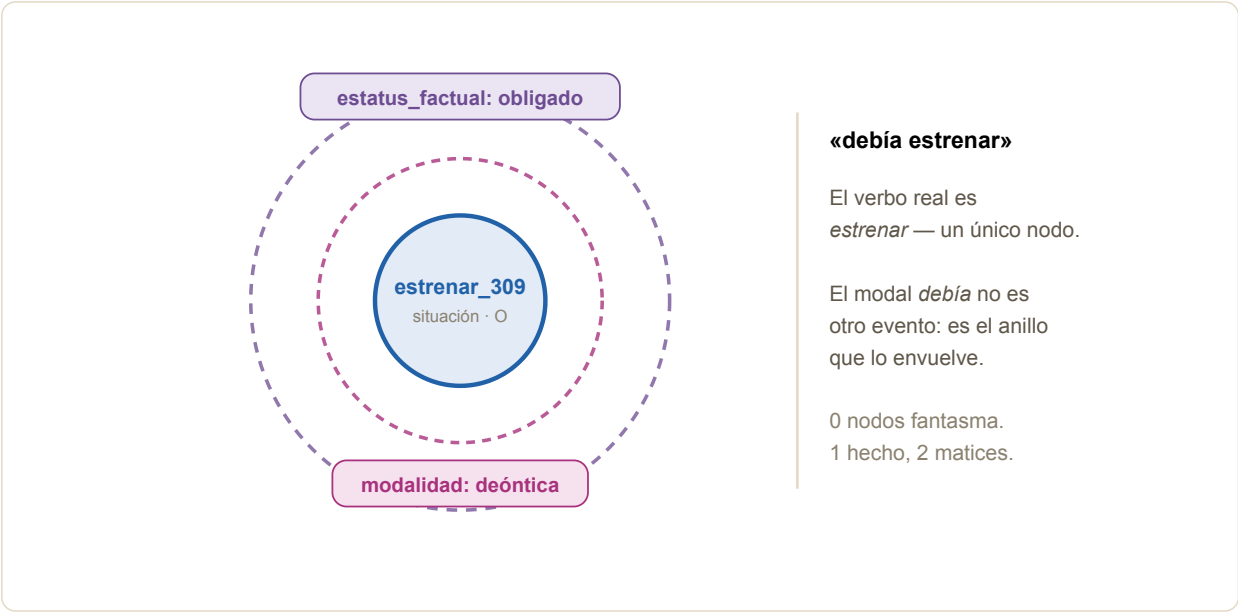


Figura 15.2. El modal no es un evento aparte: es un decorador que envuelve la situación. `estrenar_309` vive solo en el núcleo del eje O; los anillos *modalidad* (eje M) y *estatus factual* (eje T) lo matizan sin convertir *«debía»* en un nodo propio. *«Debía estrenar»* y *«estrenó»* son la misma situación con distinto modo.

EN LA PRÁCTICA · QUERER ≠ QUERER

El lexicon distingue dos *quereres* por la compañía que llevan. *«El estudio quiere estrenar»* es `querer + verbo`: un modal volitivo que decora la situación *estrenar*. *«La directora quiere a su elenco»* es `querer + persona`: un evento emocional, `accion_apreciar`, con su propio nodo. La misma palabra dispara reglas distintas según su patrón sintáctico. Esto es exactamente el oficio del lexicon del capítulo 14.

Tercera presión: idiomas y colocaciones

Llegamos a la trampa más espinosa, y a la tercera oración. *Tomar el pelo* no habla de cabello; *dar luz verde* no enciende ningún semáforo; *echar leña al fuego* rara vez involucra leña. Estas frases hechas (los **modismos**) no se pueden desarmar palabra por palabra sin convertir la frase en un disparate. Y junto a ellas vive un fenómeno más sutil, las **colocaciones**: combinaciones que el idioma fija por costumbre. Se dice *«rodar una película»*, no «conducir una película»; se *«dicta»* una ordenanza, no se «habla» una ordenanza. El verbo correcto no se deduce por lógica; se sabe de memoria.

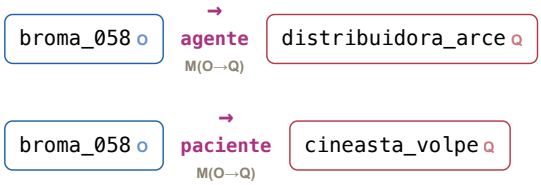
La solución arquitectónica ya la dejamos planteada en el capítulo 14, y aquí se cobra su renta: **en el lexicon, la unidad de traducción no es la palabra suelta, sino el patrón completo**. Los modismos se registran como entradas léxicas con su patrón fijo, apuntando a su significado verdadero, no al literal.

LEXICON · PATRONES

```
[
  {
    "patron": "tomar el pelo a {Q}",
    "tipo_situacion": "accion_bromear",
    "obligatorios": ["agente", "paciente"],
    "ejemplo": "la distribuidora le tomó el pelo al cineasta",
    "nota": "modismo: ignorar toda relación con el cabello"
  },
  {
    "patron": "dar luz verde a {O}",
    "tipo_situacion": "accion_autorizar",
    "obligatorios": ["agente", "tema"],
    "ejemplo": "el comité dio luz verde al rodaje"
  }
]
```

El compilador lee de lo más específico a lo más general, y ese orden es la clave. Si ve *tomar el pelo*, ejecuta la regla de `accion_bromear` y no se molesta en buscar cabello. Si ve *tomar la decisión*, cae en otra entrada, `accion_decidir`. Y solo si ningún patrón fijo encaja recurre al verbo *tomar* en su sentido general de «recibir» o «asir». Lo específico siempre gana a lo genérico. Así, la tercera oración se compila sin drama:

> *La distribuidora le tomó el pelo al cineasta novato con esa fecha.*



El sustantivo del modelo es `accion_bromear`, no `accion_tomar`; el *pelo* jamás aparece como objeto. La frase se guardó por lo que *significa*, no por lo que *dice*. Un parser ingenuo habría registrado a un cineasta despojado de su cabello; el lexicon registra una broma, que es lo que pasó.

PRECEDENTE

La idea de que ciertas combinaciones de palabras forman una *unidad de significado* indivisible no es nuestra: es el corazón de la *Construction Grammar* (Fillmore[24], Goldberg) y de los marcos de *FrameNet*[14], donde un patrón como *tomar el pelo* evoca un marco completo (el del engaño jocoso) con sus roles ya definidos. WQuestions no reinventa esa lingüística; le da un destino estructurado: la tripleta tipada.

El reto, sin embargo, no es técnico sino de escala. El español tiene miles de modismos y decenas de miles de colocaciones; poblar un lexicon a mano con todas sería una obra de años. Aquí entra la buena noticia de la era moderna: **los modelos de lenguaje ya conocen estas frases**. El LLM hace el trabajo pesado (reconoce que *tomar el pelo* es una broma, capta la ironía, resuelve el contexto) y el grafo le ofrece el molde exacto donde verter ese significado. La división de tareas es nítida, y merece su propia sección.

El límite real no está donde parece

El modelo no pretende cubrir el cien por ciento de la lingüística humana, y conviene ser preciso sobre dónde está hoy su frontera. No la digiere entera, y un buen ingeniero conoce los bordes de su herramienta. Importa ver bien *dónde* está el borde: no en los tres casos que acabamos de absorber (esos quedaron resueltos) sino en un terreno más difícil, donde el problema deja de ser de traducción y pasa a ser de **inferencia**. El modelo guarda lo que se dijo; lo que *no se dijo* pero se sigue lógicamente es harina de otro costal.

Negación de alcance. *«El comité no cree que el guion esté listo»* no es lo mismo que *«el comité cree que el guion no está listo»*. ¿A qué situación se le pega la `polaridad: negativa`? El cable existe; decidir dónde clavarlo exige análisis fino, y los analizadores automáticos a veces fallan.

Cuantificación abstracta. *«La mayoría de los socios votó a favor»* no apunta a una persona concreta del eje Q, sino a una clase entera con un cuantificador encima. El modelo puede reificar la votación, pero razonar sobre «la mayoría» como agente difuso le cuesta.

Aspecto fino del tiempo. El español distingue con maestría *empezó a rodar*, *estaba rodando*, *terminó de rodar* y *acababa de rodar*. El núcleo temporal se captura; atrapar esos matices obliga a reificar «fases» del evento, lo que infla el grafo.

Presuposición. *«El estudio dejó de coproducir»* da por hecho que antes coproducía. El modelo guarda el *dejar de* de hoy, pero no deduce solo el hábito pasado: depende de que una IA superior haga esa inferencia y se la entregue ya cocinada.

> **LA DISTINCIÓN**
> Estos no son defectos de nuestro software: son los problemas abiertos de las ciencias cognitivas de las últimas cinco décadas. La diferencia es que cuando el modelo choca con uno, *no se rompe*: guarda una representación parcial y segura, y espera refinamiento.

Hay un patrón común a estos cuatro bordes: ninguno reclama un eje nuevo. Lo que reclaman es una capacidad de *razonar* que vive por encima de la representación. El modelo es, por diseño, **literal y conservador**: registra lo que el lenguaje afirma sin alucinar lo que solo insinúa. Y esa literalidad, lejos de ser un defecto, es precisamente lo que lo vuelve un buen socio para una inteligencia que sí sabe inferir. Lo que para el modelo es un límite, para el sistema completo es una división del trabajo.

El pacto entre el grafo y la inteligencia artificial

Cierro la Parte IV con la reflexión que la vuelve actual. Hoy es tentador decir: *«si un LLM lee libros enteros en segundos, ¿para qué quiero este modelo? Le tiro a la IA los cincuenta mil PDF de la empresa y que ella busque»*. Funciona, sí. Pero es un mal negocio, por dos razones concretas.

EL COSTO

Hacer que la IA relea texto crudo en cada consulta quema *tokens* (cómputo, dinero) una y otra vez por información que pudo guardarse ya destilada. Una historia de cinco páginas en prosa se reduce a unas pocas líneas de átomos estructurados.

LA AMBIGÜEDAD

El texto humano está plagado de *él*, *ella*, *eso*, *la misma*. La IA tiene que adivinar el referente cada vez, y a veces yerra. Un grafo de tripletas no tiene pronombres: cada nodo es quien es, sin lugar para la confusión.

La relación correcta entre el grafo y la IA no es de competencia, sino de **simbiosis**, y cada uno aporta lo que el otro no tiene.

IDEA CLAVE · DOS PIEZAS, UN SISTEMA

La IA es la mente. Entiende el sarcasmo, resuelve el *«ella»*, capta que *tomar el pelo* es una broma, habla con el usuario en su idioma. Es brillante e intuitiva —y, a veces, inventa.

El grafo es la memoria. No olvida, no confunde a Volpe con Arce, distingue una intención de un hecho consumado, audita cada cable contra su signatura. Es literal e incorruptible —y, por sí solo, no infiere.

Pon a la IA de recepcionista que conversa con el usuario, y al grafo de archivo de acero que guarda detrás, y obtienes lo mejor de ambos: la elocuencia de uno sobre la disciplina del otro. La IA traduce el habla en átomos al entrar y los átomos en habla al salir; el grafo custodia la verdad entre una cosa y la otra. Las tres presiones de este capítulo (la nominalización, los modales, los modismos) se absorben justo en esa frontera, en el lexicon, sin tocar la geometría de siete ejes que sostiene todo.

Y con esto cerramos la teoría. Hemos visto las siete coordenadas, el átomo que las une, el verbo que reparte los roles, el lexicon que compila el habla y, ahora, las tres frases que más lo aprietan. La **Parte V** deja la pizarra y baja a la sala de máquinas: el modelo, de extremo a extremo, sobre industrias reales. Empezamos por un sistema de ventas.

16

Un sistema de ventas

Hasta aquí, ejemplos de una línea. A partir de ahora, un negocio entero: un spa que vende servicios, emite comprobantes, paga impuestos y cobra en dos monedas. Es donde la teoría se vuelve código que puedes ejecutar.

Son las 18:05 de un jueves en *Serena Termas*, un spa de barrio. Una clienta (Lucía) acaba de salir de la cámara de vapor, se acerca al mostrador y pide pagar: un circuito termal y un masaje descontracturante. La recepcionista teclea, el sistema calcula el subtotal, le suma el impuesto al consumo, descuenta un cupón de fidelidad, cobra la mitad en efectivo y la mitad con una tarjeta extranjera en otra moneda, e imprime una boleta numerada que esa misma noche viajará al organismo tributario. Toda esa coreografía (servicio, precio, impuesto, comprobante, divisa, fidelidad) ocurre en menos de un minuto y deja un rastro que un contador, un auditor y el dueño tendrán que poder leer meses después. Ese minuto es el examen de este capítulo.

> **EL GIRO DE LA PARTE V**
> Las cuatro primeras partes fueron un viaje conceptual con ejemplos aislados. La Parte V cambia las reglas: cada capítulo toma una empresa real y la modela de punta a punta, con código que puedes correr.

Hasta este punto, el libro te explicó cada decisión de diseño con escenas pequeñas y autocontenidas: una venta de camiseta que se registra, una jugada de fútbol con dos agentes, una ordenanza que entra en vigor a los treinta días. Eran lo bastante chicas para caber en una página. Pero ninguna arquitectura merece llamarse *universal* si no sostiene un negocio completo, con sus impuestos, sus comprobantes legales y sus monedas. Un spa parece el dominio más inofensivo del mundo (toallas, vapor, música suave) y es justo por eso que conviene empezar aquí: bajo esa superficie relajada se esconde un sistema de ventas con casi todas las exigencias de un comercio formal.

> **LO QUE PONE A PRUEBA ESTE CAPÍTULO**
> Un mismo negocio nos obliga a usar, a la vez, casi todas las herramientas de las Partes II y III: los siete ejes, los hechos atómicos, las situaciones reificadas, el catálogo de roles y el lexicon. Y le añade lo que un comercio real no perdona: **autoridad tributaria, comprobantes numerados, impuesto al consumo y cobros en varias divisas**.

Serena Termas en una página

Fijemos el negocio antes de modelarlo. *Serena Termas* opera en una sede con dos cámaras de vapor y una cámara seca a 58 °C, una sala de masajes y una pequeña barra de jugos. El catálogo es corto y claro:

CIRCUITO TERMAL

Veinte minutos por cámara y una ducha helada de cierre. Precio de lista: **S/ 80**. El producto estrella.

MASAJE DESCONTRACTURANTE

Cincuenta minutos con una terapeuta. **S/ 140**. Se vende solo o como complemento del circuito.

BARRA DE JUGOS

Bebidas y fruta. Aquí lo vendido no es un servicio sino un *bien*: importa para el impuesto, como veremos.

Y, como todo comercio, tiene reglas de negocio que la contabilidad de juguete suele maltratar:

Fidelidad. A la décima visita pagada, el circuito de la undécima sale gratis.

Impuesto al consumo. Cada venta lleva un impuesto del 18 % que el spa cobra al cliente y luego entrega al fisco. No es ingreso del negocio: es dinero en tránsito.

Comprobantes. Toda venta exige un documento numerado (boleta o factura) con una serie correlativa que la autoridad tributaria audita.

Multi-divisa. La mitad de la clientela es de paso y paga con tarjetas en dólares; la contabilidad lleva todo en la moneda local.

Usaremos tres clientes para estresar el modelo: **Lucía Rentería**, fiel y a punto de ganar su sesión gratis; **Martín Paredes**, un turista que paga en dos monedas; y **Noa Cifuentes**, que pidió factura a nombre de su empresa. Tres personas bastan para tropezar con casi todos los bordes interesantes.

Paso 1 · Acomodar el negocio en los siete ejes

El primer trabajo de quien modela datos es repartir las entidades del negocio en las coordenadas correctas. Es la decisión que más se paga después: si una cosa cae en el eje equivocado, el sistema cojea para siempre. Así se reparte *Serena Termas*.

Serena Termas, repartido en sus siete coordenadas

Figura 16.1. El dominio del spa sobre los siete ejes. Nota un detalle: la empresa `Serena Termas` y el propio fisco viven en `Q`, porque ambos son agentes que cobran y reciben dinero. Las cámaras viven en `L` (son lugares donde ocurre la sesión) pero también en `O` (son máquinas con temperatura y mantenimiento): una misma cosa puede habitar dos ejes según la pregunta. La banda inferior, el eje `M`, no contiene entidades sino los cables que las conectan.

RECORDATORIO · D1

En `K` viven los conceptos atemporales (`circuito_termal`, `soles`, el estado `pagada`); en `O` y los demás pilares, las instancias situadas: *esta* venta, de las 18:05. La distinción se enunció en el capítulo de la clase.

Repara en la barra de jugos. Un servicio y un jugo parecen lo mismo en la caja registradora, pero el fisco los trata distinto: el servicio es *prestación*, el jugo es *bien*, y de eso depende qué impuesto aplica. Modelarlos como clases distintas en `K` (`servicio_spa` frente a `bien_consumible`) no es pedantería: es lo que permite, más tarde, declarar el impuesto correcto sin un solo *if* incrustado en el código de la caja.

El nudo central · la venta

Si tuviéramos que elegir el evento alrededor del cual orbita todo el negocio, sería **la venta**. Quién la hizo, qué incluyó, dónde, cuándo, cuánto sumó, qué impuesto gravó, qué comprobante la respalda y cómo se pagó: la venta es el punto donde los siete ejes se cruzan. Tal como vimos al tratar el hecho atómico (D3), no la guardamos como una fila gorda, sino como una constelación de tripletas tipadas que se apilan.

La venta de las 18:05 a Lucía (un circuito más un masaje) se reifica como una situación en `O` (D4: lo merece, porque agrupa varias partes, tiene fecha y participa en otras relaciones) y se describe con esta lista de hechos atómicos:

TRIPLETAS

```
(venta_4471) ∈ O
  instancia_de     : transaccion_venta           # K · qué clase de cosa es
  cliente          : lucia_renteria              # Q · quién compró
  vendido_por      : serena_termas               # Q · qué agente vendió
  lugar_de         : sede_central                # L · dónde ocurrió
  inicio           : 2026-06-11T18:05-05:00      # T · cuándo
  comprende        : item_4471_a                 # O · una línea de la venta
  comprende        : item_4471_b                 # O · otra línea
  moneda_base      : soles                        # K · la moneda de la contabilidad
  subtotal         : 220.00                       # N · antes de impuesto
  impuesto_total   : 39.60                        # N · el 18 % del gravado
  total            : 259.60                       # N · lo que se cobra
  documentado_por  : boleta_4471                  # O · el comprobante legal
  estatus_factual  : pagada                        # K · su estado
```

Cada línea de la venta es a su vez un objeto en `O`, porque cada una tiene su propio precio, su propia clase de producto y —esto es lo importante— su propio tratamiento tributario. El masaje y el jugo no se gravan igual; meterlos en el mismo campo perdería esa distinción para siempre.

TRIPLETAS

```
(item_4471_a, instancia_de,   circuito_termal)     # un servicio
(item_4471_a, parte_de,       venta_4471)
(item_4471_a, precio_unitario, 80.00)              # N · soles
(item_4471_a, gravado_con,    impuesto_consumo_18) # K · le aplica el 18 %

(item_4471_b, instancia_de,   masaje_descontracturante)
(item_4471_b, parte_de,       venta_4471)
(item_4471_b, precio_unitario, 140.00)
(item_4471_b, gravado_con,    impuesto_consumo_18)
```

El cable `parte_de` hace de hilo: cada línea cuelga de su venta, y la venta de su comprobante. Recorriendo ese hilo, el sistema reconstruye la jerarquía completa sin un solo `JOIN`. Visto como tripleta, ese andamiaje se lee de un vistazo:

Y en el código real del prototipo, construir esa venta no exige redactar las tripletas a mano: una llamada de pocas líneas consulta el lexicon, desambigua el verbo, crea la situación y le enchufa los cables: validando que ninguno esté mal puesto.

PYTHON

```
venta = ingest_situation(u, lex, "vender",
    roles={
        "cliente":     lucia,
        "vendido_por": serena_termas,
        "lugar_de":    sede_central,
        "inicio":      t_venta,
    },
    complements=["circuito", "masaje"],     # el lexicon arma las dos líneas
    extra={
        "moneda_base":     soles,
        "estatus_factual": pagada,
    },
    sit_id="venta_4471",
)
```

EL SISTEMA SE NIEGA A ACEPTAR DISPARATES

Si un programador distraído intentara poner a `lucia` en el rol `lugar_de`, o un texto donde va una cantidad, `ingest_situation` aborta de inmediato con un error de firma (`SignatureError`). El rol `lugar_de` exige un habitante del eje `L`; una persona no lo es. La validación no es opcional: es la frontera que impide que la basura entre al grafo.

El impuesto al consumo · dinero que solo está de paso

Aquí es donde un sistema de ventas se separa de un cuaderno de apuntes. Ese 18 % que aparece en la boleta no es ingreso del spa: es dinero que el negocio cobra *por cuenta del fisco* y que tendrá que entregarle. Confundir ambas cosas (tratar el impuesto como ganancia) es uno de los errores contables más caros que existen. El modelo lo mantiene separado por construcción.

El impuesto no se guarda como un número suelto pegado a la venta, sino como un objeto con identidad: sabe cuánto es, sobre qué base se calculó, qué tasa aplicó y a quién se le debe.

TRIPLETAS

```
(impuesto_4471) ∈ O
  instancia_de   : impuesto_consumo_18    # K · qué tributo es
  grava          : venta_4471             # O · sobre qué venta recae
  base_imponible : 220.00                 # N · el subtotal gravado
  tasa           : 0.18                   # N · la alícuota vigente
  monto          : 39.60                  # N · lo que se debe
  acreedor       : fisco                  # Q · a quién se le entrega
  estatus_factual: por_remitir           # K · cobrado, aún no entregado
```

Modelarlo así tiene una consecuencia inmediata y deliciosa: la pregunta del contador *«¿cuánto impuesto debo entregar este mes?»* deja de ser un cálculo que alguien reprograma cada vez que cambia la ley, y pasa a ser una simple suma sobre los objetos cuyo `acreedor` es el fisco y cuyo estado sigue en `por_remitir`. Y cuando la tasa cambie del 18 % al 16 %, las ventas viejas conservan su tasa congelada: se calcularon con la alícuota que regía ese día, y esa verdad histórica no se reescribe.

VIGENCIA · D6

Esa congelación es la regla de vigencia en acción. Una tasa de impuesto es una propiedad que cambia en el tiempo; el modelo la reifica con rango `inicio`/`fin` en lugar de sobrescribirla. Un comprobante emitido bajo la tasa antigua sigue siendo legalmente coherente para siempre.

El comprobante · cuando la ley pide un número correlativo

Un comercio formal no puede limitarse a saber que vendió: debe emitir un documento numerado, de una serie correlativa que la autoridad tributaria audita. Si la boleta 4470 existe y la 4472 existe, la 4471 *tiene* que existir; un hueco en la numeración es, para un auditor, una venta sospechosamente desaparecida. El comprobante, por tanto, no es un adorno de la venta: es un objeto de pleno derecho, con su propia clase, su propio número y su propio destino regulatorio.

TRIPLETAS

```
(boleta_4471) ∈ O
  instancia_de   : tipo_boleta_venta      # K · boleta (consumidor final)
  serie          : B001                    # K · la serie correlativa
  numero         : 4471                    # N · correlativo dentro de la serie
  respalda       : venta_4471             # O · qué venta documenta
  emitido_por    : serena_termas          # Q · el emisor
  receptor       : lucia_renteria         # Q · a nombre de quién
  emitido_el     : 2026-06-11T18:06-05:00 # T
  estatus_factual: emitida                 # K · emitida, anulada, etc.
```

La distinción entre *boleta* y *factura* (una para el consumidor final, otra a nombre de una empresa que descontará el impuesto) es, en el modelo, nada más que dos clases distintas en `K`: `tipo_boleta_venta` y `tipo_factura_venta`. Cuando Noa pidió factura a nombre de su empresa, el sistema no necesitó otra tabla ni otro flujo: la misma maquinaria emitió un objeto de otra clase, con un campo extra para el identificador tributario del receptor.

COMPROBANTE DE VENTA

Documento legal, numerado en serie correlativa, que respalda una transacción ante la autoridad tributaria. En WQuestions es un objeto en `O` con su clase (`tipo_boleta` , `tipo_factura`), su número, su emisor, su receptor y un cable `respalda` hacia la venta. La numeración correlativa es una *invariante auditable*: el grafo permite verificar que no falta ninguno.

Multi-divisa · dos monedas, una contabilidad

Martín, el turista, pagó su circuito en dos partes: la mitad en efectivo, en moneda local, y la mitad con una tarjeta en dólares. La trampa clásica es guardar «pagó 22.50» sin decir de qué moneda hablamos (el pecado capital de los sistemas que asumen una sola divisa). En WQuestions, un monto nunca viaja solo: siempre arrastra su unidad, que es un habitante de `K`. Y un pago en otra moneda guarda, además, la tasa de cambio con la que se convirtió, fechada en el día de la operación.

TRIPLETAS

```
(cobro_4480_efectivo) ∈ O
  instancia_de  : pago_recibido
  paga          : venta_4480             # la venta de Martín
  monto         : 40.00
  moneda        : soles                  # K · moneda local
  medio         : efectivo
```

```
(cobro_4480_tarjeta) ∈ O
  instancia_de  : pago_recibido
  paga          : venta_4480
  monto         : 10.75                  # N · lo que se cobró
  moneda        : dolares                # K · ¡otra unidad!
  tasa_cambio   : 3.72                   # N · soles por dólar, ese día
  equiv_base    : 40.00                  # N · su valor en moneda local
  medio         : tarjeta_credito
```

Como cada cantidad declara su unidad, el sistema nunca suma peras con manzanas. Para totalizar la caja del día convierte cada cobro a la moneda base usando la tasa que ya quedó guardada con él —no la de hoy, sino la del momento del cobro—, y la contabilidad cierra al centavo aunque hayan entrado cuatro monedas distintas. La unidad como ciudadano de primera clase, que el capítulo del eje cuantitativo defendió en abstracto, aquí se cobra su sueldo.

El lexicon del spa · enseñarle a la máquina el idioma del negocio

Para que la recepcionista no tenga que aprender jerga de bases de datos, configuramos el lexicon con las palabras exactas del spa. El lexicon es el compilador del que habló la Parte IV: traduce el vocabulario humano a los roles canónicos del modelo, y de paso resuelve la polisemia. El verbo *«tomar»*, por ejemplo, significa cosas distintas según lo que lo acompañe:

PYTHON

```python
lex.register(LexiconEntry(
    verb="tomar",
    situation_type="servicio_spa",
    obligatory=["cliente", "lugar_de"],
    pattern=("sesion",),            # "tomar una sesión" → activa esta regla
))

lex.register(LexiconEntry(
    verb="tomar",
    situation_type="accion_decidir",
    obligatory=["agente", "objeto"],
    pattern=("una_decision",),      # "tomar una decisión" → otra regla
))
```

Y le instalamos el «dialecto corporativo» del spa (D8/D9): un mapa entre las palabras que el personal usa de verdad y las etiquetas canónicas que el catálogo conoce. El usuario final nunca toca esas etiquetas internas; el sistema traduce por debajo.

PYTHON

```python
lex.register_domain_dialect("serena_termas", {
    "cliente":        "agente",
    "sesion":         "servicio_spa",
    "boleta":         "tipo_boleta_venta",
    "factura":        "tipo_factura_venta",
    "sesion_gratis":  "beneficio_fidelidad",
    "igv":            "impuesto_consumo_18",
})
```

Gracias a esto, la recepcionista puede escribirle al sistema *«emítele una boleta a Lucía por una sesión, con su sesión gratis aplicada»* y el grafo lo traduce a sus identificadores internos sin que ella sepa que existen.

LA INTERFAZ ES EL LEXICON, NO EL CATÁLOGO

El catálogo canónico (`impuesto_consumo_18` , `tipo_boleta_venta`) es invisible para quien usa el negocio. Lo que ve es su propio idioma: «IGV», «boleta», «sesión gratis». Dos spas con vocabularios distintos pueden compartir el mismo catálogo y aun así cada uno hablar como siempre habló.

Tres preguntas de negocio, resueltas como recorridos del grafo

El valor de un sistema se mide por la facilidad con que responde las preguntas que de verdad importan a fin de mes. Veamos tres clásicas de este spa y cómo el código las resuelve recorriendo tripletas en vez de encadenar tablas.

Pregunta 1 · ¿Cuántos circuitos pagados lleva Lucía? ¿Le toca el gratis?
No hay `JOIN` que escribir: el motor busca un patrón en el grafo (todas las ventas de clase `circuito_termal` cuyo cliente sea Lucía y cuyo estado sea `pagada`) y las cuenta.

PYTHON

```
n = count(u, Pattern(
    fixed={"cliente": lucia, "estatus_factual": u.ind("pagada")},
    type_constraint=u.ind("circuito_termal"),
))
le_toca_gratis = n >= 10        # la regla de fidelidad, en una línea
```

El modelo no *ejecuta* la regla de negocio por su cuenta (para eso hace falta un motor externo o una IA); lo que hace es entregar el dato prístino y estructurado para que ese motor decida sin equivocarse. La división del trabajo es deliberada: el grafo guarda la verdad, la lógica vive afuera.

Pregunta 2 · ¿Cuánto impuesto debo entregarle al fisco este mes?
Una suma sobre todos los objetos impuesto cuyo acreedor es el fisco y que siguen pendientes de remitir. La ley puede cambiar la tasa: las ventas viejas no se tocan, cada una ya guardó la suya.

PYTHON

```
deuda_fiscal = suma(u, "monto", Pattern(
    fixed={"acreedor": fisco, "estatus_factual": u.ind("por_remitir")},
    type_constraint=u.ind("impuesto_consumo_18"),
))
```

Pregunta 3 · ¿Quedó algún hueco en la numeración de boletas?
La auditoría más temida del negocio se vuelve trivial: pide al grafo todos los números de la serie `B001` y verifica que sean correlativos. Un salto delata una boleta extraviada —o un fraude.

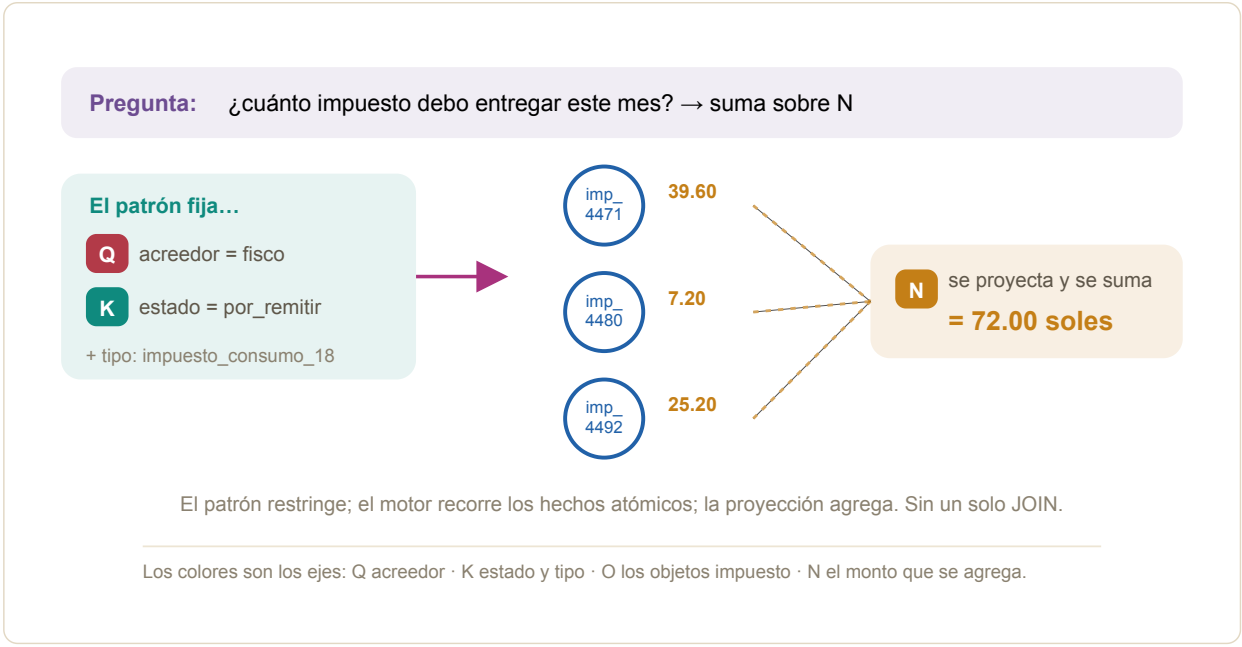


Figura 16.2. Cómo se resuelve una pregunta saltando tripletas. El patrón fija dos roles (acreedor en `Q`, estado en `K`) y restringe por tipo; el motor recorre los objetos impuesto que encajan y proyecta su campo `monto` sobre el eje `N`, donde los agrega. La misma mecánica responde «cuántos circuitos» (un conteo) o «cuánto facturé» (otra suma): cambia la proyección, no la maquinaria.

Detente un segundo aquí, porque esa suma es la primera aparición de algo que volverá en cada dominio del libro. No preguntamos por una venta del spa: preguntamos por todo el impuesto del mes a la vez. Modelar con cuidado la venta de un circuito de aguas termales fue, sin proponérnoslo, dejar listo el reporte fiscal del negocio entero.

El «por qué» de una venta · los cuatro cables

En un negocio real, saber *qué* pasó no basta; a menudo hace falta saber *por qué*. Tal como argumentó el capítulo del «por qué» (D7), no existe un eje «por qué»: el porqué se reparte en cuatro cables distintos, y un sistema de ventas los necesita a casi todos. Dos ejemplos del spa lo dejan claro.

El descuento que tiene una causa normativa. A Lucía se le aplicó su circuito gratis de la décima visita. Ese descuento no es un capricho de la recepcionista: está *justificado* por el contrato de fidelidad, una regla del negocio. El cable `justificado_por` ata el descuento a la norma que lo autoriza, de modo que cualquier auditor pueda rastrear por qué esa venta cobró menos —y comprobar que fue legítimo, no un robo.

El cliente que casi compra. Noa quiso contratar el plan mensual, pero no cerró ese día. En una caja registradora común, ese dato se pierde. En WQuestions se guarda como una *intención*: una situación con `modalidad` volitiva y estado `intencionado`, no `real`. Cuando marketing pregunte «¿a quién llamamos para ofrecer el plan?», Noa salta a la lista; cuando contabilidad pregunte «¿qué planes están activos?», Noa no aparece y nadie le cobra por error.

TRIPLETAS

```
(descuento_4471, aplica_a,        venta_4471)
(descuento_4471, justificado_por, contrato_fidelidad_lucia)   # ← regla normativa

(intencion_noa_01, instancia_de,    contrato_plan_mensual)
(intencion_noa_01, cliente,         noa_cifuentes)
(intencion_noa_01, modalidad,       volitiva)                 # ← lo quería
(intencion_noa_01, estatus_factual, intencionado)             # ← aún no es real
```

La distinción entre lo intencionado y lo real es la misma que separa, en banca, una solicitud de un préstamo desembolsado, o en una clínica un diagnóstico de sospecha de uno confirmado. Que un spa la necesite tanto como un banco es, en sí mismo, una

pequeña confirmación de que el modelo capturó algo general.

El antes y el después · del esquema fragmentado al grafo único

Antes, en SQL. En un esquema relacional clásico, *Serena Termas* vive repartido en media docena de tablas que no se hablan solas: `clientes`, `ventas`, `lineas_venta`, `pagos`, `comprobantes`, `impuestos`. Responder «¿cuánto IGV debo este mes, descontando las ventas anuladas y convertido todo a moneda local?» obliga a encadenar cinco `JOIN` frágiles, filtrar por estado, cruzar tasas de cambio y dejar la mitad de la lógica enterrada en código de aplicación que nadie documenta y que se rompe cada vez que cambia una regla.

SQL

```sql
-- Antes: venta, línea, impuesto y comprobante viven en tablas distintas;
-- la pregunta tributaria obliga a encadenarlas con JOINs frágiles.
CREATE TABLE ventas       (id INTEGER PRIMARY KEY, cliente_id INT, estado TEXT, moneda TEXT);
CREATE TABLE lineas_venta (id INTEGER PRIMARY KEY, venta_id INT, producto_id INT, precio NUMERIC);
CREATE TABLE impuestos    (id INTEGER PRIMARY KEY, venta_id INT, tasa NUMERIC, monto NUMERIC);
CREATE TABLE comprobantes (id INTEGER PRIMARY KEY, venta_id INT, serie TEXT, numero INT);

SELECT SUM(i.monto) AS igv_por_pagar
FROM impuestos i
JOIN ventas v       ON v.id = i.venta_id
JOIN comprobantes c ON c.venta_id = v.id
WHERE v.estado = 'pagada'              -- ojo: ¿y las anuladas?
  AND c.numero IS NOT NULL;           -- la lógica de moneda… vive en otra parte
```

Después, en WQuestions. La misma información es un grafo único de hechos atómicos: cada impuesto es un objeto con su cable `acreedor → fisco` y su `estatus_factual`. La pregunta se vuelve una suma sobre un patrón (sin migrar esquema, sin `JOIN`, sin lógica escondida en procedimientos almacenados). Y cuando la tasa cambie, se edita un solo hecho en el catálogo; las ventas pasadas conservan la suya y el resto del sistema ni se entera.

PYTHON

```python
# Después: la misma pregunta, como un patrón sobre el grafo de hechos.
igv_por_pagar = suma(u, "monto", Pattern(
    fixed={"acreedor": fisco, "estatus_factual": u.ind("por_remitir")},
    type_constraint=u.ind("impuesto_consumo_18")))
```

> *En un sistema de ventas, el impuesto no es un número: es una deuda con un acreedor, una base y una fecha. Tratarlo como un objeto, y no como una columna, es lo que convierte la auditoría en una consulta.*

LA LECCIÓN DEL SPA

Cuánto se vendió · el negocio en una barra

Un sistema de ventas que no sabe responder «¿qué se vendió más?» no merece el nombre. Como cada línea de venta es un objeto con su clase de producto, totalizar por servicio es —otra vez— una agregación sobre el grafo. Este fue el desglose de ingresos de un mes en *Serena Termas*:

Figura 16.3. Ingresos del mes por línea de producto, en miles de soles. El circuito termal sostiene el negocio; los planes mensuales aportan más que la barra de jugos. Ninguna de estas cifras se calculó con una consulta nueva: las cuatro son la misma suma sobre el grafo, agrupada por la clase del producto en `K`.

Balance · el spa fue un cliente amable

Cerremos repasando qué demostró el prototipo al absorber este negocio de punta a punta:

EL VEREDICTO DEL PRIMER DOMINIO INDUSTRIAL

Los siete ejes funcionan juntos. Clientes y fisco en `Q`, ventas y comprobantes en `O`, cámaras en `L`, momentos y vigencias en `T`, precios, impuestos y tasas en `N`, tipos y estados en `K`, y los predicados de `M` cosiéndolo todo.

El sistema de ventas, completo. Comprobantes numerados auditables, impuesto al consumo como dinero en tránsito, cobros en dos divisas con su tasa fechada. Nada de esto pidió ingeniería nueva: cayó sobre la maquinaria existente.

La historia se preserva (D6). Una tasa de impuesto que cambia no reescribe las boletas viejas; cada comprobante sigue siendo coherente con la ley que regía el día en que se emitió.

Cero parches. Todo el negocio se modeló con los roles base de la arquitectura y un puñado de clases en el catálogo. Donde un sistema tradicional pediría media docena de tablas y procedimientos almacenados, aquí bastaron tripletas y un lexicon.

El spa parecía el dominio más inofensivo del libro y resultó ser un sistema de ventas completo, con su autoridad tributaria mirando por encima del hombro. El modelo lo absorbió sin doblarse. Pero un comercio de barrio opera a su propio ritmo: las ventas no llegan por miles ni cambian de estado cada segundo. En el próximo capítulo subimos la apuesta hacia la velocidad y la concurrencia: un servicio on-demand, donde un viaje en taxi cambia de estado una docena de veces en quince minutos y miles ocurren a la vez.

17

Un servicio on-demand

El spa fue un estanque quieto. Aquí entramos a los rápidos: un viaje en taxi donde deciden cuatro protagonistas a la vez —uno de ellos un programa—, las situaciones se encadenan al segundo y el precio sube porque empieza a llover.

Camila Duarte sale de un edificio frente a la plaza a las 22:40 de un viernes, abre la aplicación en el teléfono y toca «Pedir viaje». Veintiocho minutos después baja del auto en la terminal de buses. Para ella ocurrió una sola cosa: pidió un taxi y llegó. Pero en los servidores de la plataforma, entre el toque del botón y el portazo final, pasaron muchas: el sistema rastreó conductores cerca, eligió a uno, el conductor aceptó, condujo hasta la plaza, recogió a Camila, la llevó a la terminal y cerró el viaje. Cada paso dejó su huella, movió un cobro, alteró un mapa de tráfico y (si algo hubiera salido mal) habría abierto un reclamo que alguien en soporte tendría que reconstruir meses más tarde.

> **EL SALTO DEL CAPÍTULO ANTERIOR**
> El spa de *Serena Termas* nos dio una venta que ocurre en un punto del tiempo. Un servicio on-demand nos da algo distinto: un proceso que cambia de estado una y otra vez en minutos, con varios protagonistas entrando y saliendo de escena.

Visto desde la ingeniería de datos, **un viaje en taxi no es un evento: es una cadena de seis situaciones que ocurren una tras otra**, donde entran y salen distintos protagonistas, con relojes cronometrados al segundo y con reglas de precio que dependen del clima y de la demanda en tiempo real. El capítulo anterior fue modelar un estanque quieto: el cliente llega, se relaja, paga y se va. Este nos lanza a la corriente, y nos obliga a poner en práctica tres cosas que hasta ahora vimos sobre todo en teoría: la *agencia contextual*, la *vigencia* de lo que cambia y los cables del *porqué*.

> **LO QUE PONE A PRUEBA ESTE CAPÍTULO**
> Tres tensiones que el spa no tenía. **Agentes múltiples:** no solo personas deciden, también un programa. **Situaciones encadenadas:** media docena de eventos enganchados por orden y por causa, agrupados bajo una unidad comercial. **Causalidad de mercado:** el precio sube por la lluvia, y hay que poder explicarle a un pasajero, meses después, exactamente por qué.

Rumbo en una página

Fijemos el negocio antes de modelarlo. *Rumbo* es una plataforma de transporte que conecta a quien necesita moverse con quien tiene un auto y tiempo. No posee vehículos ni emplea conductores en relación de dependencia: opera el *algoritmo* que los empareja y cobra una comisión por cada viaje. Esa frase aparentemente trivial («opera el algoritmo») es la que vuelve interesante el dominio, porque el algoritmo toma decisiones con consecuencias legales y económicas, y eso lo convierte en un agente de pleno derecho.

En el viaje que seguiremos intervienen cuatro protagonistas. Conviene presentarlos con sus identificadores, porque a partir de aquí los nombraremos por ellos:

🧍

LA PASAJERA · `PASAJERA_DUARTE`

Camila Duarte. El agente humano que solicita el servicio, lo recibe y, al final, lo califica.

🤖

LA PLATAFORMA · `PLATAFORMA_RUMBO`

El software que decide a qué conductor asignarle el pedido. No es un programa pasivo: es quien *asigna*.

👮

EL CONDUCTOR · `CONDUCTOR_SALAS`

Iván Salas. El agente humano que acepta el viaje, recoge a la pasajera y la traslada.

🚗

EL VEHÍCULO · `VEHICULO_4821`

La máquina física (placa 4821) que hace posible el traslado. El *instrumento* del trabajo.

Cuatro protagonistas, una sola transacción

Si le preguntas a cualquiera *«¿quién participa en el viaje?»*, la respuesta obvia es: Camila y su conductor, Iván. Pero en el modelo hay dos participantes más que suelen pasar inadvertidos. Uno es **la plataforma** (un agente de software que toma decisiones autónomas); el otro, **el vehículo**, la máquina que ejecuta el traslado. Para que la base de datos sea auditable, los cuatro tienen que dejar rastro.

Aquí entra en acción una de las decisiones de diseño más liberadoras de la arquitectura: la *agencia contextual* (D5). El rol de `agente` no está reservado a los humanos; lo puede ocupar quien *ejecute la acción del verbo*, sea persona, organización, sensor o software. Y el verbo central de este viaje (*asignar*) no lo ejecutó Camila ni Iván: lo ejecutó el algoritmo. Fue él quien decidió, a las 22:41, darle *ese* pedido a *ese* conductor. Negarle la condición de agente sería falsear quién tomó la decisión.

> **AGENCIA CONTEXTUAL · D5**
> La regla se enunció al hablar de situaciones: el rol `agente` lo pueden ocupar humanos, organizaciones, software o sensores, según el verbo. Una plataforma que *asigna* es tan agente como un vendedor que vende una camiseta. Aquí no la redefinimos; la ponemos a trabajar.

Observa cómo se acomodan los cuatro participantes en la situación de asignación. Tres caen en el eje `Q` —la pasajera (en cuyo nombre se asigna), la plataforma (que asigna) y el conductor (a quien se asigna)— y uno, el vehículo, cae en `O` como instrumento:

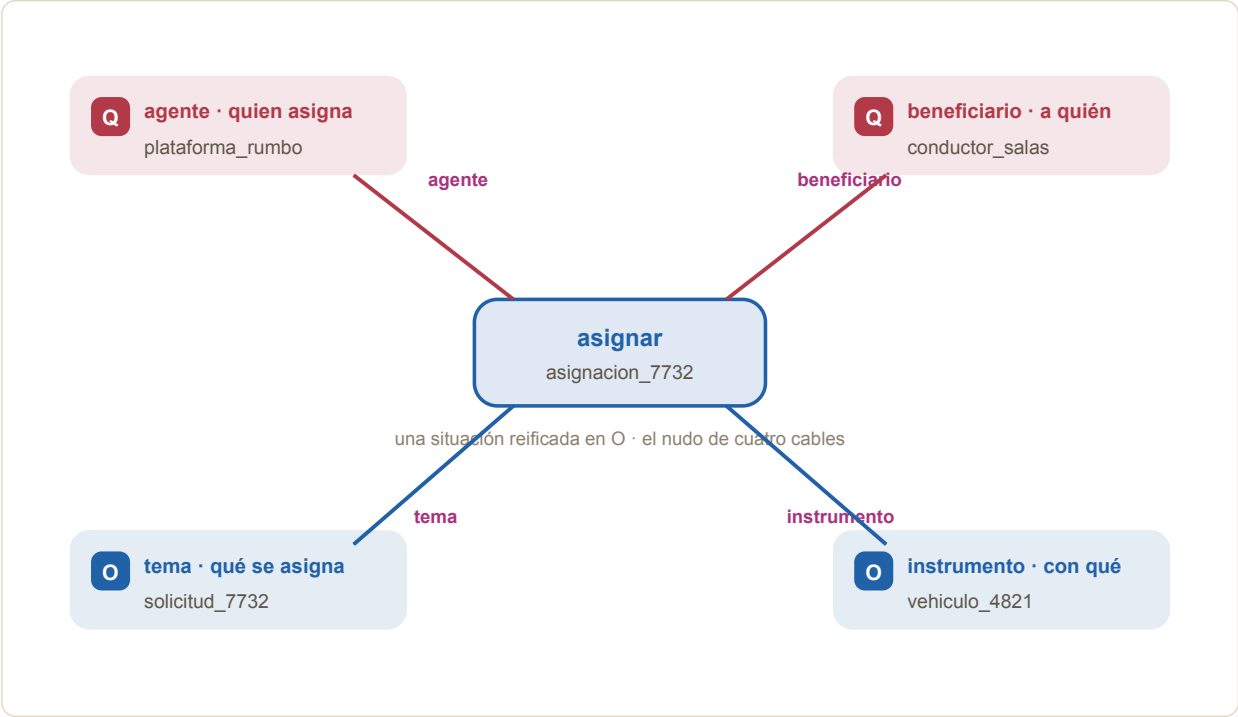


Figura 17.1. Un solo evento, cuatro participantes con roles distintos. La plataforma de software entra en `Q` como *agente*, igual que entraría una persona: para el modelo, quien decide es agente, sea de carne o de código. El vehículo entra en `O` como *instrumento*. Los dos humanos ocupan sus papeles esperables. Que el algoritmo y el conductor compartan el eje `Q` es, precisamente, la decisión D5 en funcionamiento.

Construir esa situación en el prototipo no exige redactar las tripletas a mano. Una llamada consulta el lexicon, desambigua el verbo y enchufa los cuatro cables, validando que cada uno aterrice en su eje correcto:

PYTHON

```
asignacion = ingest_situation(u, lex, "asignar",
    roles={
        "agente":       plataforma_rumbo,   # Q · el algoritmo decide  ← D5
        "tema":         solicitud_7732,     # O · qué se asigna: el pedido de Camila
        "beneficiario": conductor_salas,    # Q · a quién: al conductor
        "instrumento":  vehiculo_4821,      # O · con qué: el auto
        "momento":      t_2241,             # T · la hora exacta
    },
    sit_id="asignacion_7732",
)
```

¿Y SI MAÑANA HAY QUE AUDITAR AL GPS?

El vehículo vive en `O` como instrumento porque, casi siempre, actúa por orden de un humano. Pero si un día el coche tomara una decisión propia (un piloto automático que frena solo, un sensor que reporta una falla), no hay que rediseñar nada: se crea un agente `gps_4821` en `Q`, vinculado al vehículo físico de `O`, y ese agente pasa a ocupar el rol del verbo que ejecutó. El modelo no obliga a elegir entre humano y máquina: la agencia depende del verbo, no de la sustancia de quien actúa.

La cadena de las seis situaciones

Cronológicamente, el viaje de Camila genera seis situaciones reificadas (seis nodos independientes en el grafo), cada una con su hora y sus participantes:

LA CADENA

```
solicitar  →  asignar  →  aceptar  →  recoger  →  trasladar  →  completar
  22:40       22:41       22:42       22:51        22:53         23:08
```

Esas flechas no son adorno tipográfico: son cables reales del modelo, `precede` y su inverso `sigue_a`. La consecuencia es enorme. El sistema conoce el orden exacto de los eventos sin tener que restar marcas de reloj cada vez que se le pregunta. Si alguien quiere saber *«¿qué pasó justo después de que Iván aceptara?»*, el motor sigue el cable al nodo siguiente y responde de inmediato, sin aritmética de fechas. El orden temporal está cosido a la estructura, no calculado a posteriori.

Pero el tiempo no es lo único que liga estos eventos. Algunos ocurren *a causa* de otro. La plataforma no asignó el viaje a Iván por azar a las 22:41: lo hizo **porque** Camila tocó el botón un minuto antes. Esa relación no es temporal —es causal—, y para guardarla el modelo no tiene un eje «por qué», sino cuatro cables distintos (D7). Aquí el adecuado es `motivado_por`:

TRIPLETAS

```
(asignacion_7732, sigue_a,      solicitud_7732)   # cable temporal: el orden
(asignacion_7732, motivado_por, solicitud_7732)   # cable causal: el porqué
```

Y que Iván recoja a Camila no es solo «el siguiente paso en el tiempo»: es una obligación nacida del momento en que aceptó. Por eso la recogida lleva los dos cables a la vez, y no se estorban entre sí:

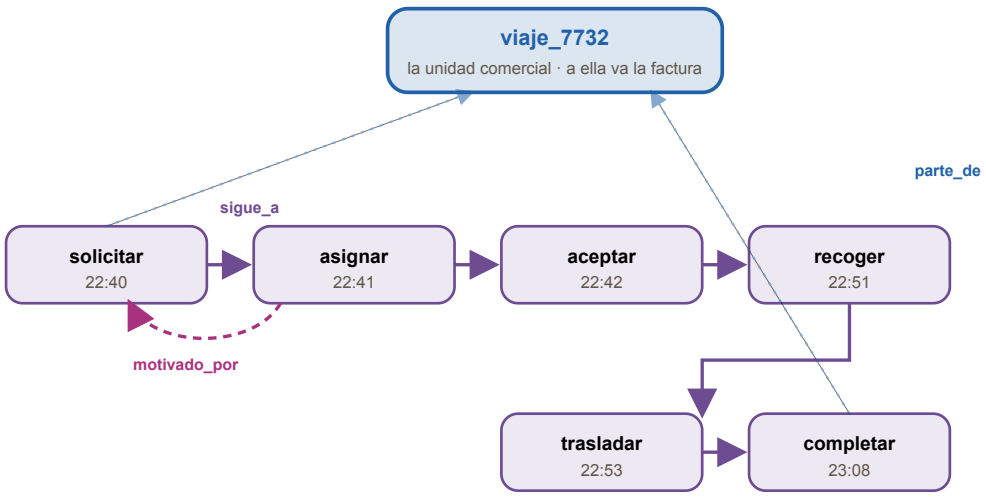
TRIPLETAS

```
(recogida_7732, sigue_a,      aceptacion_7732)   # ← cable temporal
(recogida_7732, motivado_por, solicitud_7732)    # ← cable de motivo
```

> **EL PORQUÉ NO ES UN EJE · D7**
> El capítulo del «por qué» mostró que el porqué se reparte en cuatro cables: `causado_por`, `motivado_por`, `con_finalidad`, `justificado_por`. Una asignación está *motivada* por un pedido; un alza de tarifa está *causada* por un estado del mercado. Distinguirlos es lo que vuelve la auditoría posible.

Los dos cables sirven a públicos distintos. El temporal alimenta los cálculos operativos (cuánto tardó la plataforma en asignar, cuántos minutos esperó la pasajera, qué velocidad media tuvo el traslado). El causal alimenta al auditor, que no quiere saber *cuándo* sino *por qué* se tomó cada decisión. Tenerlos separados, y a la vez en el mismo grafo, es lo que permite responder ambas preguntas sin reconstruir nada.

Falta una pieza. Si estos seis eventos quedaran sueltos en el servidor, serían huérfanos: ninguno sabría a qué viaje pertenece. Por eso los agrupamos bajo una **unidad comercial**: creamos un nodo `viaje_7732` y colgamos de él las seis situaciones con el cable `parte_de`. Ese nodo es la entidad de negocio real (a él se le pega la factura, a él acude soporte si hay un reclamo); las seis situaciones son su interior cronológico.



El orden se sigue por T (sigue_a); la causa, por M (motivado_por); la pertenencia, por O (parte_de). Tres relaciones, un grafo.

Figura 17.2. El viaje como cadena de situaciones encadenadas. Las flechas continuas son el cable temporal `sigue_a`: dan el orden sin aritmética de relojes. La flecha punteada magenta es `motivado_por`, el cable causal que ata la asignación al pedido que la provocó. Las líneas tenues hacia arriba son `parte_de`: cada situación cuelga de `viaje_7732`, la unidad comercial. Reconstruir el viaje entero es, literalmente, recorrer estos cables.

Conviene detenerse en que tres preguntas distintas conviven sobre las mismas seis situaciones, y cada una se responde siguiendo un cable que vive en un eje diferente:

T ¿en qué orden? El cable `sigue_a` es asunto del eje del tiempo. Da la secuencia sin restar relojes: preguntar «¿qué vino después de aceptar?» es seguir una flecha, no calcular una diferencia de horas. Es lo que alimenta toda métrica operativa: espera de la pasajera, duración del traslado, latencia de la asignación.

M ¿por qué ocurrió? El cable `motivado_por` es un predicado del eje *cómo*, el que enlaza hechos con hechos. No dice cuándo, sino qué provocó qué: la asignación existe porque hubo una solicitud. Es lo que un auditor persigue cuando reconstruye una decisión.
O ¿de qué viaje es parte? Y el cable `parte_de` ata cada situación a la unidad comercial en el eje de los objetos. Tres ejes, tres preguntas, un mismo grafo.

Tarifa dinámica · explicar el caos del mercado

Hay un rasgo de este negocio que tensa de verdad la estructura de datos: el precio no es fijo. Cuando llueve, o cuando hay más pasajeros que autos disponibles, la tarifa sube. Un mismo trayecto que ayer costó 18 unidades hoy cuesta 30. El reto no es *cobrar* ese recargo (eso lo hace cualquier sistema) sino *explicarlo* seis meses después, cuando la pasajera escribe enfadada preguntando por qué le cobraron casi el doble.

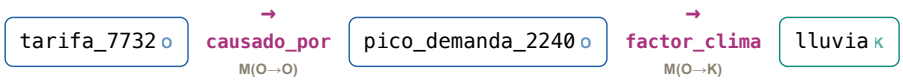
Un sistema mediocre pegaría a la factura un número suelto: `multiplicador: 1.67`. Funciona para el cobro y arruina la auditoría. Ese número no sabe *por qué* era 1.67; no recuerda que esa noche llovía sobre la plaza ni que había tres veces más pedidos que autos. La explicación se evaporó en el instante mismo del cobro.

La solución del modelo es **reificar el estado del mercado**: convertir «esa noche, en esa zona, había pico de demanda y lluvia» en un objeto con identidad propia en `O`, y atar la tarifa a él con el cable causal `causado_por` (D7). El estado deja de ser un adjetivo perdido y pasa a ser un hecho consultable, con su lugar, su hora y su intensidad:

PYTHON

```
# 1 · El estado de la calle como objeto con identidad
estado_mercado = u.add_individual(Individual(id="pico_demanda_2240", axis=Axis.O))
u.assert_fact(estado_mercado, "instancia_de", "estado_alta_demanda")
u.assert_fact(estado_mercado, "lugar_de",     zona_plaza)        # L · dónde
u.assert_fact(estado_mercado, "momento",      t_2240)            # T · cuándo
u.assert_fact(estado_mercado, "factor_clima", lluvia)            # K · por qué subió
u.assert_fact(estado_mercado, "multiplicador", n_1_67)           # N · cuánto

# 2 · La tarifa, atada a su causa
tarifa = u.add_individual(Individual(id="tarifa_7732", axis=Axis.O))
u.assert_fact(tarifa, "paga_de",     viaje_7732)
u.assert_fact(tarifa, "monto",       n_30_00)
u.assert_fact(tarifa, "moneda",      moneda_local)
u.assert_fact(tarifa, "causado_por", estado_mercado)   # ← el porqué, como cable
```

La diferencia, a la hora de operar el negocio, es nítida. Visto como tripletas, el recargo ya no es un número anónimo, sino un hecho que apunta a su causa:



Cuando Camila escriba *«¿por qué pagué 30 si la otra vez fueron 18?»*, el sistema no necesita que un empleado bucee en registros de tráfico. Sigue el cable `causado_por` y responde solo: *«La tarifa se calculó por un estado de alta demanda registrado a las 22:40 en la zona de la plaza, con un factor de lluvia, que aplicó un multiplicador de 1.67»*. La justificación estaba guardada como un hecho, no reconstruida a mano.

UN PORQUÉ NO ES UN CAMPO: ES UN CABLE A UN HECHO

La tentación es modelar el recargo como una columna numérica. Pero un número no se puede auditar: no recuerda su causa. Al reificar el estado del mercado y unirlo con `causado_por`, el «por qué subió» deja de ser folclore de soporte y se vuelve un dato de primera clase, con su lugar, su hora y su factor. La auditoría más temida del rubro se convierte en seguir un cable.

Cuando algo se cancela · el arte de no borrar nada

El rubro del transporte vive de cancelaciones. Alguien pide un auto y se arrepiente; un conductor acepta y se le pincha una llanta. La pregunta perenne de quien programa es: ¿qué hago con el viaje cancelado? ¿Lo borro? ¿Lo marco en rojo? Ambas respuestas destruyen información. Borrar deja un hueco; sobrescribir miente sobre el pasado.

La convención del modelo es la **inmutabilidad**: nunca se borra, nunca se reescribe la historia. Una cancelación no es un cambio sobre el viaje viejo: es *un evento nuevo* que lo apunta. Y ese principio es exactamente la *vigencia* (D6) en su forma más concreta: el estado del viaje no se modifica en su lugar, sino que se registra como un hecho fechado que abre un intervalo de validez.

VIGENCIA · D6

La regla se enunció con las propiedades que cambian en el tiempo: en vez de sobrescribir, se reifican con rango `inicio` / `fin`. Un viaje que pasa de `solicitado` a `cancelado` no pierde su pasado: gana una línea nueva, fechada. La verdad de cada instante queda intacta para siempre.

PYTHON

```
# La cancelación es un evento nuevo, con su autor y su hora
cancelacion = ingest_situation(u, lex, "cancelar",
    roles={
        "agente":  pasajera_duarte,    # Q · quién canceló (queda registrado)
```

```
        "tema":    viaje_7732,         # O · qué se cancela: el viaje
        "momento": t_2243,             # T · cuándo
    },
    sit_id="cancelacion_7732",
)
u.assert_fact(cancelacion, "cancela", viaje_7732)   # el cable hacia el original
```

El cambio de estado del viaje no machaca un campo: se asienta como un hecho con vigencia, igual que un libro contable que solo agrega asientos y jamás tacha. El estado anterior conserva su intervalo cerrado; el nuevo abre el suyo:

TRIPLETAS

```
(viaje_7732, estado, solicitado,  inicio=22:40, fin=22:43)   # vigente y luego cerrado
(viaje_7732, estado, cancelado,   inicio=22:43, fin=∞)        # el estado vigente ahora
```

Nacieron varios hechos nuevos (la situación de cancelación con su autor y su hora, el cable `cancela` y la línea de estado fechada) y ninguno destruyó nada. El viaje sigue entero en el grafo, solo que ahora su estado vigente es `cancelado`. Cuando la empresa calcule los viajes facturables del día, el código simplemente ignora los que tengan estado vigente `cancelado`. La base de datos se vuelve un registro histórico indestructible: perfecto para una disputa legal, donde la pregunta no es «¿cuál es el estado hoy?» sino «¿en qué estado exacto estaba a las 22:42?».

Tres consultas operativas

El valor de un sistema se mide por la facilidad con que responde lo que de verdad importa al cierre del día. Veamos tres preguntas críticas del rubro, resueltas como recorridos del grafo en lugar de cadenas de tablas.

Consulta 1 · ¿Adónde llevó Iván a Camila?
El motor busca un patrón (las situaciones de clase `trasladar` donde el agente sea Iván y la pasajera sea Camila) y proyecta el destino. No hay tablas que unir: hay un patrón que se satisface.

PYTHON

```
r = query(u, Pattern(
    fixed={"agente": conductor_salas, "pasajero": pasajera_duarte},
    ask={"destino": Var()},
    type_constraint=u.ind("traslado"),
))
# → terminal_buses
```

Consulta 2 · ¿Cuántos viajes completó Iván esta noche?
Un conteo sobre el patrón de viajes con agente Iván y estado vigente `completado`.

PYTHON

```
n = count(u, Pattern(
    fixed={"agente": conductor_salas, "estatus_factual": u.ind("completado")},
    type_constraint=u.ind("viaje"),
))
```

Repara en algo que conviene no dejar pasar: este conteo es *estructuralmente idéntico* al que usamos en el spa para contar los circuitos pagados de Lucía. Cambian los nombres de los roles y de las clases (`conductor_salas` en vez de `lucia`, `viaje` en vez de `circuito_termal`) pero la maquinaria es la misma. La uniformidad del modelo permite reciclar la misma consulta entre un spa de barrio y una plataforma transnacional, dos negocios sin nada en común salvo su forma profunda.

Consulta 3 · ¿Por qué la tarifa fue de 30?
Se siguen los hechos de la tarifa y se filtra el cable causal. La respuesta es el estado del mercado que la provocó: no un número opaco, sino un hecho con lugar y hora.

PYTHON

```
hechos = u.facts_about(u.ind("tarifa_7732"))
causa  = [f for f in hechos if f.role == "causado_por"]
# → pico_demanda_2240   (alta demanda + lluvia, 22:40, zona de la plaza)
```

De un viaje a la flota entera

Hasta aquí seguimos un viaje: la solicitud de Camila, la asignación a Iván, el cobro de la tarifa. Pero quien opera Rumbo no vive en un viaje, vive en diez mil. Y la pregunta que paga el sueldo no es «¿adónde llevó Iván a Camila?», sino «¿qué zona se queda sin autos a las siete de la tarde?» o «¿qué conductor completa más carreras de las que rechaza?». Esas no son consultas nuevas: son las mismas situaciones de viaje, agrupadas por destino, por hora, por conductor. El conteo de los viajes de Iván que vimos recién es ya un corte de ese tablero; correrlo sobre cada conductor da el ranking de la flota.

PYTHON

```
# Viajes completados hacia una zona — un corte; el tablero cruza todas las zonas y conductores
count(u, Pattern(fixed={"destino": u.ind("terminal_buses"), "estatus_factual": u.ind("completado")},
                 type_constraint=u.ind("viaje")))
```

El antes y el después · del esquema fragmentado al grafo único

Antes, en SQL. El esquema clásico de un servicio de transporte reparte la realidad en tablas separadas que no se hablan solas: `viajes`, `conductores`, `pasajeros`, `tarifas`, `eventos_viaje`. Reconstruir la cadena completa (qué ocurrió, en qué orden y por qué el precio subió) obliga a encadenar `JOIN` en cascada y luego ordenar en código por marca de tiempo. Y la pregunta que de verdad duele (*«¿cuál fue la causa concreta de que esa tarifa subiera a 30 ese minuto?»*) sencillamente no tiene respuesta: no existe una tabla de causas; el motivo (lluvia, alta demanda) se perdió o agoniza en un campo de texto libre que nadie consulta con coherencia.

SQL

```
-- Antes: el viaje, sus eventos y su tarifa viven en tablas distintas;
-- el orden se reconstruye ordenando por timestamp y el "porqué" no existe.
CREATE TABLE viajes        (id INTEGER PRIMARY KEY, pasajero_id INT, conductor_id INT, estado TEXT);
CREATE TABLE eventos_viaje (id INTEGER PRIMARY KEY, viaje_id INT, tipo TEXT, ts TIMESTAMP);
CREATE TABLE tarifas       (id INTEGER PRIMARY KEY, viaje_id INT, monto NUMERIC, multiplicador
NUMERIC);

SELECT e.tipo, e.ts
FROM eventos_viaje e
JOIN viajes v ON v.id = e.viaje_id
WHERE v.id = 7732
ORDER BY e.ts;                      -- el orden se calcula; no está en la estructura
-- ¿la causa del multiplicador 1.67? No hay columna que lo diga.
```

Después, en WQuestions. La misma realidad es un grafo único. `viaje_7732` agrupa las seis situaciones reificadas (`solicitar`, `asignar`, `aceptar`, `recoger`, `trasladar`, `completar`) con cables `parte_de` y `sigue_a`. La tarifa es un objeto en `0` unido al

estado de alta demanda por `causado_por`. Reconstruir el viaje es seguir los cables desde `viaje_7732`; obtener la causa del precio es leer un solo hecho. No hay `JOIN`, no hay pegamento en código que ordene por fecha, no hay pregunta que el esquema no admita —porque el motivo nunca fue un número: siempre fue un evento con identidad.

```python
# Después: la cadena se recorre por el cable; la causa se lee como un hecho.
cadena = u.follow(u.ind("viaje_7732"), "parte_de")        # las seis situaciones
causa  = u.facts_about(u.ind("tarifa_7732"), role="causado_por")   # un solo paso
```

> *En un servicio on-demand, el porqué de un precio no es una columna: es un evento con lugar y hora. Reificar el estado del mercado, y no esconderlo en un número, es lo que convierte el peor dolor de cabeza de soporte en una sola consulta.*

LA LECCIÓN DEL TAXI

Lo que el taxi enseñó que el spa ocultaba

El spa fue un cliente amable: una venta que ocurre en un punto del tiempo. El servicio on-demand exigió más, y el modelo lo sostuvo sin un solo parche. Cerremos con el balance de lo que demostró el prototipo al absorber este dominio de punta a punta.

EL VEREDICTO DEL SEGUNDO DOMINIO INDUSTRIAL

Agentes de software (D5). Le dimos poder de decisión al algoritmo: la plataforma ocupa el rol de `agente` en `Q`, junto al conductor humano. El modelo no distingue de qué sustancia está hecho quien decide; solo le importa que ejecute el verbo.

Situaciones encadenadas. Seis eventos enganchados por orden (`sigue_a`) y por causa (`motivado_por`), todos agrupados bajo una unidad comercial (`parte_de`). El orden vive en la estructura, no en una resta de relojes.

Los porqués del mercado (D7). «La lluvia» y «el alza» dejaron de ser folclore: se reificaron como un estado en `O` y se ataron al precio con `causado_por`. La tarifa más reclamada del rubro se explica sola.

Inmutabilidad y vigencia (D6). Una cancelación no borra ni reescribe: agrega un evento y un estado fechado. El grafo se vuelve un registro histórico que responde no solo «¿cómo está hoy?» sino «¿cómo estaba a las 22:42?».

La prueba de código superó a la simulación: no hubo que inventar un rol nuevo ni torcer el catálogo de la arquitectura para sostener una operación que cambia de estado una docena de veces en media hora y multiplica precios con la lluvia. En el próximo capítulo subimos otra vez la apuesta, pero en una dirección distinta. En una historia clínica no hay viajes rápidos ni tarifas que suban con el clima: hay diagnósticos, fármacos, contraindicaciones, protocolos y una densidad de información donde un error no cuesta dinero, sino salud. Veamos cómo el modelo, que ya domó un comercio y una plataforma, se enfrenta al peso de la medicina sin doblarse.

18

Una historia clínica

El spa y el taxi eran negocios transaccionales: eventos que ocurren, se cobran y se olvidan. La medicina es lo contrario. Aquí lo valioso no es la cita, sino el conocimiento denso que nace dentro de ella —y que tiene que sobrevivir décadas.

Son las 09:40 de un martes en un consultorio de cardiología. El paciente (el mismo Vega que meses atrás entró por urgencias con una fibrilación) llega ahora a un control de rutina. No hay drama: se sienta, cuenta que duerme mal y que a veces siente el corazón «a destiempo», la doctora le toma la presión, repasa su última analítica, confirma que la hipertensión que arrastra desde el episodio agudo sigue ahí pero algo peor, ajusta la dosis de un fármaco, anota una contraindicación que el paciente no conocía y agenda otro control en treinta días. Quince minutos. Cuando Vega sale por la puerta, la cita ya terminó; pero casi nada de lo que importa terminó con ella. El diagnóstico seguirá vigente un año, la prescripción habrá que reevaluarla mes a mes, la alergia recién descubierta deberá gritar cada vez que alguien intente recetarle ese fármaco. Esos quince minutos dejaron un sedimento que un médico distinto, en un hospital distinto, dentro de cinco años, tendrá que poder leer sin ambigüedad. Ese sedimento es el examen de este capítulo.

> **EL CAMBIO DE MARCHA**
> Los dos dominios anteriores eran *transaccionales*: la venta del spa, el viaje del taxi. El valor estaba en la línea de tiempo. En la medicina el valor está en el peso intelectual de cada decisión, y ese peso no caduca cuando termina la cita.

Hasta aquí, la Parte V ha modelado negocios cuyo corazón es la transacción. Una sesión de spa empieza, termina, se cobra y el mundo sigue girando; un viaje en taxi ocurre, se paga y se archiva. Modelarlos fue, en buena medida, aprender a usar bien las herramientas y luego repetir la fórmula cambiando las palabras del lexicon. La medicina no concede esa comodidad. Una consulta es un evento, sí, pero a nadie le interesa cuánto duró: lo que interesa es el **contenido** que se produjo dentro (un diagnóstico, una receta, una alerta de alergia, un control programado), y ese contenido tiene que vivir mucho más que la cita que lo engendró.

Por eso este capítulo cambia de foco. No vamos a presumir de soluciones ya hechas: vamos a mostrar *cómo se modela un dominio nuevo desde cero*. La medicina es el banco de pruebas perfecto para esa metodología porque es un campo en el que nadie llega con el esquema resuelto: hay que elicitarlo, decisión a decisión, viendo dónde la intuición humana acierta y dónde la rigidez del modelo nos corrige a tiempo.

> **LA PREGUNTA QUE GOBIERNA EL CAPÍTULO**
> ¿Aguanta el modelo un dominio para el que *no* fue diseñado, sin inventar maquinaria nueva? La apuesta: una consulta médica completa (síntoma, medición, diagnóstico, prescripción y control futuro, con su incertidumbre clínica, su evidencia y su vigencia en el tiempo) cae sobre los mismos siete ejes, las mismas situaciones reificadas y el mismo lexicon que ya conoces. La fricción será mínima. Donde la haya, la mostraré tal cual.

Cómo se elicita un dominio que no conoces

Modelar un dominio nuevo no empieza escribiendo tripletas. Empieza con tres preguntas que uno le hace al negocio, en este orden, antes de tocar el teclado. Son la columna vertebral de la metodología, y conviene enunciarlas porque sirven igual para una clínica que para una aseguradora o una fábrica.

📦

¿CUÁL ES EL EVENTO ANCLA?

El acontecimiento alrededor del cual orbita todo lo demás. En el spa era la venta; aquí es **la consulta**. De él colgarán las demás piezas.

🧩

¿QUÉ NACE DENTRO DEL EVENTO?

Los «objetos de información» que la consulta produce: síntoma, medición, diagnóstico, prescripción, control. Cada uno con vida propia.

🔮

¿SOBRE QUÉ SE PREGUNTARÁ MAÑANA?

La prueba decisiva de qué merece volverse entidad. Si alguien va a interrogar un dato en el futuro, ese dato necesita identidad propia.

La tercera pregunta es la más sutil y la dejaremos para el final del capítulo, porque es la que de verdad separa a quien modela bien de quien acumula campos. Por ahora basta con la primera: el evento ancla. En medicina, ese evento es la consulta.

La consulta como carpeta maestra

Igual que en el taxi un `viaje` abrazaba a todos sus pasos menudos, la consulta médica actúa como una gran carpeta articuladora. Una revisión típica produce cinco piezas internas, y conviene nombrarlas como las nombraría el propio médico:

1 · El síntoma. Lo que dice el paciente, en sus palabras: «duermo mal y siento el corazón a destiempo». Subjetivo, fechado, con una intensidad que mañana querremos comparar.
2 · La medición. Los números crudos que toma la doctora: presión en 148/95. Objetivos, con unidad y momento.
3 · El diagnóstico. La conclusión clínica: hipertensión de grado 2. Una inferencia, no un hecho duro, y con fecha de caducidad.

4 · La prescripción. La conducta indicada: subir el antihipertensivo a 20 mg, una vez al día. Tiene un porqué, una finalidad y un control de seguridad.
5 · El control. El plan a futuro: nueva cita en treinta días. No es algo que pasó, sino algo *previsto*, y el sistema debe saber distinguirlo de lo real.

En el modelo, estas cinco piezas **no son campos de texto** de la consulta. Son cinco situaciones reificadas independientes que cuelgan del evento ancla con el viejo y confiable cable `parte_de`. La consulta reúne; cada pieza vive su propia vida.

TRIPLETAS

```
(consulta_8842) ∈ O
  instancia_de : consulta_cardiologia    # K · qué clase de evento es
  agente       : dra_navarro             # Q · quién atiende
  paciente     : vega                    # Q · a quién se atiende
  lugar_de     : consultorio_card_2      # L · dónde ocurre
  inicio       : 2026-06-09T09:40-05:00  # T · cuándo
  motivo       : control_hipertension    # K · por qué viene

(sintoma_8842    ∈ O,  parte_de,  consulta_8842)
(medicion_8842   ∈ O,  parte_de,  consulta_8842)
(diag_hta_8842   ∈ O,  parte_de,  consulta_8842)
(prescrip_8842   ∈ O,  parte_de,  consulta_8842)
(control_8842    ∈ O,  parte_de,  consulta_8842)
```

La arquitectura no se inmuta: una entidad grande que contiene un racimo de sub-situaciones, exactamente como la venta del spa contenía sus líneas. Que la consulta merezca reificarse no es un capricho (cumple de sobra las condiciones para volverse situación

en `O`: agrupa varias partes, está fechada, tiene participantes con roles y, sobre todo, otras cosas le harán referencia). Lo que vuelve fascinante a la medicina no es el contenedor, sino el interior de cada pieza.

> **RECORDATORIO · D4**
> Reificar la consulta (convertir el evento en una entidad de pleno derecho en `O`) es justo lo que la regla de situaciones autoriza cuando un evento agrupa partes, lleva participantes y será referenciado por otros hechos. La condición se enunció al tratar las situaciones; aquí la cobramos.

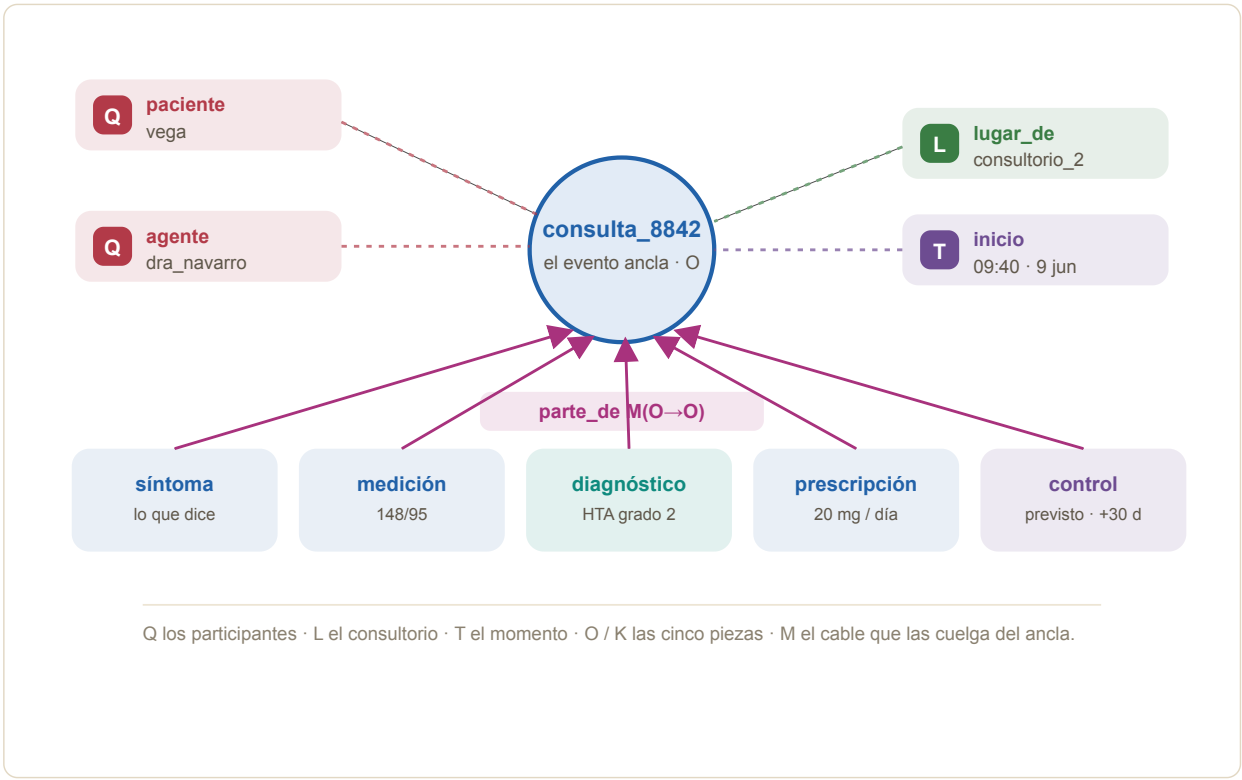


Figura 18.1. La consulta reificada como evento ancla. Arriba, sus participantes repartidos en los ejes (`vega` y la doctora en `Q`, el consultorio en `L`, el momento en `T`). Abajo, las cinco sub-situaciones que nacen dentro de ella y cuelgan con `parte_de`. La consulta no «guarda» el diagnóstico como un campo: lo *articula* como una entidad vecina con vida propia.

El bache: cuando el eje O se queda corto

Aquí apareció la primera fricción real al programar el prototipo de este dominio, y vale la pena contarla porque enseña más que cualquier acierto. La intuición humana, al modelar la medición de presión, dice: «el *tema* de esta medición es la presión arterial». Y al modelar la receta: «el *tema* de esta prescripción es el fármaco». Parece obvio.

Pero detente. **«Presión arterial» y «el antihipertensivo» no son objetos físicos con identidad propia.** No puedes guardar «la presión arterial universal» en un cajón: es una *magnitud*, una categoría teórica. La caja de pastillas que Vega compra en la farmacia sí es un objeto del eje `O` (tiene lote, vencimiento, ubicación), pero el concepto científico del fármaco es una *clase*, un habitante del eje `K`. El instinto humano mezcla, sin notarlo, lo físico con lo conceptual.

El catálogo estricto del modelo no se deja engañar. El cable `tema` tiene una firma que solo admite conectar `O` con `O`. Si intento usarlo para apuntar de una medición (en `O`) hacia la magnitud presión (en `K`), el sistema aborta con un error de firma y me regaña por mezclar peras con manzanas.

EL MODELO SE NIEGA A ACEPTAR EL DISPARATE

Conectar una situación de `O` con una clase de `K` usando un cable cuya firma exige `O→O` lanza un `SignatureError`. No es un estorbo: es la frontera que impide que un dato mal tipado entre al grafo. El error no estaba en el modelo; estaba en mi cabeza, que confunde el objeto concreto con la idea abstracta todo el tiempo. La rigidez de las matemáticas me salvó de un desastre semántico que más tarde habría confundido a cualquier máquina que leyera el expediente.

¿La salida? No forzar el cable genérico, sino declarar *cables de dominio*: roles propios de la medicina, diseñados de fábrica para apuntar hacia el eje `K`. Son parte de la elicitación: cada industria muy técnica reclama un puñado de cables exclusivos para sus conceptos abstractos.

PYTHON

```
# Mal · forzar el cable genérico "tema" hacia una clase → SignatureError
# Bien · declarar cables de dominio cuya firma apunta a K

u.assert_fact(medicion_8842, "medida_evaluada",      presion_arterial)  # O → K
u.assert_fact(prescrip_8842, "medicamento_prescrito", antihipertensivo) # O → K
```

DOS LECCIONES DE LA FRICCIÓN

El catálogo universal cubre casi todo, pero no todo. Los roles base sirven para la inmensa mayoría de los hechos; cada dominio altamente técnico (medicina, química, derecho) pedirá unos pocos cables propios para sus abstracciones. Eso es esperado, no un fallo.

El error fue del modelador, no del modelo. La firma estricta no es burocracia: es la que distingue «la caja de pastillas» (objeto, `O`) del «fármaco como concepto» (clase, `K`). Sin esa distinción, una IA no podría razonar sobre interacciones farmacológicas.

Diseccionar un diagnóstico: duda, evidencia y tiempo

Un diagnóstico es la pieza más rica de toda la consulta, porque cruza a la vez tres dimensiones que un sistema de juguete suele aplastar en un solo campo de texto. El modelo las resuelve por separado, sin esfuerzo.

1 · El nivel de duda

Un diagnóstico casi nunca es una verdad tallada en piedra: es lo que la doctora *cree* en ese instante, con la evidencia que tiene. Para no perder esa incertidumbre clínica, decoramos la situación con su modalidad epistémica y un estado factual que puede ser *sospechado*, *confirmado* o *descartado*. La diferencia entre «sospecho una arritmia» y «confirmo una arritmia» no es un matiz de prosa: cambia qué se puede recetar y qué hay que vigilar.

TRIPLETAS

```
(diag_hta_8842, modalidad,       epistemica)   # K · pertenece al saber, no al deseo
(diag_hta_8842, estatus_factual, confirmado)    # K · sospechado | confirmado | descartado
```

Esta es la misma distinción que en el spa separaba la venta real de la mera intención de compra, o que en banca separa una solicitud de un préstamo desembolsado. Que un dato pueda ser «todavía no seguro» sin por eso ser falso es una necesidad que

aparece en cada dominio serio.

2 · La evidencia

Un diagnóstico no nace de la nada: nace de los números. Si la doctora escribe «hipertensión», es porque vio la medición de presión. Esa conexión de causa y efecto no se pierde en un comentario: se ata con un cable causal. Tal como argumentó el capítulo del «por qué», no hay un eje «por qué»; el porqué se reparte en cables distintos, y aquí el que corresponde es el que apunta a la *razón* que motivó la conclusión.

TRIPLETAS

```
(diag_hta_8842, motivado_por, medicion_8842)   # M · la medición es la evidencia
```

Con ese único cable, la medición deja de ser un número flotando y se vuelve la **evidencia** del diagnóstico. Si un auditor médico o una máquina revisan el expediente y preguntan «¿en qué se basó este diagnóstico?», la respuesta no exige interpretar prosa: se lee siguiendo el cable. La justificación viaja con el dato.

3 · El viaje en el tiempo

Esta es la dimensión que separa el software hospitalario de verdad del software de juguete. Los diagnósticos caducan y mutan. Supongamos que tras el episodio agudo, en marzo, a Vega se le asignó hipertensión de grado 1. En esta consulta de junio la doctora confirma que empeoró: ahora es grado 2. Si se reescribiera el diagnóstico viejo encima del nuevo, se borraría la historia del paciente —un crimen, no solo de negligencia clínica, sino de imposibilidad forense—.

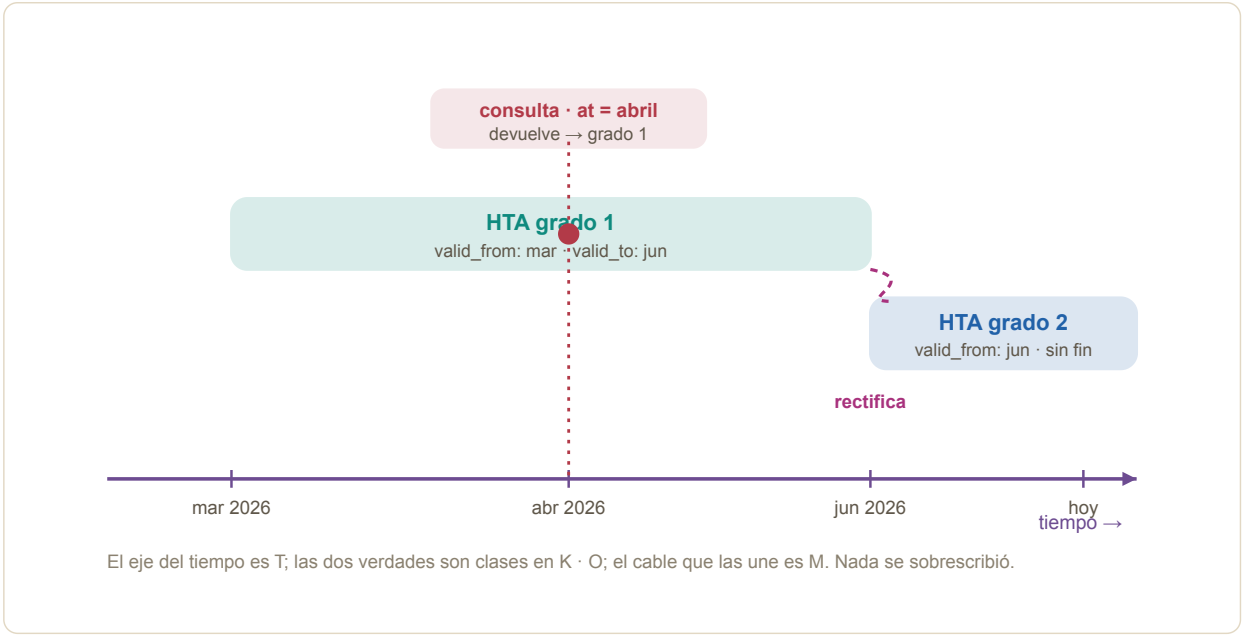
La regla de vigencia entra a salvarnos. No se sobrescribe nada: se guardan **ambos** diagnósticos, cada uno con su rango de validez, y se los conecta para que la cronología sea legible.

> **VIGENCIA · D6**
> Una propiedad que cambia en el tiempo (el grado de una hipertensión) no se edita: se reifica con un rango `inicio` / `fin`. La verdad pasada se conserva intacta. Esa es la regla de vigencia, enunciada al tratar las situaciones; aquí decide si el sistema puede o no responder a un juez.

PYTHON

```
# El diagnóstico viejo: válido desde marzo, cerrado cuando se emite el nuevo
u.assert_fact(diag_marzo, "diagnostico_asignado", hta_grado_1,
              valid_from="2026-03-12", valid_to="2026-06-09")

# El diagnóstico nuevo: vigente desde hoy, sin fin (sigue activo)
u.assert_fact(diag_hta_8842, "diagnostico_asignado", hta_grado_2,
              valid_from="2026-06-09")

# Y se enlazan, para que la cronología no quede a interpretación
u.assert_fact(diag_hta_8842, "rectifica", diag_marzo)   # M · este corrige a aquel
```

El premio de modelarlo así es una capacidad forense que viene de fábrica. Si mañana hay una demanda y el juez pregunta *«¿qué sabía el hospital sobre este paciente en abril?»*, la base de datos no devuelve el estado de hoy: devuelve exactamente la verdad clínica que regía en abril (grado 1), ignorando todo lo que se supo después. Una consulta con un filtro temporal basta.



El eje del tiempo es T; las dos verdades son clases en K · O; el cable que las une es M. Nada se sobrescribió.

Figura 18.2. La vigencia del diagnóstico (D6) sobre la línea de tiempo. El grado 1 fue válido de marzo a junio; el grado 2 lo sustituye desde junio, con su ventana abierta. El cable `rectifica` conecta ambos. Una consulta fechada en abril (la línea vertical) atraviesa solo la barra del grado 1 y devuelve esa verdad —no la de hoy—. La reconstrucción del pasado es una proyección, no un archivo de respaldo.

La prescripción y la tormenta de los porqués

Cuando la doctora ajusta el fármaco, se activan a la vez varias de las fuerzas causales que el modelo distingue. Una receta no es un dato suelto: es una decisión con un motivo, una finalidad y un control de seguridad, y el modelo guarda cada uno en su propio cable en lugar de fundirlos en un comentario.

PYTHON

```
pres = ingest_situation(u, lex, "prescribir",
    roles={
        "agente":              dra_navarro,
        "paciente":            vega,
        "medicamento_prescrito": antihipertensivo,   # O → K
        "dosis":               "20 mg",
        "frecuencia":          cada_manana,
    },
    sit_id="prescrip_8842",
)

u.assert_fact(pres, "motivado_por",      diag_hta_8842)        # ← por qué se receta
u.assert_fact(pres, "con_finalidad",     meta_presion_objetivo) # ← para qué se receta
u.assert_fact(pres, "verificado_contra", regla_renal_alerta)    # ← qué ley clínica la valida
```

Tres respuestas profundas a tres preguntas distintas. La doctora no receta por gusto: lo hace *motivada* por el diagnóstico, *con la finalidad* de llevar la presión a una meta, y *verificando* contra un manual de farmacología que el ajuste no dañe los riñones de un paciente que ya viene comprometido. Tres cables, tres porqués, ningún campo de texto ambiguo.

CONTRAINDICACIÓN COMO OBJETO, NO COMO CÓDIGO

La regla de seguridad (*«no combinar este fármaco con tal condición renal»*) no es una línea escondida en el código de la aplicación. Es un evento reificado más en el eje `O`, con su clase, su fármaco implicado, su condición disparadora y su orden. Como cualquier norma, tiene autor y fecha de vigencia. Vive en el grafo, a la vista, y por eso es auditable.

TRIPLETAS

```
(regla_renal_alerta) ∈ O
  instancia_de        : contraindicacion_clinica   # K · qué clase de norma es
  medicamento_implicado: antihipertensivo          # K · sobre qué fármaco recae
  condicion_gatillo   : funcion_renal_reducida      # K · qué condición la dispara
  orden_oficial       : revisar_antes_de_recetar    # K · qué exige
  vigente_desde       : 2025-01-01                   # T · desde cuándo rige
  emitida_por         : guia_cardiologia_nacional    # Q · qué autoridad la dicta
```

Con la regla viviendo en el grafo, la base de datos se vuelve inteligente sin volverse rígida. Cuando la doctora presiona «recetar», un motor externo (o una IA) escanea el grafo, encuentra esta norma, comprueba la función renal del paciente y, si hace falta, levanta una alerta antes de que el daño ocurra. El grafo guarda la verdad; la lógica que la aplica vive afuera. Esa separación es deliberada, igual que en el spa: el modelo no *ejecuta* la regla de negocio, la deja *legible* para que quien la ejecute no se equivoque.

El control futuro: cuando el dato aún no es real

La quinta pieza, el control a treinta días, parece trivial y esconde una sutileza importante. Todavía no ocurrió: es un plan. Si el sistema lo guardara como un hecho consumado, contaminaría cualquier informe (contaríamos como «atendida» una cita que no existe). El modelo lo marca como *previsto*, no como *real*, exactamente con la misma maquinaria de modalidad que usamos para el diagnóstico sospechado o para la intención de compra del spa.

TRIPLETAS

```
(control_8842) ∈ O
  instancia_de    : consulta_cardiologia
  paciente        : vega
  agente          : dra_navarro
  inicio_previsto : 2026-07-09          # T · la fecha agendada
  estatus_factual : previsto            # K · ¡aún no es real!
  motivado_por    : diag_hta_8842       # M · se controla esta hipertensión
```

Gracias a ese único rótulo, el sistema responde sin tropezar tanto «¿qué citas tiene Vega agendadas?» (donde el control aparece) como «¿cuántos pacientes atendió la doctora en junio?» (donde no aparece, porque aún no sucedió). Lo previsto y lo real conviven en el mismo grafo sin confundirse.

La historia clínica como serie temporal

Hasta aquí hemos diseccionado *una* consulta. Pero Vega no es una consulta: es un paciente con años de roce con el sistema sanitario. El mismo Vega que entró por urgencias con una fibrilación, el que en marzo recibió su primer diagnóstico de hipertensión, el que hoy vuelve al control de cardiología, y el que (lo veremos enseguida) pasará unos días ingresado. Su identidad es una sola, cosida con el mismo `vega` que ya recorre la tienda y la clínica; lo que cambia es el rosario de situaciones que cuelgan de ella a lo largo del tiempo. Eso es una **historia clínica**: no un documento, sino una serie temporal de episodios.

> **UNA SOLA IDENTIDAD**
> El `vega` que compra en la tienda y el `vega` que se atiende en cardiología son la misma entidad en `Q`. La historia clínica no «posee» al paciente: lo *referencia*. Por eso un mismo identificador puede cruzar módulos (tienda, urgencias, consulta, ingreso) sin duplicarse jamás.

En el modelo, la historia no es una tabla con una fila por visita. Es la proyección de todas las situaciones cuyo `paciente` es `vega`, ordenadas por su eje `T` y filtradas por la vigencia que cada hecho declara. Cada episodio (una urgencia, una analítica, una consulta, un ingreso) es una situación reificada más en `O` que referencia al mismo paciente y se sitúa en el tiempo. Recuperar «la historia de Vega» es barrer el grafo por ese nodo y leer la cinta cronológica.

TRIPLETAS

```
# Cada episodio es una situación que apunta al MISMO paciente, situada en T
(urgencias_2403, paciente, vega)   # T: 2026-02-14 · la fibrilación
(analitica_2511, paciente, vega)   # T: 2026-03-05 · sangre y lípidos
(consulta_3019, paciente, vega)    # T: 2026-03-12 · 1er dx de HTA grado 1
(consulta_8842, paciente, vega)    # T: 2026-06-09 · el control de hoy
(ingreso_9120,  paciente, vega)    # T: 2026-06-14 · el ingreso (más abajo)

# La historia = proyección por paciente, ordenada por T, con la vigencia de cada hecho
historia = u.timeline(vega, valid_at="2026-06-19")
```

La clave es que esa cinta es **bitemporal**: no solo registra *cuándo* ocurrió cada episodio (el tiempo del eje `T`), sino *desde cuándo y hasta cuándo se tuvo por cierta* cada afirmación clínica. Es la misma vigencia (D6) que salvó al diagnóstico de pisarse a sí mismo, ahora aplicada a la historia entera. Por eso podemos preguntar dos cosas distintas: «¿qué le pasó a Vega en abril?» (un corte por el tiempo del hecho) y «¿qué *creía saber* el hospital sobre Vega en abril?» (un corte por el tiempo del registro). Un expediente de juguete confunde ambas; el grafo las distingue de fábrica.

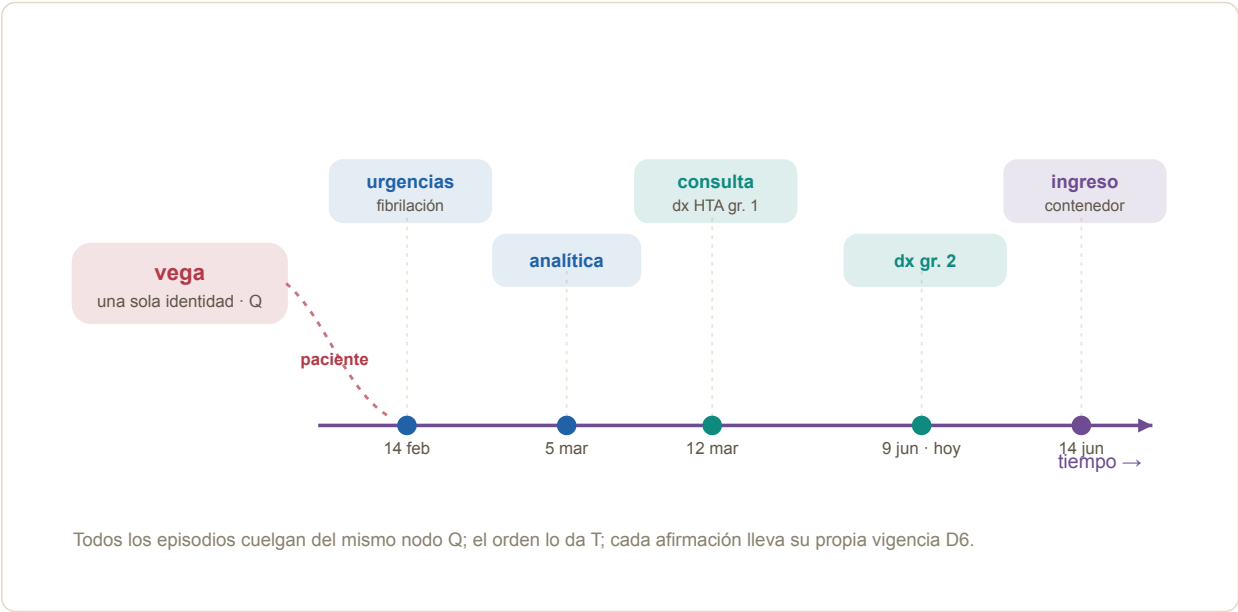


Figura 18.3. La historia clínica de Vega como serie temporal de episodios. La identidad del paciente (`vega`, en `Q`) es una sola; cada urgencia, analítica, consulta o ingreso es una situación independiente en `0` que la referencia y se ordena por su eje `T`. La bitemporalidad (D6) permite leer la cinta «como ocurrió» o «como se creía» en cualquier fecha pasada.

La hospitalización: una situación-marco que contiene eventos

El control de hoy detectó algo que no tranquiliza: la presión no cede y el paciente refiere mareos. La doctora decide ingresarlo unos días para vigilancia. Y aquí aparece una estructura que la consulta ya insinuó pero que el ingreso lleva al extremo: una **situación-marco que contiene a otras**. Si la consulta era una carpeta de cinco piezas, la hospitalización es una *carpeta de carpetas*, una caja con paredes de tiempo dentro de la cual ocurren decenas de eventos menudos (admisión, asignación de cama, signos vitales cada turno, rondas médicas, cada dosis administrada, la evolución diaria) hasta que se cierra con el alta.

El instinto erróneo sería crear una tabla `hospitalizaciones` con columnas para «cama», «fecha de alta» y un campo de texto «notas». El modelo no necesita nada de eso. El ingreso es una situación más en `0`, con su propio identificador fresco, y todo lo que ocurre dentro le cuelga con el mismo cable de siempre: `parte_de`. La hospitalización no es un registro: es un *contenedor*.

TRIPLETAS

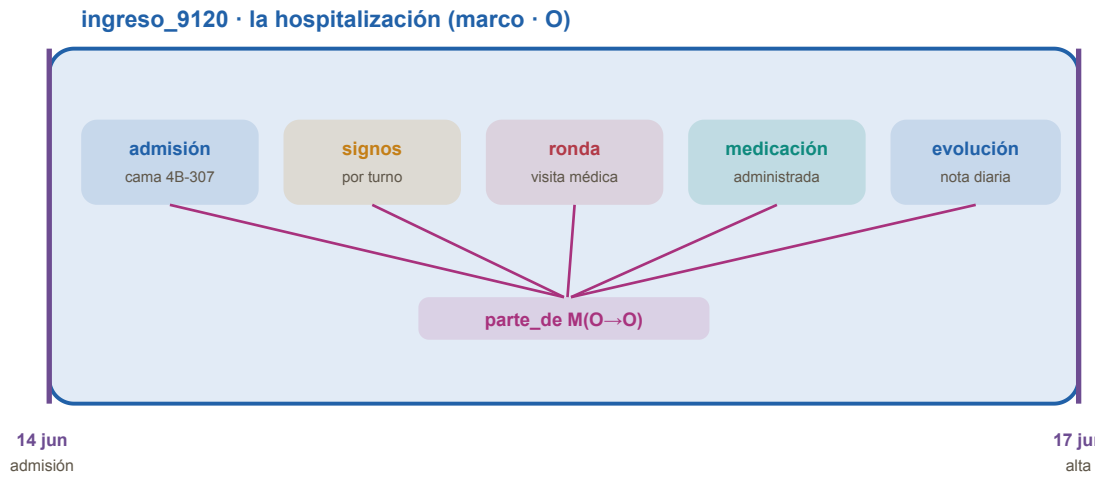
```
(ingreso_9120) ∈ O
  instancia_de : hospitalizacion          # K · qué clase de marco es
  paciente     : vega                      # Q · a quién se ingresa
  agente       : dra_navarro               # Q · quién lo indica
  servicio     : cardiologia               # K · bajo qué servicio
  cama         : cama_4b_307               # O · la cama asignada (objeto físico)
  inicio       : 2026-06-14T11:20-05:00    # T · cuándo se admite
  estatus      : en_curso                  # K · admitido | en_curso | de_alta

# Todo lo que ocurre adentro cuelga del marco con parte_de
(admision_9120,    ∈ O,  parte_de,  ingreso_9120)   # T: 14 jun 11:20
(signos_9120_t1,   ∈ O,  parte_de,  ingreso_9120)   # T: 14 jun, ronda mañana
(ronda_9120_d1,    ∈ O,  parte_de,  ingreso_9120)   # T: 14 jun, visita médica
(adm_farmaco_9120, ∈ O,  parte_de,  ingreso_9120)   # T: 14 jun, dosis administrada
(evolucion_9120_d2,∈ O,  parte_de,  ingreso_9120)   # T: 15 jun, nota de evolución
(alta_9120,        ∈ O,  parte_de,  ingreso_9120)   # T: 17 jun, cierre del marco
```

Fíjate en la cama: `cama_4b_307` es un objeto físico en `O` (tiene ubicación, ocupación, mantenimiento), no una etiqueta. Que el ingreso la referencie por `L` / `O` y no la copie como texto es lo que permite responder «¿qué camas están libres en cardiología ahora mismo?» sin inventar otra tabla. El servicio (`cardiologia`), en cambio, es una clase en `K`: no se interroga su identidad, se usa como categoría.

EL MARCO TAMBIÉN CADUCA · D6

El ingreso tiene `inicio` pero su `fin` queda abierto hasta el alta. Mientras está `en_curso`, su ventana de vigencia no se cierra; el alta no borra el ingreso, le pone `fin`. Así, «¿quién estaba internado el 15 de junio?» es de nuevo una proyección temporal, no un volcado de la tabla de hoy.

El premio de modelar el ingreso como marco es que cada evento interior conserva su contexto sin repetirlo. La dosis administrada el 14 de junio no necesita arrastrar «paciente Vega, servicio cardiología, cama 307»: todo eso se hereda subiendo por `parte_de` hasta el marco. La trazabilidad es estructural, no copiada.



Las paredes son el eje T (admisión → alta); el interior son eventos en O; el cable que los cuelga del marco es M.

Figura 18.4. La hospitalización como situación-marco. Las paredes verticales son el eje `T` (la admisión abre la caja, el alta la cierra); dentro ocurren, en orden, la asignación de cama, los signos vitales, las rondas, la medicación administrada y las notas de evolución. Todos cuelgan del marco con `parte_de` y heredan su contexto sin repetirlo. Un contenedor, no un registro.

El enlace con la farmacia interna: prescripción, dispensación, administración

Hay un eslabón que el software hospitalario de juguete casi siempre rompe: lo que la doctora *receta* no es lo que el paciente *recibe*. Entre la prescripción y el cuerpo de Vega hay una cadena de custodia que cruza el módulo clínico con el de farmacia, exactamente como en un ERP el módulo de ventas conversa con el de inventario. Tres situaciones distintas, encadenadas, cada una con su autor y su momento: la **prescripción** (la doctora ordena), la **dispensación** (la farmacia entrega un lote concreto) y la **administración** (la enfermería lo aplica al paciente).

El detalle que lo cambia todo es el *lote*. La farmacia no entrega «el antihipertensivo» (un concepto en `K`); entrega una caja física de un lote concreto, con su vencimiento y su trazabilidad, igual que en minería un saco de mineral arrastra su lote de origen. Si mañana ese lote se retira por un defecto de fabricación, el grafo debe poder responder en segundos: «¿a qué pacientes se les administró algo de este lote?». Esa pregunta solo tiene respuesta si el lote es un objeto en `O`, no un texto en un comentario.

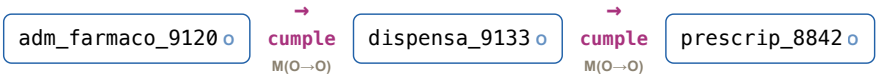
TRIPLETAS

```
# El lote es un OBJETO físico en O, con su vencimiento y su clase de fármaco
(lote_av2207) ∈ O
  instancia_de : lote_farmaceutico        # K · qué clase de objeto es
  es_de        : antihipertensivo         # K · de qué fármaco (concepto)
  vence        : 2027-09-30                # T · caducidad del lote
  custodia     : farmacia_central         # Q · quién lo guarda

# La cadena de custodia: tres situaciones encadenadas, cada una con su momento
(prescrip_8842) ∈ O    # ya existía: la doctora ordena (consulta de hoy)
(dispensa_9133) ∈ O
  instancia_de : dispensacion             # K
  cumple       : prescrip_8842            # M · qué orden satisface
  agente       : farmacia_central         # Q · quién dispensa
  lote_entregado: lote_av2207             # O · qué lote concreto sale
  cantidad     : 14                        # N · unidades entregadas
  inicio       : 2026-06-14T12:05-05:00    # T
(adm_farmaco_9120) ∈ O
  instancia_de : administracion_farmaco   # K
  cumple       : dispensa_9133            # M · qué dispensación consume
  agente       : enf_rojas                # Q · quién la aplica
  paciente     : vega                     # Q · a quién
  lote_usado   : lote_av2207              # O · trazabilidad del lote hasta el cuerpo
  parte_de     : ingreso_9120             # M · ocurre dentro de la hospitalización
  inicio       : 2026-06-14T20:00-05:00    # T
```

La cadena se lee de un tirón siguiendo los cables `cumple`: la administración consume una dispensación, que satisface una prescripción. No hay tablas que cruzar ni claves foráneas que adivinar; la trazabilidad es el propio tejido del grafo. Y como cada eslabón referencia el mismo `lote_av2207`, la pregunta forense del retiro de lote se contesta con una sola proyección.

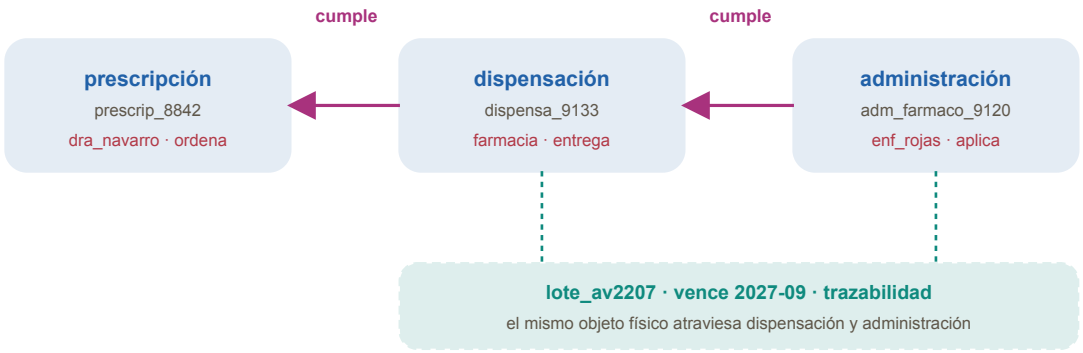
PYTHON

```
# Retiro de lote: ¿a qué pacientes llegó algo de lote_av2207?
afectados = u.subjects_where("lote_usado", lote_av2207, role="paciente")
# → [vega, ...]   en segundos, cruzando clínica y farmacia sin un solo JOIN
```



CROSS-MÓDULO SIN PEGAMENTO

Clínica y farmacia son «módulos» distintos solo en la cabeza del organigrama. En el grafo son el mismo tejido: la prescripción vive en el módulo clínico, el lote en el de farmacia, y el cable `cumple` los une sin que nadie escriba código de integración. Es la misma lección que dará el capítulo del ERP, anticipada aquí: la trazabilidad del lote es a la farmacia lo que la del mineral será a la minera.



Tres situaciones en O encadenadas por M (cumple); el lote, objeto en O clasificado por K, las cruza para la trazabilidad.

Figura 18.5. La cadena de la farmacia interna. La administración de la enfermería *cumple* una dispensación de la farmacia, que a su vez *cumple* la prescripción de la doctora (tres situaciones en `O` encadenadas por `cumple`). El lote concreto (`lote_av2207`) atraviesa la dispensación y la administración, de modo que un retiro de lote se resuelve con una sola proyección, sin cruzar tablas entre el módulo clínico y el de farmacia.

Del paciente a la población

Todo este capítulo diseccionó a un paciente. Vega tiene nombre, una fibrilación en febrero, un diagnóstico que cambió de grado en marzo. Pero quien dirige el hospital rara vez pregunta por Vega: pregunta por la población que Vega integra. Cuántos pacientes arrastran hoy una hipertensión grado 2, qué diagnóstico encabezó el trimestre, cuántas camas sigue ocupando cardiología esta mañana.

Y ahí está el dividendo que no se ve mientras modelas: ninguna de esas preguntas pide una tabla nueva. El diagnóstico de Vega es una situación más, del mismo tipo que los de los otros mil pacientes, y contar los de un tipo es contarlos a todos a la vez. Diseccionar con cuidado una sola consulta fue, sin proponérnoslo, dejar armado el tablero epidemiológico.

PYTHON

```
# Cuántos diagnósticos de hipertensión grado 2 hay vigentes — la cohorte que Vega integra
count(u, Pattern(fixed={"diagnostico_asignado": u.ind("hta_grado_2")},
                 type_constraint=u.ind("diagnostico")))
```

El lexicon de la clínica

Para que la doctora (o el sistema que ella usa) no tenga que aprender jerga de bases de datos, configuramos el lexicon con las palabras exactas de la medicina. El lexicon es el compilador del que habló la Parte IV: traduce el vocabulario humano a los roles canónicos y resuelve la polisemia. El verbo *«controlar»*, por ejemplo, significa cosas distintas según lo que lo acompañe.

PYTHON

```
lex.register(LexiconEntry(
    verb="controlar",
    situation_type="consulta_cardiologia",
    obligatory=["agente", "paciente"],
    pattern=("a_un_paciente",),        # "controlar a un paciente" → seguimiento clínico
))

lex.register(LexiconEntry(
    verb="controlar",
    situation_type="medicion_signo_vital",
    obligatory=["agente", "medida_evaluada"],
    pattern=("un_signo",),             # "controlar la presión" → una medición
))
```

Y le instalamos el dialecto del dominio: un mapa entre las palabras que el personal clínico usa de verdad y las etiquetas canónicas del catálogo. El usuario final nunca toca esas etiquetas internas (el sistema traduce por debajo, exactamente como el spa traducía «IGV» a su clase de impuesto).

PYTHON

```
lex.register_domain_dialect("cardiologia_central", {
    "presión":        "presion_arterial",
    "el paciente":    "paciente",
    "receta":         "prescripcion_medica",
    "control":        "consulta_cardiologia",
    "alergia":        "contraindicacion_clinica",
    "la pastilla":    "antihipertensivo",
})
```

Gracias a esto, la doctora puede dictarle al sistema *«controlé al paciente, la presión salió alta, le subí la pastilla y lo cito en un mes»* y el grafo lo traduce a sus identificadores internos sin que ella sepa que existen. La interfaz es su idioma, no el catálogo (tal como dejó establecido la Parte IV).

El antes y el después: del expediente fragmentado al grafo único

Antes, en SQL. En un sistema hospitalario típico, el expediente de Vega vive repartido en cinco o seis tablas que no se hablan solas: `pacientes`, `consultas`, `diagnosticos`, `prescripciones`, `contraindicaciones` y (si el equipo fue cuidadoso) una tabla `diagnostico_historial` para versionar los cambios. Cuando la doctora quiere saber si el ajuste que acaba de recetar choca con una contraindicación *vigente*, el sistema debe cruzar `prescripciones` con `contraindicaciones`; y para saber si la hipertensión sigue activa, otro `JOIN` contra `diagnostico_historial` filtrando por fechas. La respuesta no está en ninguna tabla sola: vive en el pegamento que el programador escribe y reescribe cada vez que la lógica clínica cambia.

SQL

```
-- Antes: el diagnóstico vigente vive en otra tabla, versionado por fechas;
-- la pregunta clínica obliga a encadenar JOINs y filtrar por rango temporal.
CREATE TABLE diagnosticos          (id INTEGER PRIMARY KEY, paciente_id INT, codigo TEXT);
CREATE TABLE diagnostico_historial (id INTEGER PRIMARY KEY, diag_id INT,
                                    grado TEXT, valido_desde DATE, valido_hasta DATE);
CREATE TABLE prescripciones        (id INTEGER PRIMARY KEY, paciente_id INT, farmaco_id INT);

SELECT h.grado
```

```
FROM diagnostico_historial h
JOIN diagnosticos d ON d.id = h.diag_id
WHERE d.paciente_id = :paciente
  AND h.valido_desde <= :fecha            -- la lógica temporal,
  AND (h.valido_hasta IS NULL OR h.valido_hasta > :fecha);  -- a mano, cada vez
```

Después, en WQuestions. La misma información es un grafo único de hechos atómicos: `diag_hta_8842` cuelga de `consulta_8842` con `parte_de`, lleva sus propias fechas de validez internas, y `prescrip_8842` apunta a `regla_renal_alerta` con `verificado_contra`. La pregunta (¿esta prescripción choca con una contraindicación vigente?) es una proyección sobre el grafo con un filtro temporal: el motor entrega solo los hechos cuya ventana de validez cubre la fecha consultada. No hay `JOIN` entre tablas ni código de pegamento; la vigencia y las relaciones causales ya estaban tejidas en el mismo tejido de datos desde el primer hecho.

PYTHON

```
# Después: el diagnóstico vigente a una fecha es una proyección con filtro temporal.
vigente = u.value_at(vega, "diagnostico_asignado", at="2026-04-15")  # → hta_grado_1
```

> *En medicina, un diagnóstico no es un campo de texto: es una creencia fechada, con su evidencia detrás y su caducidad delante. Tratarlo como una entidad, y no como una columna que se sobrescribe, es lo que convierte un expediente en algo que un juez puede leer.*
>
> LA LECCIÓN DE LA CLÍNICA

Cuánto pesó el dominio entero

Vale la pena medir la huella, porque desmiente el prejuicio de que tanta expresividad debe costar un diluvio de datos. Toda la consulta de Vega (las cinco piezas, los tres porqués de la receta, los dos diagnósticos versionados, la contraindicación y el control previsto) se guardó como un puñado modesto de hechos atómicos. Comparado con lo que un esquema relacional necesita para sostener la misma riqueza histórica, la diferencia es notable.

Figura 18.6. Huella de la consulta por bloque, en hechos atómicos. La pieza más cara es el diagnóstico, justamente porque carga la vigencia (D6): guardar dos verdades históricas en vez de pisar una sola cuesta unos hechos de más —y vale cada uno—. El total ronda las seis decenas de tripletas: un volumen ridículo frente a la maraña de tablas, historiales y procedimientos que el mismo caso exige en SQL.

La pepita metodológica: el arte de saber qué reificar

Llegamos a la tercera pregunta de la elicitación, la que dejamos pendiente al abrir el capítulo, porque es la que de verdad separa a quien modela bien. A la hora de diseñar un dominio nuevo, ¿cómo decides qué dato merece convertirse en una entidad con identidad propia en `O` y qué dato debe quedarse como un número callado en `N` o una etiqueta inerte en `K`?

LA REGLA, EN UNA LÍNEA

Si crees que en el futuro **alguien va a hacer una pregunta o una afirmación sobre ese dato**, reifícalo. Si nadie va a interrogarlo nunca, déjalo como valor.

Es una regla simple y brutal, y se aplica caso por caso. Mira cómo decide cada pieza de esta consulta su propio destino:

¿Reificamos el síntoma? Sí. Mañana el médico querrá saber cuándo empezó, qué tan intenso era y si mejoró. Se vuelve entidad en `0`.
¿Reificamos la medición de presión? Sí. Mañana la conectaremos como evidencia de un diagnóstico. Entidad en `0`.
¿Reificamos la contraindicación? Sí. Tiene autor, fecha de vigencia y condición disparadora; alguien la consultará. Entidad en `0`.

¿Reificamos «cada mañana»? No. Es un valor terminal. Nadie va a interrogar al concepto «cada mañana». Se queda como etiqueta en `K`.
¿Reificamos «20 mg»? No. Es una cantidad con su unidad. Vive en `N` y ahí muere.
¿Reificamos «148/95»? El número, no; pero la *medición* que lo contiene, sí (porque la medición sí será interrogada).

El costo de reificar es de unos pocos bytes en el servidor. El costo de *no* hacerlo (y descubrir, años después, que un regulador exige el historial completo de un dato que guardaste como texto plano) se mide en una refactorización carísima y, en medicina, en algo peor que dinero. Por eso, ante la duda, esta regla se inclina siempre hacia reificar.

Balance: un dominio nuevo, casi sin fricción

Cerremos repasando qué demostró el prototipo al absorber, desde cero, un dominio para el que la arquitectura no había sido diseñada.

EL VEREDICTO DE LA CLÍNICA

La carpeta maestra funcionó. La consulta abrazó sus cinco sub-situaciones con `parte_de`, igual que la venta abrazó sus líneas. Ni una tabla paralela, ni un esquema nuevo.

La única fricción fue sana. El error de firma al mezclar `0` con `K` no fue un fallo del modelo: fue el modelo corrigiendo al modelador. Se resolvió declarando un puñado de cables de dominio.

El diagnóstico se modeló completo. Su duda (modalidad), su evidencia (`motivado_por`) y su caducidad (D6) se guardaron en dimensiones separadas, sin aplastarse en un campo de texto.

La historia se preserva (D6). Dos diagnósticos contrarios conviven sin que se pierda el rastro forense. El sistema puede responder qué se sabía en cualquier fecha pasada.

El método quedó explícito. Evento ancla, piezas que nacen dentro, y la regla de qué reificar. Esa receta de elicitación sirve para el próximo dominio, sea cual sea.

La clínica parecía exigir una arquitectura especializada y resultó caber en la misma que ya conocíamos, con un par de cables propios y ni una sola excepción estructural. Pero hubo una exigencia que aquí solo asomó (la de que varios custodios distintos, en hospitales distintos, puedan leer y escribir sobre el mismo paciente sin pisarse) y que en el próximo capítulo se vuelve la ley. Subimos al dominio más implacable de todos: un banco, donde no borrar el pasado deja de ser buen gusto y pasa a ser una condición que dicta el regulador.

19

El dominio más exigente: un banco

En casi todos los dominios, las decisiones de diseño son cuestión de buen gusto. En un banco son cuestión de ley. Aquí veremos si el modelo soporta el peso del dinero sin doblarse.

Son las 02:13 de un martes. En un servidor de un banco regional, un proceso de cierre acaba de detenerse: dos cifras que llevaban diez años coincidiendo dejaron de coincidir por catorce centavos. No es un error de cálculo; es un error de *historia*. Alguien, en algún sistema satélite, sobrescribió un dato que un auditor necesitaba leer tal como estaba la semana pasada, y ahora ese pasado ya no existe. Multiplica esos catorce centavos por millones de operaciones y por la firma de un regulador que exige reconstruir cualquier instante, y tendrás el motivo por el cual el dominio bancario es la prueba de fuego de cualquier modelo de datos. No perdona la elegancia que no resiste una citación judicial.

Conviene fijar la escala. Una empresa mediana cualquiera guarda su operación en unas ciento cincuenta tablas, un par de sistemas heredados y un programa de contabilidad. Un banco regional de tamaño medio juega en otra liga: alrededor de mil quinientas tablas solo para el núcleo, otro millar para tarjetas de crédito, otro tanto para seguros, y una nube de aplicaciones satélite para microcréditos, prevención de fraude y la aplicación móvil. Cada sistema tiene su propio dueño, sus propias reglas y, lo más corrosivo, su propio idioma.

En el capítulo del spa demostramos que el modelo era ágil; en el del taxi, que aguantaba la velocidad; en la historia clínica, que sabía repartir la verdad entre varios custodios. El banco reúne las tres exigencias y añade una cuarta que las domina a todas: **el regulador**. Aquí las decisiones de diseño que en otros capítulos eran preferencias (no borrar nunca el pasado, tratar al software como un agente con responsabilidad) dejan de ser lujos teóricos y se vuelven condiciones de supervivencia dictadas por la ley.

> **ALCANCE**
> No vamos a modelar un banco entero (eso pediría otro libro). Tomaremos cuatro situaciones críticas y mostraremos que cada una cae, sin forcejeo, sobre la misma maquinaria que ya conoces.

Este capítulo no aspira a ser un manual de *core banking*. Aspira a algo más modesto y más contundente: tomar cuatro escenas (una transferencia, la vida entera de un préstamo, una investigación de fraude y el catálogo de productos) y demostrar que el modelo absorbe la complejidad industrial sin trucos. Si aguanta el dinero, aguanta cualquier cosa.

La realidad honesta: cinco islas que no se hablan

Antes de presumir de soluciones, describamos el enfermo. Quien trabaja en banca asentirá; quien no, se asombrará. Por dentro, un banco no es un sistema: es un archipiélago de cinco islas que apenas se saludan.

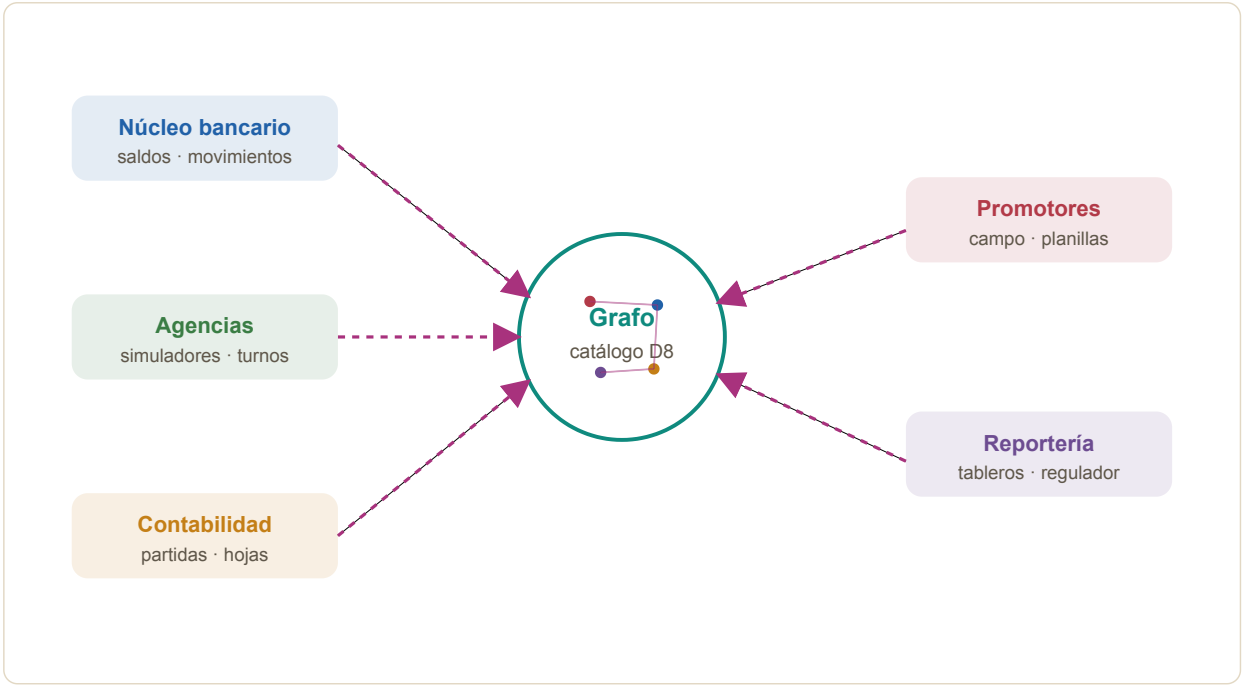


Figura 19.1. Las cinco islas de un banco regional (núcleo, agencias, contabilidad, promotores y reportería), cada una con su tecnología y su jerga. WQuestions no reemplaza a ninguna: cada isla *publica* sus hechos al grafo común con su dialecto, mapeado una sola vez al catálogo D8. El archipiélago se vuelve consultable sin disolverse.

Isla 1 · El núcleo. El servidor central, el *core bancario*. Allí viven los saldos, los movimientos oficiales y la contabilidad pesada. Es la «verdad» que el Estado audita.

Isla 2 · Las agencias. Cada sucursal corre pequeños programas para simular créditos o repartir turnos. Toman decisiones todos los días, pero rara vez le explican al núcleo *cómo* llegaron a ellas.

Isla 3 · Contabilidad. El zoológico oculto. Una parte vive en el servidor; otra, enorme, vive en hojas de cálculo que se ajustan a mano a fin de mes. Cuando el núcleo y la contabilidad discrepan, suele ganar contabilidad —y casi nadie sabe explicar por qué.

Isla 4 · Los promotores. Los agentes de calle que venden créditos anotan datos en planillas y papeles. Esa información valiosa del cliente es «fantasma» hasta el día en que el crédito se aprueba y recién entonces ingresa al banco.

Isla 5 · La reportería. Cuando la gerencia pide un reporte que cruce las cuatro islas anteriores, el equipo técnico tarda semanas en cavar un túnel temporal que junte todo en un *data lake*. A veces los números no cuadran y el proyecto naufraga.

¿Por qué ocurre? Porque cuando un banco intenta modernizarse, comete el mismo error: obligar a todas las islas a hablar un único idioma robótico. Y reprogramar un sistema de quince años para que hable «moderno» sale tan caro que los bancos prefieren dejarlo morir en paz. El resultado es un empate técnico permanente: nadie migra, nadie se entiende.

EL GIRO DE WQUESTIONS

El modelo no le exige a la sucursal ni a contabilidad que cambien su software. Solo les pide que **traduzcan** lo que ya hacen y lo publiquen al mapa central (el grafo) usando su propia jerga. Contabilidad habla de «partidas»; la agencia habla de «simulaciones». El lexicon traduce ambos dialectos al catálogo universal una sola vez, y a partir de allí cada isla sigue operando como siempre mientras la gerencia consulta el mapa en milisegundos.

El banco sobre las siete coordenadas

Para domar el caos, repartamos las piezas del banco en los ejes de valor. El ejercicio es revelador por sí solo: lo que parecía una maraña de mil quinientas tablas se ordena en seis preguntas.

Q QUIÉN · AGENTES

Clientes físicos, empresas (personas jurídicas) y —decisivo— los **sistemas**. El motor antifraude y el autorizador de la red de tarjetas deciden solos; por lo tanto, viven en `Q` como agentes activos con responsabilidad.

O QUÉ · ENTIDADES Y EVENTOS

Cuentas, préstamos, movimientos de dinero, tarjetas, asientos contables e investigaciones de fraude. Aquí se concentra el noventa por ciento del peso del banco.

L DÓNDE · LUGARES

Sucursales físicas, cajeros automáticos y lugares virtuales como la aplicación móvil o la banca web. El canal importa: una operación por cajero no se gobierna igual que una por app.

T CUÁNDO · TIEMPOS

Cuidado aquí: hay **triple reloj**. La hora en que el cliente ordenó el pago, la hora en que el servidor lo procesó y la fecha del cierre contable. El modelo anota los tres sin confundirlos.

N CUÁNTO · MAGNITUDES

Dinero, tasas de interés, plazos en meses y puntajes de riesgo. Cada magnitud arrastra su unidad: ningún número anda suelto.

K CUÁL · CLASES

Tipos de cuenta, monedas (dólar, euro, sol), estados de mora y códigos legales. El zócalo categórico que da sentido a todo lo demás.

Los predicados del eje *cómo* (el séptimo eje, **M**) son los cables que conectan todo: `agente`, `parte_de`, `cubierto_por`, `cancela`. Sin ellos las seis cajas serían seis listas inertes. Con ellos, un hecho bancario se vuelve un punto navegable. Pasemos a verlo funcionar.

Caso 1 · Una transferencia y sus cinco agentes ocultos

Para un humano, una transferencia es una frase de seis palabras: «Mariana le envía 480 dólares a Bruno». Para el sistema de un banco, ese gesto moviliza a cinco agentes y deja dos registros espejo en contabilidad. Veamos cómo lo despliega el modelo.

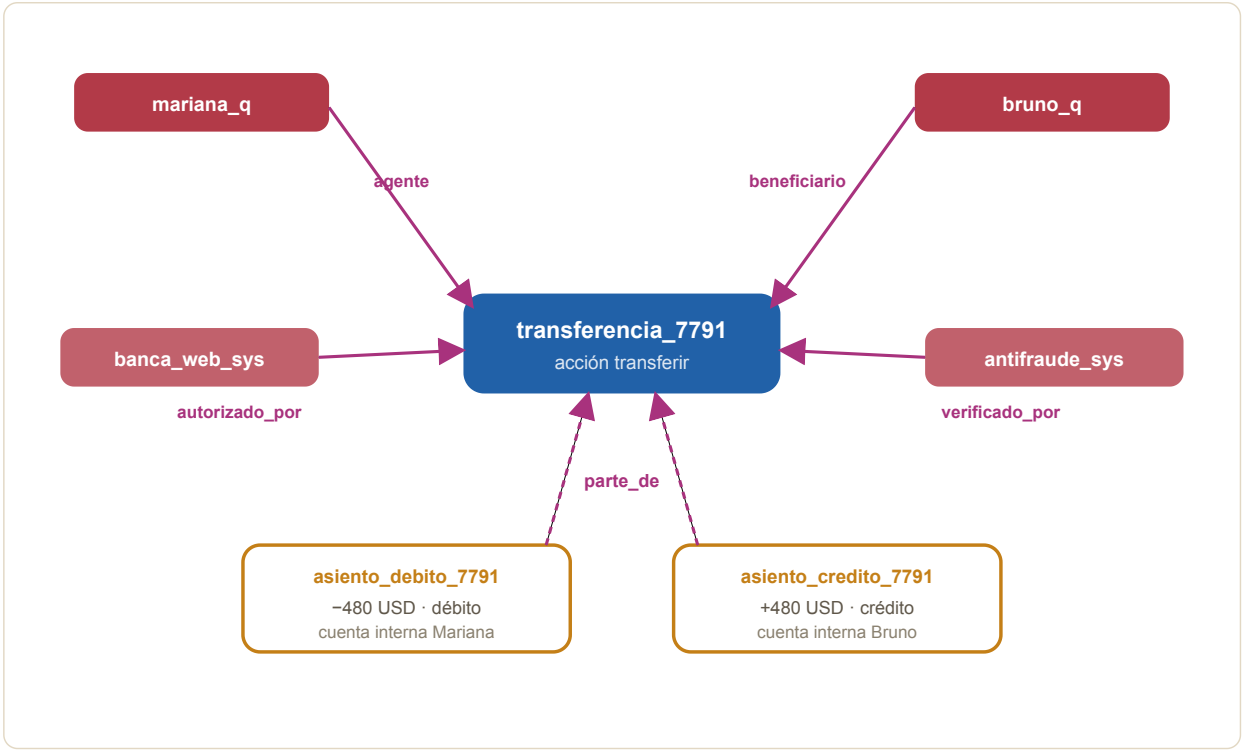


Figura 19.2. La transferencia `transferencia_7791` como evento central. Cuatro agentes la rodean (quien inicia, quien recibe, el sistema que autoriza y el motor que verifica) y dos asientos contables cuelgan de ella por `parte_de`; el departamento de contabilidad es el quinto agente, implícito en las partidas. Nada de esto cabe en una sola fila de una tabla.

Primero declaramos el evento central. Observa que tres de sus enlaces apuntan a agentes y que uno de ellos (el motor antifraude) no es una persona:

TRIPLETAS

```
(transferencia_7791) ∈ 0
  instancia_de:     accion_transferir
  agente:           mariana_q            # Agente 1 — inicia
  beneficiario:     bruno_q              # Agente 2 — recibe
  cuenta_origen:    cta_mariana_4410
  cuenta_destino:   cta_bruno_8820
  monto:            n_480_usd
  lugar_de:         app_movil_banco
  autorizado_por:   banca_web_sys        # Agente 3 — da el visto bueno
  verificado_por:   antifraude_sys       # Agente 4 — descarta el robo
  ts_orden:         2026-05-19T21:07:44  # reloj del cliente
  ts_proceso:       2026-05-19T21:07:46  # reloj del servidor
```

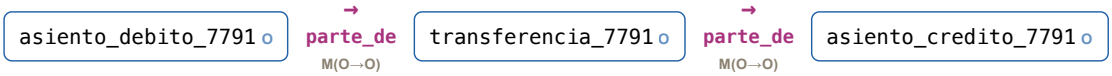
Pero la transferencia no termina ahí. Tras bambalinas, contabilidad (el quinto agente) debe cuadrar sus libros. La partida doble es una ley de la profesión: a todo débito le corresponde un crédito. En el modelo creamos dos **subeventos** y los colgamos de la transferencia con el cable `parte_de`:

TRIPLETAS

```
(asiento_debito_7791) ∈ 0              # le retiramos saldo a Mariana
  parte_de:         transferencia_7791
  cuenta_contable:  ahorros_mariana_interna
  monto:            n_480_usd
  tipo_movimiento:  debito

(asiento_credito_7791) ∈ 0             # le abonamos saldo a Bruno
  parte_de:         transferencia_7791
```

```
  cuenta_contable:  ahorros_bruno_interna
  monto:            n_480_usd
  tipo_movimiento:  credito
```

La misma escena, vista como tres hechos atómicos, queda así de explícita: el evento y sus dos contrapartidas, unidos por un solo predicado.



D5 EL SOFTWARE ES UN AGENTE

El motor antifraude no es un campo ni una bandera `bool`: es un agente del eje `Q` con responsabilidad sobre el hecho. Cuando `antifraude_sys` deja pasar una operación, esa decisión queda firmada con su nombre, igual que la firma de un cajero humano. Si mañana hay que rendir cuentas de por qué pasó, el responsable está en el grafo, no perdido en un log que nadie guardó.

Y si esta operación resultara mal hecha, **jamás la borramos**. Creamos una transferencia rectificativa que apunta a la vieja y la anula. La ley obliga a los bancos a conservar el historial completo de los errores, no solo el resultado pulido; el modelo lo hace por defecto, sin tablas de auditoría aparte. Volveremos sobre esto en el Caso 3.

Queda una última arruga de la transferencia, y no se deshace con agentes ni con asientos, sino con el reloj. La de Mariana a Bruno se liquidó en dos segundos porque era interna: el mismo banco movía dinero entre dos de sus cuentas. Invierte ahora los papeles. A Mariana le envían un giro desde **otro banco** y, peor aún, desde **el extranjero**, un viernes por la tarde.

FECHA VALOR

La hora en que ordenan el giro (viernes) y la fecha en que el dinero ya es de Mariana (lunes) no son la misma. A los tres relojes del eje T el comercio exterior le añade este matiz (la *fecha valor*); confundirlo es lo que produce el clásico «ya te lo envié / a mí no me ha llegado», donde las dos personas creen tener razón.

El dinero ya salió de la cuenta del remitente, pero a la de Mariana todavía no entra: queda suspendido en una cámara de compensación, esperando a que los sistemas abran. Y como el viernes la ventana ya cerró, tendrá que esperar hasta el **lunes** para disponer de su propio dinero. En pleno siglo XXI, un par de días de limbo para un giro que la pantalla ya da por enviado, y el dinero podía ser urgente.

Que esa espera exista no es culpa de ningún modelo de datos: responde a ventanas de compensación, husos horarios, fines de semana y reglas regulatorias que ningún grafo borra. Pero sí es culpa del modelo plano que Mariana no pueda saber *dónde* está su dinero mientras tanto; un sistema clásico le muestra un opaco «pendiente» y poco más. WQuestions no trata la transferencia como un instante, sino como un evento con estados fechados (la misma vigencia temporal que el siguiente caso convertirá en regla):

TRIPLETAS

```
(giro_int_8043, estado, ordenada,        inicio=2026-05-15T17:40, fin=2026-05-15T17:41)
(giro_int_8043, estado, en_compensacion, inicio=2026-05-15T17:41, fin=2026-05-18T09:02)
(giro_int_8043, estado, disponible,      inicio=2026-05-18T09:02, fin=hoy)
(giro_int_8043, fecha_valor, 2026-05-18)           # cuándo el dinero es de Mariana
(giro_int_8043, ts_orden,    2026-05-15T17:40)     # viernes: cuándo se ordenó
```

Modelada así, la pregunta angustiada del cliente («¿dónde está mi dinero y cuándo lo tendré?») deja de ser un secreto que solo el banco conoce: es una consulta que el grafo contesta al instante, con el estado vigente y la fecha en que el dinero se libera. El modelo no acelera el giro; los dos días de espera siguen ahí. Lo que hace es convertir ese limbo opaco en un hecho explícito, fechado y consultable, que es justo lo que el cliente (y el regulador) tienen derecho a ver.

Caso 2 · La novela de un préstamo y la regla del tiempo

Un préstamo no es un suceso de un segundo: es una novela que dura años. Se aprueba, se pagan cuotas, el cliente se atrasa (entra en mora), a veces la deuda se reestructura y, con suerte, todo termina cancelado. En un sistema heredado, la celda «estado» del préstamo se va borrando y reescribiendo, y con cada reescritura desaparece un capítulo de la historia.

Aquí aplicamos la regla de **vigencia temporal**. Cada vez que el préstamo de Mariana cambia de estado, no borramos nada: inyectamos una línea nueva con su fecha de inicio y su fecha de fin. El pasado se acumula, no se pierde.

> **D6 VIGENCIA TEMPORAL: EL PASADO NO SE SOBRESCRIBE**
>
> Ningún hecho que pueda cambiar se guarda como un valor desnudo; se guarda con un intervalo `[inicio, fin)`. Cambiar de estado no es modificar una celda: es *cerrar* el intervalo vigente y *abrir* uno nuevo. La consecuencia es que el grafo conserva, gratis, toda la trayectoria de cualquier atributo a lo largo del tiempo.

TRIPLETAS

```
(prestamo_5503, estado, vigente,         inicio=2026-01-20, fin=2026-08-15)
(prestamo_5503, estado, mora_30_dias,    inicio=2026-08-15, fin=2026-09-15)
(prestamo_5503, estado, mora_60_dias,    inicio=2026-09-15, fin=2026-10-22)
(prestamo_5503, estado, reestructurado,  inicio=2026-10-22, fin=hoy)
```

Cinco años después, en plena audiencia, el juez pregunta: «¿En qué estado exacto se encontraba este préstamo el 1 de septiembre de 2026?». La base filtra por fecha y responde sin titubear: «Mora de treinta días». Sin la regla D6, los bancos tienen que construir costosas tablas paralelas solo para almacenar estos fantasmas del pasado —y rezar para que alguien las haya mantenido al día.

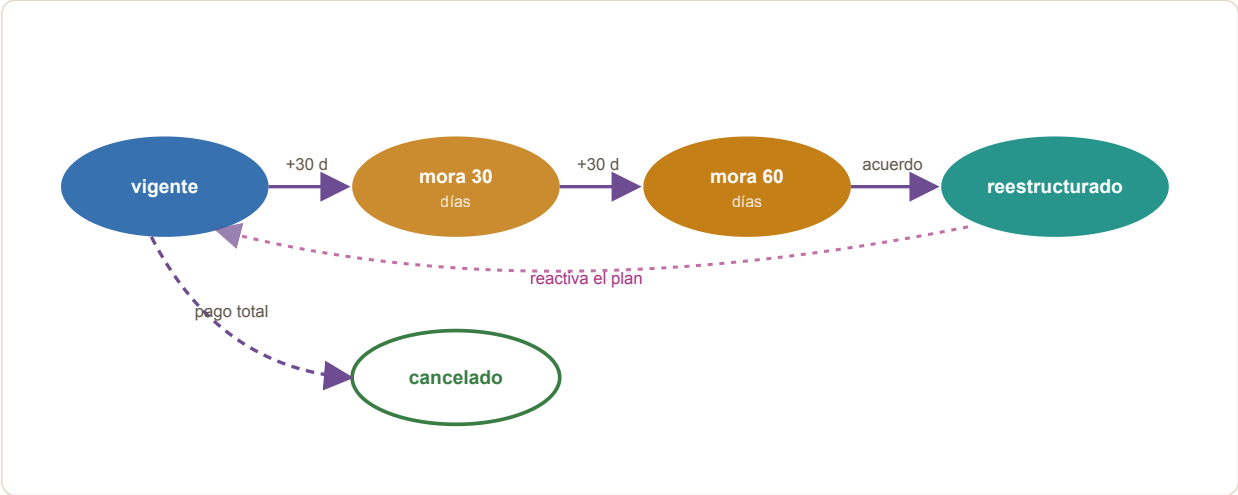


Figura 19.3. El ciclo de vida del préstamo como máquina de estados. Cada transición (flechas en color *tiempo*) es un hecho nuevo con vigencia D6 y su propio `motivado_por`; el rastro entero se preserva. La reestructuración puede devolver el préstamo a estado vigente sin borrar la mora que ocurrió: el pasado sigue ahí, fechado.

Hay un detalle fino que la figura insinúa. Una transición no es solo un cambio de etiqueta: es un hecho que merece su propia causa. Por eso cada cambio de estado lleva un enlace `motivado_por` que apunta al evento que lo provocó (un pago vencido, un acuerdo firmado) y, cuando hace falta, un `justificado_por` que apunta al documento legal. El «por qué» no es un eje, como vimos en su momento; es una relación entre hechos, y aquí esa relación es la que un auditor leerá primero.

Caso 3 · Investigación de fraude: reconstruir la noche del cargo

Mariana llama indignada: alguien usó su tarjeta para pagar 1.840 dólares en una tienda de electrónica al otro lado del país, de madrugada, mientras ella dormía. El banco debe abrir una investigación. Y es justo aquí donde la arquitectura se separa de la competencia por una distancia que no se cierra con más servidores.

El investigador no necesita el saldo de Mariana de *hoy*. Necesita viajar a la noche del cargo y preguntar dos cosas que un modelo plano no sabe responder: ¿dónde creía el banco que estaba Mariana esa noche?, y ¿qué decía su perfil de riesgo a esa hora exacta? La respuesta correcta no es el valor actual del perfil; es el valor que el perfil *tenía entonces*.

> **BITEMPORALIDAD**
> Qué ocurrió y qué sabía el sistema cuando ocurrió son dos ejes de tiempo distintos. El esquema clásico los confunde en uno; la vigencia D6 los mantiene separados. Esa distinción, abstracta en otros capítulos, aquí decide si el banco cobra o no el seguro.

Gracias a la vigencia, el perfil de riesgo de Mariana no es un dato que se pisa: es una serie de versiones fechadas. El investigador recupera la versión que estaba activa la noche del fraude:

TRIPLETAS

```
(perfil_riesgo_mariana_v4, instancia_de, perfil_antifraude,
                           inicio=2026-04-30, fin=2026-06-02)
(perfil_riesgo_mariana_v4, score_riesgo, 0.27)
(perfil_riesgo_mariana_v4, geo_habitual, ciudad_costa)
```

Con eso a la vista, el investigador entiende por qué la tarjeta pasó esa noche: el `score_riesgo` era bajo y el comercio no levantó alertas suficientes. Al confirmar el robo, **no borra** el pago de la tienda (el cargo fue real y movió dinero de verdad). En su lugar crea un evento de reverso que apunta al original y deja escrita la causa:

TRIPLETAS

```
(reverso_cargo_3120, cancela,          autorizacion_original_3120)
(reverso_cargo_3120, justificado_por,  investigacion_fraude_0457)
(reverso_cargo_3120, monto,            n_1840_usd)
(reverso_cargo_3120, ts_proceso,       2026-05-21T09:40:11)
```

El historial queda intacto: el cargo fraudulento sigue allí, fechado, y a su lado el reverso que lo neutraliza. El dinero vuelve y el banco conserva una prueba impecable para presentar al seguro. Esto es lo que un sistema plano no puede ofrecer sin reconstrucciones heroicas: la capacidad de mostrar no solo el resultado, sino la secuencia exacta de lo que se supo y cuándo.

> **LA TRAMPA DEL `UPDATE`**
> En el modelo plano, corregir un fraude tienta a ejecutar un `UPDATE perfiles_riesgo SET nivel='alto'` y un `DELETE` del cargo falso. Es la decisión que destruye el caso: al actualizar el perfil, se borra justamente la prueba de que esa noche el sistema lo creía bajo; al borrar el cargo, se borra la evidencia del robo. El reflejo más natural del programador es, aquí, el peor enemigo del banco. El modelo lo previene haciendo que el `UPDATE` destructivo no sea siquiera una operación disponible.

Caso 4 · Una tarjeta «Platino» no es una idea: es un objeto

Resolvamos, por último, un error de modelado tan común como costoso. Cuando el banco le entrega a Mariana una «Tarjeta Platino», muchos programadores asumen que «Platino» es una categoría abstracta y la guardan en la caja `K`. Es falso, y la falsedad tiene consecuencias millonarias.

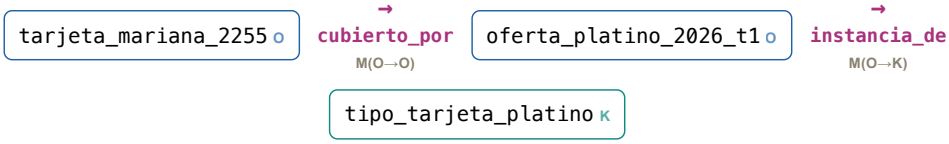
> **PRODUCTO COMO OFERTA REIFICADA**
>
> La «Oferta Platino del primer trimestre de 2026» no es una etiqueta: es un **objeto real y reificado** de la caja `O`, con un contrato anexo, unas tasas, una cuota de manejo y una fecha de caducidad. El tipo *tarjeta platino* sí vive en `K`; la *oferta concreta* que un cliente firmó, no. Confundir la clase con la instancia es confundir la receta con el plato servido.

Cuando el banco entrega el plástico, conecta la tarjeta de Mariana a la oferta específica que estaba vigente ese día —no a la idea de «tarjeta platino», sino al contrato exacto que ella aceptó:

TRIPLETAS

```
(oferta_platino_2026_t1) ∈ O           # el contrato de oferta del banco
  instancia_de:         tipo_tarjeta_platino   # ← su clase en K
  cuota_manejo_mensual: n_9_usd
  tasa_anual:           n_38_pct
  vigencia:             inicio=2026-01-01, fin=2026-06-30

(tarjeta_mariana_2255) ∈ O             # el plástico que tiene Mariana
  cliente:              mariana_q
  cubierto_por:         oferta_platino_2026_t1
```

¿Por qué es vital separar la oferta de la clase? Porque si en julio el banco lanza una versión nueva de la tarjeta con cuota de manejo de doce dólares, la tarjeta de Mariana no puede encarecerse por arte de magia: ella firmó el contrato de la oferta de enero, y ese contrato sigue siendo un objeto congelado en el tiempo, con su propia vigencia. Al tratar los productos financieros como objetos independientes y fechados, el banco se blinda contra demandas multimillonarias por cambiar las reglas sin avisar.



> *En banca, los productos no son adjetivos: son contratos firmados que el tiempo no debería poder reescribir.*
>
> LA LECCIÓN DEL CASO 4

El antes y el después: del esquema fragmentado al grafo único

Pongamos las dos arquitecturas frente a frente con una sola pregunta, deliberadamente modesta: *¿cuáles fueron las dos contrapartidas contables de la transferencia 7791 y qué perfil de riesgo tenía Mariana en ese instante exacto?*

Antes, en el modelo relacional. La información se reparte entre tablas como `clientes`, `cuentas`, `transferencias`, `asientos` y `perfiles_riesgo`. La primera mitad de la pregunta se resuelve con un *join* de cuatro tablas (incómodo, pero posible). La

segunda mitad fracasa, porque `perfiles_riesgo` solo guarda el registro *actual*. Para saber qué decía el perfil la noche del cargo hay que rebuscar en tablas de auditoría separadas, si es que alguien las creó; de lo contrario, la respuesta sencillamente no existe en los datos.

```sql
-- Antes: perfiles_riesgo solo guarda el estado ACTUAL — la historia se pierde.
CREATE TABLE transferencias (id INTEGER PRIMARY KEY, monto NUMERIC, ts TEXT);
CREATE TABLE asientos       (id INTEGER PRIMARY KEY, transferencia_id INT, tipo TEXT, monto
NUMERIC);
CREATE TABLE perfiles_riesgo(cliente_id INTEGER PRIMARY KEY, nivel TEXT);  -- sin vigencia

-- Las dos contrapartidas contables de la transferencia: sale bien.
SELECT tipo, monto FROM asientos WHERE transferencia_id = 7791;

-- "¿Qué nivel de riesgo tenía Mariana la noche del cargo?"
-- Imposible: la tabla solo tiene el valor de hoy, no el de entonces.
SELECT nivel FROM perfiles_riesgo WHERE cliente_id = 4410;
```

Eso no es un defecto de un banco en particular: es el defecto estructural del modelo plano frente a la bitemporalidad. *Qué ocurrió* y *qué sabía el sistema cuando ocurrió* son dos ejes de tiempo distintos, y el esquema clásico los aplasta en una sola columna.

Después, en WQuestions. La misma información vive como un grafo de hechos. La transferencia `transferencia_7791` se liga por `parte_de` a sus dos asientos; el perfil `perfil_riesgo_mariana_v4` lleva su propio rango de vigencia `inicio=2026-04-30, fin=2026-06-02`. La pregunta entera se vuelve un patrón de proyección: recoge los hechos cuyos extremos cuelgan de la transferencia y suma el perfil cuyo intervalo temporal solapa con la marca de tiempo del cargo.

No hay *join* de cuatro tablas, no hay tabla de auditoría paralela (el tiempo es un atributo de primer orden en cada hecho) y, sobre todo, no hay que migrar ningún esquema para que funcione. El banco no abandona sus mil quinientas tablas: las deja publicar al grafo. La consulta que antes era un proyecto de semanas pasa a ser una sola pregunta bien formada, y la respuesta que antes «no existía en los datos» pasa a estar siempre disponible, fechada al segundo.

Del préstamo de Mariana a la cartera entera

Una transferencia de Mariana a Bruno, la novela de un préstamo, un cargo fraudulento: casos que se entienden de a uno. El regulador, en cambio, nunca pregunta por Mariana. Pregunta por la cartera entera: cuánto saldo sigue vivo en cada estado de mora, qué proporción de los préstamos lleva más de sesenta días vencido, cómo se movió ese número de un mes al otro. Y esas respuestas no piden tablas nuevas. El préstamo `prestamo_5503` ya guarda su estado fechado con la vigencia D6; el reporte regulatorio es leer ese mismo estado sobre todos los préstamos a la vez y sumar sus montos.

```python
# Saldo vivo de los préstamos en mora de 60 días — un corte; el reporte recorre cada estado
suma(u, "monto", Pattern(fixed={"estado": u.ind("mora_60_dias")},
                         type_constraint=u.ind("prestamo")))
```

Lo que el banco le exigió al modelo

Vale la pena cuantificar de qué tamaño es el problema que el modelo absorbe. Estas son las tablas aproximadas que un banco regional dedica a cada frente (el orden de magnitud que obliga a las cinco islas a existir):

Figura 19.4. Aproximación al número de tablas por frente en un banco regional. WQuestions no reduce ese inventario: cada sistema sigue con sus tablas. Lo que aporta es una sola superficie común (un grafo) donde todas publican, de modo que el cruce entre frentes deja de costar semanas.

Reunamos lo que estas cuatro escenas pusieron a prueba. Ninguna pidió una extensión del modelo; todas cayeron sobre la misma maquinaria que el libro venía construyendo:

EL VEREDICTO DEL DOMINIO MÁS EXIGENTE

Agentes no humanos con responsabilidad. Sistemas que autorizan y verifican operaciones viven en el eje `Q` y firman sus decisiones (D5). El motor antifraude rinde cuentas igual que un cajero.

Contabilidad de partida doble. Eventos que se ramifican en una cara operativa y dos caras contables sin perder el hilo que las une, a través de `parte_de`.

Auditoría indestructible del pasado. La vigencia temporal (D6) preserva el rastro de estados, deudas y reglas obsoletas, y blinda al banco ante juicios y auditorías sin tablas paralelas.

Productos como objetos fechados. La oferta firmada se congela en el tiempo, de modo que un cambio futuro de tarifas nunca reescribe un contrato vigente.

El banco no logró tumbar al modelo. Y lo decisivo no es que «aguantara», sino *cómo* lo hizo: las mismas reglas que en el spa parecían refinamientos elegantes resultaron ser, en banca, las únicas que sobreviven al regulador. Cuando una decisión de diseño que nació por gusto reaparece como exigencia legal en el dominio más duro, deja de ser una hipótesis. En el próximo capítulo subiremos un peldaño en complejidad organizativa: un ERP donde una sola venta atraviesa inventario, finanzas y logística a la vez.

20

Un ERP multi-módulo

El banco demostró que el modelo resiste el peso del dinero. El ERP plantea un reto distinto: una sola venta que, en el mismo instante, cuenta como cuatro cosas a la vez. La prueba ya no es la presión, sino la interoperabilidad interna.

Son las 11:20 de un martes en *Viento Norte*, una fábrica mediana de bombas de riego. La vendedora Ortiz cierra un pedido por teléfono: ocho bombas del modelo K7 para Ferrocom, nueve mil seiscientos dólares. Cuelga, teclea una línea en el sistema y se va a almorzar. En ese gesto de quince segundos acaban de ocurrir cuatro cosas distintas, y ninguna es la venta entera. El almacén del norte debe descontar ocho unidades de su stock. Contabilidad debe registrar un ingreso de nueve mil seiscientos dólares con su asiento. A Ortiz le corresponde una comisión del cuatro por ciento. Y, río arriba, alguien tendrá que reponer ocho bombas, lo que disparará una orden de producción que consumirá cobre y horas de taller. Una frase, cuatro módulos. El examen de este capítulo es lograr que los cuatro vean *el mismo hecho* sin que nadie lo copie.

> **QUÉ ES UN ERP**
> Un *Enterprise Resource Planning* es el software que coordina la operación interna de una empresa: inventario, contabilidad, compras, producción, recursos humanos, logística. Históricamente, cada una de esas piezas nació como un sistema separado.

En los capítulos anteriores de esta parte, el modelo enfrentó adversarios de un solo frente. El spa nos exigió agilidad; el taxi, velocidad; el historial clínico, repartir la verdad entre varios custodios; el banco añadió al regulador. El ERP no sube ninguno de esos picos en particular. Su dificultad es de otra clase: la **convivencia simultánea** de módulos que la industria siempre construyó como mundos aparte. Lo que aquí se pone a prueba no es cuánto aguanta el modelo, sino si los módulos pueden hablarse sin un traductor en el medio.

La maldición de los módulos que no se hablan

Quien haya trabajado cerca de un SAP, un Oracle o un Dynamics reconocerá la escena. Un ERP tradicional es una federación de subsistemas, cada uno con su propio esquema, sus propias tablas y sus propios procesos:

Empleados, jerarquías, salarios, evaluaciones de desempeño, comisiones.

Asientos, presupuestos, cuentas por pagar y por cobrar, cierres.

Stocks, movimientos de almacén, valorizaciones, reposición.

Pedidos, clientes, listas de precios, comisiones de fuerza de ventas.

Órdenes a proveedores, aprobaciones, recepciones, vencimientos.

Lista de materiales (BOM), órdenes de producción, consumos de insumos.

Cada uno de esos módulos vive, históricamente, en su propio archipiélago de tablas. Y la consecuencia es bien conocida por cualquier consultor: **las integraciones entre módulos son el grueso del proyecto**. Se estima que conectar los subsistemas (lograr que la venta «avise» al inventario, que el inventario «avise» a contabilidad, que contabilidad «avise» a recursos humanos) consume del orden del 60 al 70 por ciento del esfuerzo total de una implementación. No se paga por modelar la venta; se paga por coserla a todo lo demás.

Figura 20.1. Reparto aproximado del esfuerzo en una implementación de ERP clásica: el grueso no se va en describir cada módulo, sino en integrarlos entre sí. La promesa de este capítulo es que esa barra (la integración) tiende a cero cuando todos los hechos comparten la misma estructura atómica.

La razón profunda de ese costo es que cada módulo bautiza la realidad a su manera. Para ventas, la operación de Ortiz es un `pedido` con un `cliente_id`; para inventario, es un `movimiento` con un `sku` y un signo negativo; para contabilidad, es un `asiento` con un `haber`. Tres nombres, tres tablas, tres identidades para *el mismo suceso*. Y para volver a juntarlos (para responder «¿cuánta comisión generó la venta que vació ocho bombas del almacén norte?») hace falta cruzar tres bases de datos con claves foráneas frágiles y, casi siempre, un proceso *ETL* nocturno mantenido a mano.

Esto debería sonarte familiar. Es la misma torre de Babel del primer capítulo, pero esta vez no entre empresas distintas: *dentro* de una sola, entre sus propios departamentos. La diferencia es que aquí la cura es más limpia, porque todos los módulos pertenecen al mismo dueño y pueden, de entrada, hablar el mismo idioma.

LA APUESTA DEL ERP

La integración entre módulos **no exige ningún sistema de unificación**. Cuando todos los hechos del negocio comparten la estructura atómica `(sujeto, rol, valor)`, el grafo *ya es* la integración. Una venta no genera «registros en tres tablas distintas»: genera **una situación reificada con varias sub-situaciones ligadas por** `parte_de`, y todas viven en el mismo grafo. Nadie copia el hecho; cada módulo lo mira desde su rol.

El ERP sobre las siete coordenadas

Antes de los escenarios, repartamos las piezas de la fábrica en los ejes de valor. El ejercicio ordena, en seis preguntas, lo que en un ERP tradicional son docenas de tablas inconexas.

Q QUIÉN · AGENTES

Empleados, clientes y proveedores, y —decisivo— la empresa misma. `vientonorte_sa` es una persona jurídica: un agente tan válido como la vendedora Ortiz (D5). El director reporta a la empresa, no al vacío.

O QUÉ · ENTIDADES Y EVENTOS

Productos, órdenes, asientos, comisiones, evaluaciones de desempeño, movimientos de almacén. Todo lo que tiene fecha de creación, historia y trazabilidad es una entidad de primera clase.

L DÓNDE · LUGARES

Departamentos, almacenes, plantas, sucursales. Forman una jerarquía territorial-organizacional que se arma con `parte_de`, igual que cualquier otra.

T CUÁNDO · TIEMPOS

Fechas de pedido, períodos contables, vencimientos de órdenes de compra, vigencias de salario. El tiempo gobierna los estados que cambian.

N CUÁNTO · MAGNITUDES

Unidades vendidas, montos, horas-hombre, kilos de insumo. Cada número arrastra su unidad anclada en `K`: nada anda suelto.

K CUÁL · CLASES

Tipos de producto, categorías de empleado, estados de orden, monedas. El zócalo categórico que da sentido a las magnitudes y los estados.

Y atravesando las seis cajas, los predicados del séptimo eje (M, el *cómo*) son los cables que articulan todo: `agente`, `parte_de`, `reporta_a`, `motivado_por`. Las decisiones de diseño que el ERP más ejercita son cuatro, y conviene tenerlas a mano porque reaparecerán en cada escenario: la **D3** (todo es tripleta atómica), la **D4** (una operación que tocaría varios módulos se reifica como una situación con sub-situaciones), la **D6** (los estados, salarios y precios que cambian guardan su historial) y la **D7** (cada acción lleva su `motivado_por` y su `justificado_por`: la auditoría es nativa, no un añadido).

Caso 1 · El organigrama es un solo cable

Todo ERP empieza por el organigrama. Una empresa tiene un director, sus gerentes, sus equipos. En SAP esa estructura vive en una tabla llamada `HRP1001` con un esquema gimnástico de «relaciones» genéricas que hay que decodificar con catálogos auxiliares. En WQuestions es un único rol de dominio, `reporta_a`, y la jerarquía completa de Viento Norte cabe en cuatro tripletas:

TRIPLETAS

```
(vendedora_ortiz, reporta_a, gerente_duarte)
(jefe_planta_cordero, reporta_a, gerente_duarte)
(gerente_duarte, reporta_a, directora_salas)
(directora_salas, reporta_a, vientonorte_sa)
```

La última línea es la que delata la potencia del eje `Q`: la empresa también es un agente. La directora no reporta a la nada; reporta a la persona jurídica `vientonorte_sa`, que firma contratos y rinde cuentas igual que un humano (D5). La pregunta «¿quién es el jefe del jefe de Ortiz?» es un recorrido transitivo de dos saltos sobre `reporta_a`; «todo el árbol bajo Duarte» es la relación inversa recorrida en forma recursiva. No hace falta una tabla de *cierre transitivo*: el motor la calcula al vuelo.

POLÍTICA LIBERAL

`reporta_a` no figura en el catálogo canónico de roles, pero el sistema lo acepta sin protestar: es un rol de dominio, admitido por política liberal. El catálogo (D8) no crece; el lexicon, sí. La diferencia se desarrolla en el capítulo 14.

Asignar a cada empleado un departamento físico es igual de directo, y aquí el departamento es un lugar del eje `L`:

TRIPLETAS

```
(vendedora_ortiz, trabaja_en, dpto_ventas)        # dpto_ventas ∈ L
(jefe_planta_cordero, trabaja_en, dpto_produccion) # dpto_produccion ∈ L
(dpto_ventas, parte_de, sede_central)             # la jerarquía territorial
(almacen_norte, parte_de, sede_central)           # se arma con parte_de
```

Si más adelante un departamento contuviera sub-departamentos, o una planta agrupara varias líneas, usarías `parte_de` exactamente igual. La jerarquía de personas y la de lugares se modelan con la misma maquinaria; solo cambia el eje sobre el que viven los nodos.

Caso 2 · La lista de materiales (BOM) es `parte_de` recursivo

Aquí el modelo luce. Una *lista de materiales* (el BOM, por *bill of materials*) es la descomposición jerárquica de un producto en sus subproductos y, al final, en su materia prima. Es uno de los conceptos más notoriamente incómodos de implementar en bases relacionales: los MRP clásicos lo resuelven con tablas como `mast`, `matl` y `caps` unidas por *joins* que pocos disfrutan escribir.

Modelemos la bomba de riego K7. Tiene una carcasa de acero, un motor y un impulsor; el motor, a su vez, lleva un bobinado de cobre. Cuatro niveles de anidamiento:

TEXTO

```
bomba_riego_k7  ─┬─ carcasa_acero
                 ├─ impulsor_bronce
                 └─ motor_electrico ─┬─ eje_motor
                                     └─ bobinado_cobre ── mp_cobre
```

En WQuestions, cada arista de ese árbol es **una sola tripleta** con `parte_de`:

TRIPLETAS

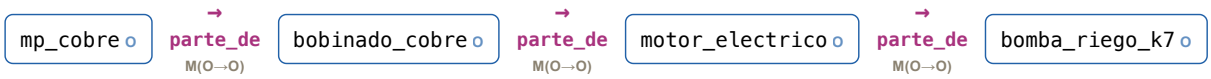
```
(carcasa_acero,    parte_de, bomba_riego_k7)
(impulsor_bronce,  parte_de, bomba_riego_k7)
(motor_electrico,  parte_de, bomba_riego_k7)
(eje_motor,        parte_de, motor_electrico)
(bobinado_cobre,   parte_de, motor_electrico)
(mp_cobre,         parte_de, bobinado_cobre)
```

¿Y cuántas unidades de cada subproducto entran? Esa cantidad no es del producto: es del *enlace*. Por eso se modela como un atributo de la relación, con su número y su unidad ancladas:

PYTHON

```
u.assert_fact(bobinado_cobre, "cantidad_en_bom",
              n(2, "unidades", "n_2_uds_bobinado"))
u.assert_fact(mp_cobre, "cantidad_en_bom",
              n(0.4, "kg", "n_04_kg_cobre"))
```

Para responder «¿qué materia prima necesita la bomba?» basta con recorrer `parte_de` en forma recursiva hasta tocar las hojas del árbol. El recorrido no conoce la profundidad de antemano y no le importa: la misma maquinaria sirve para una bomba de cuatro niveles, un electrodoméstico de seis o un tablero electrónico de diez.



UNA SOLA REGLA, DOS JERARQUÍAS DISTINTAS

Fíjate en que `parte_de` ya hizo dos trabajos en este capítulo: armó la jerarquía de *lugares* (el almacén es parte de la sede) y la de *componentes* (el cobre es parte del bobinado). No son dos mecanismos parecidos: son el **mismo** cable del eje `M`, recorrido sobre nodos de distinto eje. La recursión arbitraria del BOM no pide ningún esquema especial; es un caso particular de algo que el modelo ya tenía.

Caso 3 · La venta cross-módulo: el patrón que vale oro

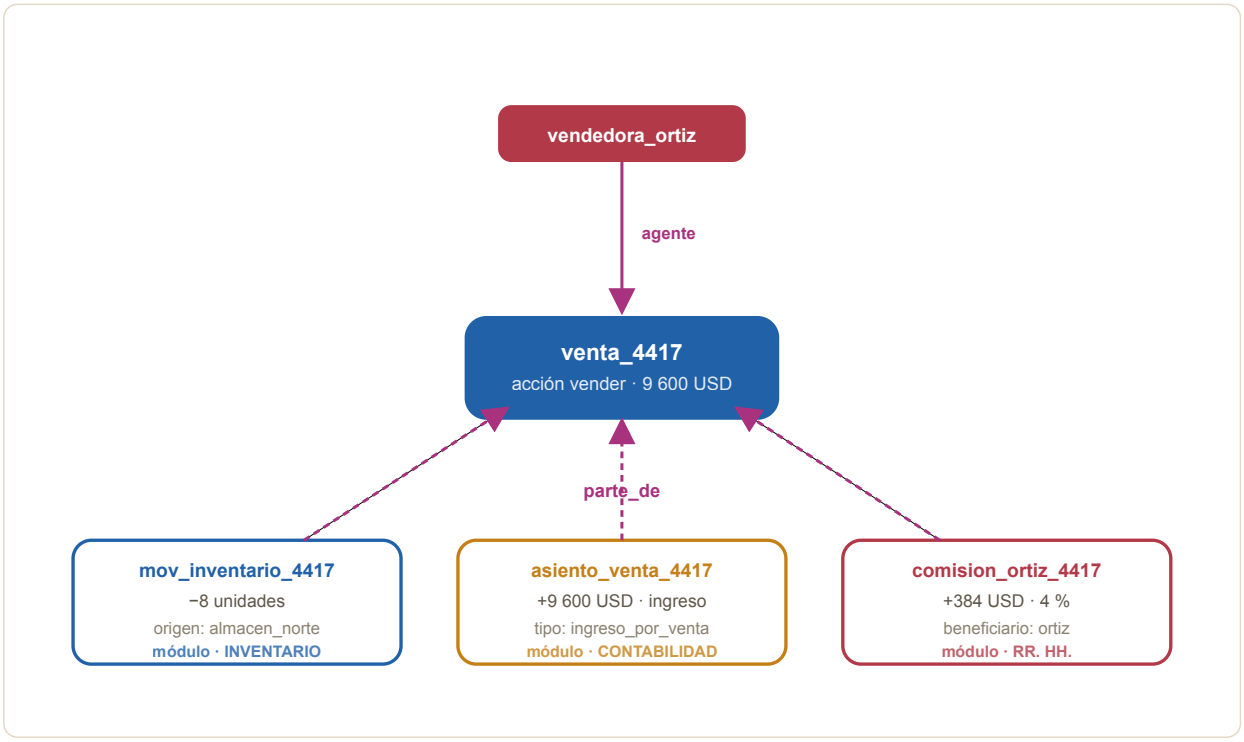
Llegamos al caso emblemático, el que da título al capítulo. Volvamos a la llamada de las 11:20: ocho bombas K7 para Ferrocom por nueve mil seiscientos dólares. Ese único hecho toca, a la vez, tres módulos:

Inventario: hay que descontar 8 unidades del almacén del norte.

Contabilidad: hay que registrar el ingreso de 9.600 USD con su asiento.

Recursos humanos: a Ortiz le corresponde una comisión del 4 % sobre la venta: 384 USD.

En un ERP tradicional, esta venta dispara *tres procesos separados* que escriben en tres tablas distintas, unidas por claves foráneas y convenciones de nombres. La pregunta de auditoría más simple («explícame qué pasó con esta venta») obliga a cruzar las tres bases. En WQuestions el modelado es directo: **una situación reificada con tres sub-situaciones**. Esto es exactamente la D4 en acción (la operación que tocaría varios módulos se reifica) y la figura lo muestra antes que el código.



> **Figura 20.2.** La venta `venta_4417` como situación central, referida por tres sub-situaciones que cuelgan de ella por `parte_de`: el movimiento de inventario, el asiento contable y la comisión. Tres módulos miran *el mismo hecho* desde su rol —ninguno lo copia—. En un ERP clásico, esto serían tres tablas y un *ETL* nocturno para volver a juntarlas.

Primero declaramos la situación articuladora con sus roles principales; luego colgamos de ella, por `parte_de`, una sub-situación por módulo:

PYTHON

```
venta = ingest_situation(u, lex, "vender", roles={
    "agente":   vendedora_ortiz,
    "tema":     bomba_riego_k7,
    "cliente":  cliente_ferrocom,
    "monto":    n(9600, "USD"),
    "momento":  at("2026-07-15T11:20"),
    "lugar_de": dpto_ventas,
}, sit_id="venta_4417")

# Sub-situación 1 — INVENTARIO: descuenta 8 unidades del almacén norte
u.assert_fact(mov_inventario, "parte_de", venta)
u.assert_fact(mov_inventario, "tema",     bomba_riego_k7)
u.assert_fact(mov_inventario, "cantidad", n(8, "unidades"))
```

```
u.assert_fact(mov_inventario, "origen",   almacen_norte)

# Sub-situación 2 — CONTABILIDAD: registra el ingreso
u.assert_fact(asiento_venta, "parte_de",        venta)
u.assert_fact(asiento_venta, "monto",           n(9600, "USD"))
u.assert_fact(asiento_venta, "tipo_movimiento", ingreso_por_venta)

# Sub-situación 3 — RR. HH.: comisión del 4 % para Ortiz
u.assert_fact(comision_ortiz, "parte_de",        venta)
u.assert_fact(comision_ortiz, "beneficiario",    vendedora_ortiz)
u.assert_fact(comision_ortiz, "monto",           n(384, "USD"))
u.assert_fact(comision_ortiz, "calculado_segun", politica_comisiones_v2)
```

Tres sub-situaciones, una situación articuladora, **ningún cruce de tablas necesario**. Y como cada sub-situación tiene su propia identidad, podemos hacer preguntas finas que en un ERP clásico exigirían su propio reporte. «Explícame todo lo que pasó con la venta 4417» es una proyección directa por `parte_de`:

PYTHON

```
sub_situaciones = [
    f.subject for f in u.facts_with_role("parte_de")
    if f.value.id == "venta_4417"
]
# → [mov_inventario_4417, asiento_venta_4417, comision_ortiz_4417]
```

«¿Qué comisión generó esta venta?», «¿cuál es su asiento contable?», «¿de qué almacén salió la mercancía?»: todas son recorridos de un salto sobre el mismo nodo central. Sin *ETL*, sin sistema intermedio, sin tabla de unificación. Lo que en un ERP tradicional es un proyecto de integración, aquí es la forma normal de escribir el hecho.

> *La venta no se reparte en tres registros que después hay que volver a unir. Es un solo hecho que cada módulo mira desde su rol: el grafo ya es la integración.*

LA LECCIÓN DEL CASO 3

De la venta 4417 al cierre de mes

Acompañamos una venta que cruzó inventario, contabilidad y compras. La gerencia no mira la `venta_4417`: mira el cierre de mes. Cuánto facturó cada línea de producto, qué margen dejó cada sede, cuántas bombas K7 quedan en el almacén norte, qué comisiones acumuló Ortiz en el trimestre. Lo notable es que ninguno de esos reportes vive en un módulo. Son cortes distintos del mismo grafo de situaciones —cada uno fija una coordenada y deja libre el resto—, y todos los sumandos ya estaban escritos cuando cerró el pedido de las 11:20. Sumar las ventas de una línea es el corte más simple del consolidado:

PYTHON

```
# Facturación de una línea de producto — un corte; el cierre de mes recorre todas
suma(u, "monto", Pattern(fixed={"tema": u.ind("bomba_riego_k7")},
                         type_constraint=u.ind("vender")))
```

Caso 4 · La orden de compra y su aprobación bitemporal

Río arriba de la venta hay que reponer stock, y eso dispara compras. Las órdenes grandes (digamos 48.000 USD de cobre al proveedor) no se ejecutan al instante: pasan por un ciclo. Alguien la redacta, alguien la manda a aprobación, alguien con autoridad la aprueba o la rechaza. Cada estado tiene su fecha de entrada y su fecha de salida.

En sistemas tradicionales, esto vive en una columna `estado` que **se sobrescribe** en cada transición. Cuando alguien pregunta «¿en qué estado estaba esta orden el 20 de julio a la una de la tarde?», hay que rebuscar en la tabla de auditoría (si existe) o estimarlo a partir de los *logs*. En WQuestions, el estado es un hecho con vigencia temporal (D6): no se pisa, se acumula.

> **D6 VIGENCIA: EL ESTADO SE ACUMULA, NO SE PISA**
>
> Ningún atributo que pueda cambiar se guarda como un valor desnudo, sino con un intervalo `[inicio, fin)`. Cambiar de estado no es modificar una celda: es *cerrar* el intervalo vigente y *abrir* uno nuevo. La consecuencia es que el grafo conserva, gratis, toda la trayectoria del atributo a lo largo del tiempo.

PYTHON

```
u.assert_fact(oc_2208, "estado", borrador,
              valid_from=t_creacion,    valid_to=t_envio_aprob)
u.assert_fact(oc_2208, "estado", pendiente_aprobacion,
              valid_from=t_envio_aprob, valid_to=t_aprobacion)
u.assert_fact(oc_2208, "estado", aprobada,
              valid_from=t_aprobacion)
```

Tres tripletas en lugar de tres `update` destructivos. La consulta bitemporal «¿en qué estado estaba el 20 de julio a las 13:00?» devuelve `pendiente_aprobacion`; «¿y el 22 de julio?», devuelve `aprobada`. El historial completo es nativo, no un módulo aparte. La aprobación misma, además, no es una simple marca: es una situación reificada con su agente y su justificación normativa.

PYTHON

```
aprobacion = ingest_situation(u, lex, "aprobar", roles={
    "agente":          gerente_duarte,
    "tema":            oc_2208,
    "momento":         at("2026-07-21T15:30"),
    "justificado_por": politica_aprobacion_oc_v4,
}, sit_id="aprobacion_oc_2208")
```

Duarte no aprueba «porque sí»: aprueba **porque la política corporativa lo autoriza** para órdenes mayores a 30.000 USD. Aquí asoma la D7: el «por qué» no es un eje, sino una relación entre hechos. La autorización se ata por `justificado_por` al documento normativo, que es a su vez otro objeto reificado con su umbral y su vigencia. Y esto tiene una consecuencia elegante: si mañana cambia el umbral, la política se reifica como versión nueva y las aprobaciones viejas siguen apuntando a la anterior, sin alterar el historial.

> **ECO DEL BANCO**
>
> Esta es la misma maquinaria de vigencia que en el capítulo anterior reconstruía el estado de un préstamo en una audiencia. Lo que allí era exigencia del regulador, aquí es simple buen gobierno de compras: la misma regla, dos motivaciones.

Caso 5 · La orden de producción que consume insumos

Las ocho bombas que salieron del almacén hay que reponerlas: la planta de Cordero va a producir cincuenta. Cada bomba consume cierta cantidad de cobre (para el bobinado) y cierta cantidad de horas-hombre de taller. El ERP necesita registrar tres cosas (la orden como evento principal, el consumo de insumos y el consumo de mano de obra) y el patrón es, felizmente, el mismo de la venta cross-módulo:

```python
op = ingest_situation(u, lex, "producir", roles={
    "agente":   jefe_planta_cordero,
    "tema":     bomba_riego_k7,
    "cantidad": n(50, "unidades"),
    "lugar_de": dpto_produccion,
}, sit_id="op_3390")

# Consumo de materia prima
u.assert_fact(consumo_cobre, "parte_de", op)
u.assert_fact(consumo_cobre, "tema",     mp_cobre)
u.assert_fact(consumo_cobre, "cantidad", n(20, "kg"))

# Consumo de mano de obra
u.assert_fact(consumo_hh, "parte_de",      op)
u.assert_fact(consumo_hh, "cantidad",      n(160, "horas"))
u.assert_fact(consumo_hh, "ejecutado_por", jefe_planta_cordero)
```

Lo notable es que la **misma estructura** que modela una bomba producida por un equipo humano podría modelar, sin cambiar una línea, una reacción química industrial donde *no hay agente humano* y los reactivos se transforman solos siguiendo su estequiometría. El catálogo ya admite que el rol `agente` sea opcional cuando el verbo lo permite (D5); por eso el mismo molde sirve para «Cordero ensambla» y para «la mezcla reacciona». Ese caso límite (química sin agente) se explora en el capítulo 25.

La consulta «¿cuántos kilos de cobre consumió esta orden?» es, otra vez, una proyección: encuentra las sub-situaciones de `op_3390`, filtra por `tema = mp_cobre` y suma sus `cantidad`. Tres operaciones simples sobre el grafo. Cero *ETL*. Y nota lo que acaba de pasar entre bastidores: el cobre que esta orden consume es el mismo `mp_cobre` que el BOM del Caso 2 declaró como hoja del árbol de la bomba. Inventario de insumos y lista de materiales no son dos módulos que haya que sincronizar: son el mismo nodo, visto desde dos preguntas.

Caso 6 · El rastro bitemporal de un salario

Cerramos con una situación que en cualquier ERP es notoriamente difícil. Ortiz fue contratada el 10 de enero con un salario de 2.500 USD. El 1 de abril le subieron a 2.800. El 1 de agosto, tras una evaluación de desempeño con calificación «excelente», le subieron de nuevo a 3.200. La pregunta que el modelo tiene que responder sin esfuerzo es la que hace todo auditor laboral al revisar un legajo: **¿cuál era el salario de Ortiz el 15 de marzo? ¿Y el 15 de junio? ¿Y el 15 de septiembre?**

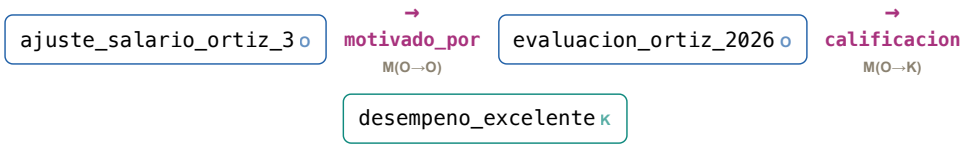
En un ERP tradicional, esto exige una tabla histórica auxiliar (`emp_salary_hist` y parientes) que la mitad de las implementaciones nunca termina de mantener al día. En WQuestions, es la misma D6 del Caso 4 aplicada a un número: **tres tripletas** que conviven en el grafo.

```python
u.assert_fact(vendedora_ortiz, "salario_mensual", n(2500, "USD"),
              valid_from=t_contrato, valid_to=t_aumento1)
u.assert_fact(vendedora_ortiz, "salario_mensual", n(2800, "USD"),
              valid_from=t_aumento1, valid_to=t_aumento2)
u.assert_fact(vendedora_ortiz, "salario_mensual", n(3200, "USD"),
              valid_from=t_aumento2)
```

La consulta `query(..., at=fecha)` filtra por vigencia y devuelve el valor que regía en ese instante, no el de hoy:

```texto
marzo       → 2 500 USD
junio       → 2 800 USD
septiembre  → 3 200 USD
```

Y el último aumento no flota suelto: se conecta con su causa. La evaluación de desempeño es un objeto reificado propio, y el ajuste de salario lleva un `motivado_por` que apunta a ella (D7). Si el auditor pregunta «¿por qué le subieron el salario a Ortiz en agosto?», el grafo responde sin gimnasia: porque `evaluacion_ortiz_2026`, con calificación excelente, lo motivó.



El antes y el después: del esquema fragmentado al grafo único

Pongamos las dos arquitecturas frente a frente con una pregunta transversal deliberadamente incómoda: *¿cuántas horas-hombre de producción demandó cada unidad que Ortiz vendió este trimestre, y cuánto cobró ella de comisión por esa venta?*

Antes, en el modelo relacional. La información vive en islas: `employees` y `emp_salary_hist` en RR. HH.; `sales_orders` y `sales_commissions` en Ventas; `production_orders` y `wh_stock_movements` en Manufactura e Inventario. La pregunta obliga a cruzar cuatro tablas de tres módulos distintos con *joins* frágiles, claves foráneas de convención y, casi siempre, una vista *ETL* que el equipo de inteligencia de negocio mantiene a mano. Cualquier cambio de esquema en un módulo rompe el reporte en silencio.

```sql
-- Antes: cuatro tablas, tres módulos, joins por convención.
-- Y la venta vive partida en tres registros que hay que volver a unir.
SELECT  so.id              AS venta,
        sc.monto           AS comision,
        SUM(wh.horas)      AS horas_hombre
FROM        sales_orders        so
JOIN        sales_commissions   sc ON sc.order_id = so.id
JOIN        production_orders    po ON po.product_id = so.product_id
JOIN        wh_stock_movements  wh ON wh.po_id = po.id   -- FK frágil
WHERE   so.seller_id = 'ORTIZ'
  AND   so.fecha BETWEEN '2026-07-01' AND '2026-09-30'
GROUP BY so.id, sc.monto;
-- Si Manufactura renombra wh.po_id, el reporte calla y miente.
```

Después, en WQuestions. La misma información ya existe como un único grafo de hechos atómicos. La `venta_4417` es una situación reificada que contiene, por `parte_de`, la comisión de Ortiz, el movimiento de inventario y (a través de la orden de producción que repuso el stock) el consumo de horas-hombre. La pregunta transversal es una proyección de unos pocos saltos: localiza las ventas de Ortiz del trimestre, recorre sus sub-situaciones, lee `monto` en la comisión y `cantidad` en el consumo de mano de obra.

No hay *join* de cuatro tablas, no hay vista *ETL* paralela, y sobre todo no hay silos, porque **nunca hubo fronteras de módulo**: hubo una sola venta, y el grafo la modeló entera. Un cambio de vocabulario en un módulo no rompe nada, porque no existe el módulo como tabla separada: existe el rol, mapeado una sola vez en el lexicon. La consulta que antes era un proyecto pasa a ser una sola pregunta bien formada.

El veredicto del dominio multi-módulo

El ERP es, probablemente, el dominio donde la promesa de WQuestions paga su dividendo más concreto: **la integración entre módulos deja de ser un proyecto y se vuelve gratuita**. Repasemos qué pidió cada escena y con qué la pagó el modelo, porque la lista entera cae sobre maquinaria que el libro ya tenía construida:

> **LO QUE EL ERP PUSO A PRUEBA**
>
> **La venta cross-módulo.** No genera «registros en tres tablas»: genera una situación con tres sub-situaciones ligadas por `parte_de` (D4). Cada módulo la mira desde su rol, ninguno la copia.
>
> **La lista de materiales.** No necesita esquemas auxiliares; es `parte_de` recorrido en forma recursiva, el mismo cable que arma la jerarquía de lugares.
>
> **El rastro bitemporal.** Estados de órdenes y salarios que cambian no se sobrescriben; conservan su historial nativo con vigencia (D6). El auditor reconstruye cualquier instante sin tablas paralelas.
>
> **La aprobación y el aumento.** No son *workflows* ni banderas externas: son situaciones reificadas con su `justificado_por` y su `motivado_por` (D7). La auditoría es nativa, no un añadido.

Y todo se modeló con los mismos roles canónicos del catálogo (D8) más un puñado de roles de dominio (`reporta_a`, `trabaja_en`, `calculado_segun`, `cantidad_en_bom`, `ejecutado_por`) admitidos por la política liberal. El motor no creció ni una línea para absorber el ERP. Lo único que creció fue el lexicon, con un puñado de verbos nuevos (`vender`, `ordenar_compra`, `aprobar`, `producir`, `contratar`, `ajustar_salario`) y sus signaturas.

Si el banco demostró que el modelo soporta el peso del dinero, el ERP demuestra que soporta el peso de la *coordinación*: muchas voces internas describiendo, cada una a su manera, hechos que en el fondo son uno solo. El próximo capítulo cambia de terreno (de la fábrica al aula) y trae un grafo de otra naturaleza: una universidad, donde los prerrequisitos forman cadenas que el modelo deberá recorrer como un dependiente al que nunca le faltó un eslabón.

21

Una universidad

Aquí la dificultad no es el volumen ni la ley, sino la forma: el saber se ordena en un grafo de dependencias y una vida académica dura décadas. Veamos si el modelo sabe recorrer caminos.

Es lunes de matrícula y Tomás Quiroz tiene quince navegadores abiertos. Cursa el cuarto semestre de Ciencia de la Computación y quiere inscribirse en *Algoritmos*, el curso que todos dicen que «abre puertas». El portal de matrícula lo rechaza con un mensaje escueto: *«No cumple prerrequisitos»*. Pero Tomás está seguro de haber aprobado lo que hacía falta. ¿Le falta *Estructuras de Datos*, que aprobó el ciclo pasado? ¿O es la rama de matemática, *Cálculo*, que arrastra desde primero? El sistema no se lo dice: solo sabe responder sí o no, nunca *por dónde*. Y detrás de ese silencio se esconde el problema técnico que define a una universidad como dominio.

> **POR QUÉ ESTE DOMINIO**
> El banco era el dominio de la *ley*; el ERP, el de la *integración*. La universidad es el dominio de la *forma*: dependencias que se encadenan y un reloj que no se detiene en cinco años, sino en cincuenta.

Porque una universidad no se parece a un comercio ni a un banco. Sus dos rasgos propios son de otra naturaleza. El primero: **los tiempos son largos**. Una carrera dura cinco años, pero el historial de un egresado lo acompaña toda la vida: la nota de un curso que rindió a los veinte le sirve para un posgrado a los cuarenta. El segundo, y el que da título a este capítulo: **las cosas dependen unas de otras**. No puedes tomar *Algoritmos* sin antes haber pasado por *Estructuras de Datos*, y a esa no llegas sin *Programación*. La malla curricular es, literalmente, un grafo de dependencias.

Cualquier sistema académico serio tiene que poder responder, sin sudar, preguntas como estas:

- ¿Qué cursos *puede* tomar Tomás este semestre, dado lo que ya aprobó?
- ¿Qué nota sacó Lucía en *Cálculo I* antes de su reclamo? ¿Y después de la rectificación?
- ¿Quiénes seguían matriculados en *Bases de Datos* el 1 de mayo?
- ¿Qué cadena de prerrequisitos arrastra *Aprendizaje de Máquina*, y cuáles de ellos le faltan a Tomás?
- ¿Quiénes integraron el jurado de la tesis de Renata?

En una base relacional clásica, cada una de estas preguntas pide algo distinto: una tabla histórica aquí, un `WITH RECURSIVE` allá, un procedimiento almacenado que replica en código la lógica del recorrido. En WQuestions, como veremos, **cada una es un recorrido de uno o dos saltos sobre el mismo grafo de hechos**. No hay máquina distinta para cada pregunta: hay un solo mapa y muchas rutas sobre él.

La universidad sobre las siete coordenadas

Antes de los casos, repartamos las piezas del mundo académico entre los ejes de valor. El ejercicio ordena de un golpe lo que en un sistema heredado vive disperso en una veintena de tablas:

Q QUIÉN · AGENTES

Estudiantes, docentes, autoridades, miembros de jurado y la propia universidad como persona jurídica. Una sola persona puede aparecer en muchos roles: Renata es estudiante, jefa de práctica y tesista a la vez.

O QUÉ · ENTIDADES Y EVENTOS

Carreras, planes, cursos, matrículas, evaluaciones, reclamos, tesis, defensas y graduaciones. Cada uno con identidad propia y trazabilidad. Aquí vive el grueso del peso académico.

L DÓNDE · LUGARES

Facultades, aulas, laboratorios, auditorios y campus. Forman jerarquías como cualquier organización territorial: el aula es parte del pabellón, el pabellón del campus.

T CUÁNDO · TIEMPOS

Inicio y fin de cada ciclo, momento exacto de una matrícula, fecha de un examen, día de una defensa. El reloj académico es largo y no olvida: una nota de hace veinte años sigue siendo consultable.

N CUÁNTO · MAGNITUDES

Créditos, notas numéricas, horas semanales, montos de pensión. Cada magnitud arrastra su unidad: cuatro créditos no son cuatro puntos ni cuatro horas.

K CUÁL · CLASES

Tipos de carrera, estados de matrícula (vigente / retirado), modalidades de evaluación y niveles cualitativos (aprobado / desaprobado). El zócalo categórico que da sentido al resto.

Y el séptimo eje, **M** (el *cómo*), aporta los cables que amarran todo: `parte_de` arma las jerarquías, `requiere_prereq` teje la malla, `agente` ata cada persona a su situación. Sin esos enlaces, las seis cajas anteriores serían seis listas inertes. Con ellos, una vida académica entera se vuelve un grafo navegable. Recorrámoslo.

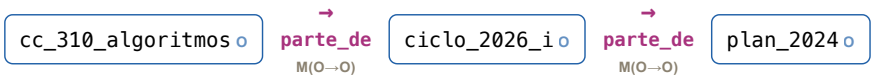
Caso 1 · La estructura académica como jerarquía

El plan de estudios de Tomás tiene una forma de árbol que el modelo absorbe sin esfuerzo. La carrera contiene años; cada año, ciclos; cada ciclo, cursos. Una sola relación, `parte_de`, expresa los cuatro niveles:

TRIPLETAS

```
(carrera_cc) ∈ O                        # Ciencia de la Computación
(plan_2024)        parte_de  carrera_cc        # la malla vigente desde 2024
(ciclo_2026_i)     parte_de  plan_2024         # el ciclo en curso
(cc_310_algoritmos) parte_de ciclo_2026_i      # un curso del ciclo
(cc_310_algoritmos) creditos  n_4_cr           # arrastra su magnitud (N)
(cc_310_algoritmos) docente   prof_salazar     # apunta a un agente (Q)
(cc_310_algoritmos) dictado_en lab_b2          # y a un lugar (L)
```

Cada nivel es una entidad en `O`, y *es parte de* el nivel superior. Tres tripletas de jerarquía y la columna vertebral de la carrera queda armada. Cuando Tomás pregunta «¿qué cursos hay en mi ciclo actual?», la consulta es un único salto: trae todas las entidades de `O` cuyo `parte_de` sea `ciclo_2026_i`. Y cada curso, además de colgar de su ciclo, lleva sus propios atributos (créditos, docente, aula, código, sílabo) sin que ninguno necesite una tabla aparte. Todo el inventario académico tiene la misma forma.



Caso 2 · Los prerrequisitos como grafo dirigido acíclico

Llegamos al corazón del capítulo. La malla curricular (qué curso exige haber pasado por cuál) es por su propia naturaleza un **grafo dirigido acíclico**, un DAG: las aristas tienen sentido (de requisito a curso), no hay ciclos (ningún curso puede exigirse a sí mismo, ni directa ni indirectamente) y las dependencias se cruzan (un curso puede tener varios requisitos, y un requisito puede habilitar varios cursos).

> **DAG · GRAFO DIRIGIDO ACÍCLICO**
>
> Un conjunto de nodos unidos por flechas con sentido, sin que ninguna ruta te devuelva al punto de partida. Es la estructura natural de toda dependencia ordenada: las tareas de un proyecto, las versiones de un documento, las migraciones de una base de datos —y los prerrequisitos de una carrera—. Que no haya ciclos es justo lo que garantiza que siempre exista un orden válido para avanzar.

En un sistema relacional, esto se modela con una tabla `curso_prereq(curso, prereq)` y se consulta con SQL recursivo. En WQuestions, **cada prerrequisito es una sola tripleta**. Tomamos un fragmento real de la malla de Tomás (dos líneas que confluyen, la de matemática y la de programación):

TRIPLETAS

```
# línea de programación
(cc_210_estructuras) requiere_prereq  cc_101_progra
(cc_220_basesdatos)  requiere_prereq  cc_210_estructuras
(cc_310_algoritmos)  requiere_prereq  cc_210_estructuras

# línea de matemática
(ma_120_calculo)     requiere_prereq  ma_110_discreta
(ma_230_probabilidad) requiere_prereq ma_120_calculo

# el curso que las une: necesita las DOS ramas a la vez
(cc_310_algoritmos)  requiere_prereq  ma_120_calculo
(cc_420_aprendizaje) requiere_prereq  cc_310_algoritmos
(cc_420_aprendizaje) requiere_prereq  ma_230_probabilidad
```

Algoritmos (`cc_310`) exige **dos** requisitos simultáneos (uno de cada rama), y *Aprendizaje de Máquina* (`cc_420`), la cima de la carrera, exige otros dos. El modelo lo absorbe sin pestañear porque `requiere_prereq` es un cable múltiple: un mismo curso puede emitir varias flechas de requisito, cada una su propio hecho atómico, independiente de las demás. No hay que inventar una «tabla de dependencias compuestas»; basta apilar tripletas.

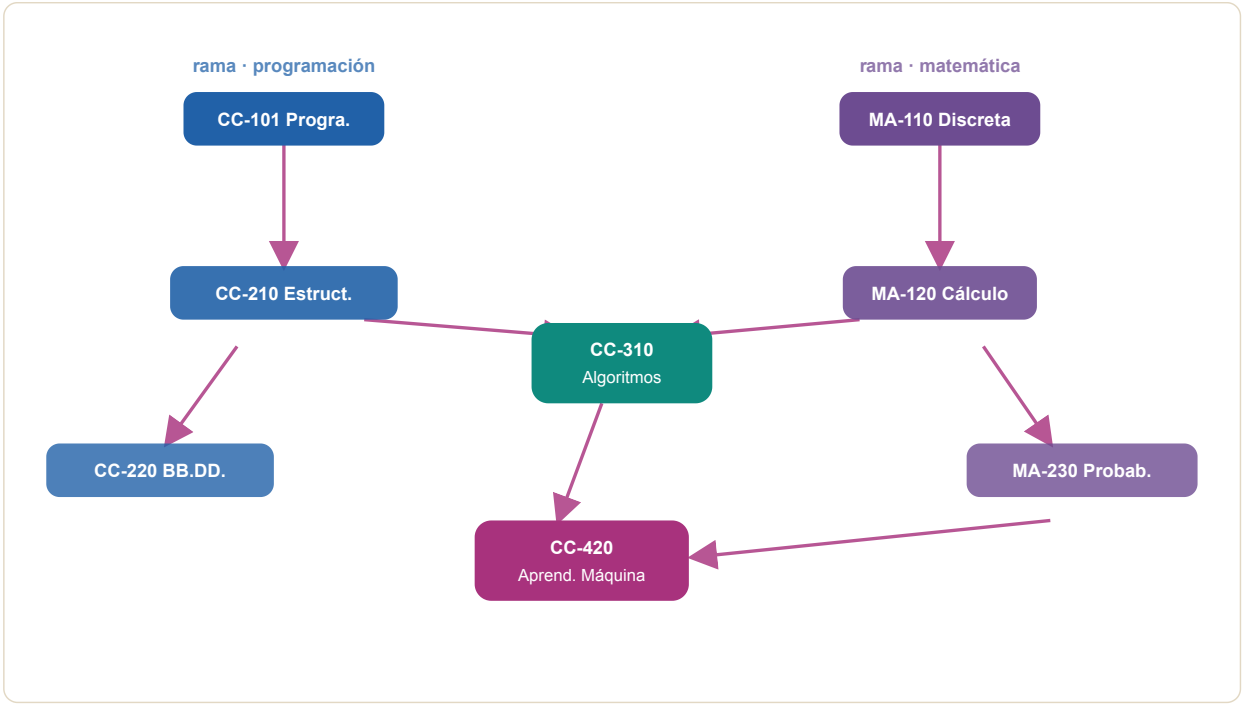


Figura 21.1. La malla de Tomás como grafo dirigido acíclico. Cada curso es un individuo de `0`; cada flecha es una tripleta `requiere_prereq` (en color *cómo*). Las dos ramas (programación y matemática) confluyen en *Algoritmos*, que requiere ambas a la vez, y la carrera culmina en *Aprendizaje de Máquina*, que depende de *Algoritmos* y de *Probabilidad*. Cada dependencia es un hecho atómico independiente.

Ahora podemos resolver el drama del lunes. La pregunta «¿qué cursos puede tomar Tomás este ciclo?» se vuelve un recorrido directo sobre el grafo. Para cada curso del ciclo, el motor mira sus flechas `requiere_prereq` y verifica que **todos** los requisitos tengan, en el historial de Tomás, una evaluación con estado *aprobado*. Si falta uno solo, el curso queda excluido. Y (esto es lo que el portal del lunes no sabía hacer) el mismo recorrido puede *devolver el requisito que falta*, no solo un «no».

EL RECORRIDO DE ELEGIBILIDAD, EN UNA FRASE

Un curso es elegible para un estudiante si, y solo si, el conjunto de los destinos de sus aristas `requiere_prereq` está contenido en el conjunto de cursos que ese estudiante aprobó. Es una operación de conjuntos sobre el grafo: sin *join*, sin SQL recursivo, sin procedimiento almacenado. La diferencia entre los dos conjuntos es, además, la lista exacta de lo que le falta.

Apliquémoslo. Tomás aprobó *Programación*, *Estructuras de Datos* y *Discreta*, pero no *Cálculo*. *Algoritmos* exige `cc_210_estructuras` ✓ y `ma_120_calculo` ✗. El recorrido devuelve un único faltante: Cálculo. El portal podría haberle dicho, en vez de un «no» mudo: *«Te falta Cálculo I para habilitar Algoritmos»*. El grafo conocía la respuesta desde el principio; solo había que recorrerlo en lugar de comparar banderas.

Caso 3 · La matrícula que cambia de estado sin perder el pasado

Tomás se matricula en *Bases de Datos* el 10 de marzo. Lucía también, dos días después. Pero a mediados de abril Lucía se retira del curso. ¿Cómo guardamos esto sin perder información? La tentación clásica (y el error más común) es poner una columna `estado` en la tabla de matrículas y *actualizarla* cuando alguien se retira. El problema aparece en julio, cuando la secretaría pregunta: «¿quiénes estaban matriculados el 1 de abril?». La respuesta fiel ya no existe: el `UPDATE` borró el estado anterior.

En WQuestions los dos estados **conviven** en el grafo, separados por vigencia. No se sobrescribe nada: se cierra el intervalo del estado viejo y se abre el del nuevo.

D6 VIGENCIA: EL PASADO NO SE SOBRESCRIBE

Como vimos al modelar el banco, ningún atributo que pueda cambiar se guarda como un valor desnudo, sino con un intervalo `[inicio, fin)`. Cambiar el estado de una matrícula no es modificar una celda: es cerrar el intervalo vigente y abrir uno nuevo. La consecuencia es que el grafo conserva, gratis, toda la trayectoria de la matrícula a lo largo del tiempo.

TRIPLETAS

```
(matricula_lucia_bd, estado, vigente,
                     inicio=2026-03-12, fin=2026-04-18)
(matricula_lucia_bd, estado, retirada,
                     inicio=2026-04-18, fin=hoy)
```

La pregunta «¿estaba Lucía matriculada el 1 de abril?» se responde directo: una consulta con `al=2026-04-01` filtra por vigencia y devuelve *vigente*. La misma consulta con `al=2026-05-01` devuelve *retirada*. Sin tabla histórica auxiliar, sin esfuerzo extra. La matrícula nunca se borra: cambia de estado y el historial queda fechado para siempre.

Caso 4 · Una persona, muchos roles

Una verdad incómoda del mundo académico: las personas no son una sola cosa. Renata Ferro es estudiante de quinto año *y*, al mismo tiempo, jefa de práctica del curso de Programación. En unos años será docente titular; más adelante, directora de tesis; quizá, decana. Es **la misma persona** (un solo individuo en `Q`), pero ocupa roles distintos en situaciones distintas.

En un sistema tradicional esto se vuelve un dolor de cabeza: una tabla `estudiantes`, otra `docentes`, claves cruzadas y un proceso de sincronización para el día en que un estudiante se convierte en profesor. En WQuestions el problema **no existe**, y la razón es una decisión de diseño que ya conoces:

D5 AGENCIA CONTEXTUAL

El rol de agente lo determina el verbo de la situación, no un «tipo» fijo de la persona. Renata es `renata_q`: aparece como agente en una matrícula (rol *estudiante*), como agente en una asignación docente (rol *jefa de práctica*), como agente en una tesis (rol *tesista*). El sistema no distingue entre tipos de personas; distingue entre tipos de situaciones. Cada rol vive en su situación.

TRIPLETAS

```
# Renata como estudiante
(matricula_renata_ml, agente, renata_q)
(matricula_renata_ml, tema,   cc_420_aprendizaje)

# Renata como jefa de práctica
(asignacion_jp_2026, agente, renata_q)
(asignacion_jp_2026, rol,    rol_jefe_practica)     # ← rol de dominio (K)
(asignacion_jp_2026, tema,   cc_101_progra)

# Renata como tesista
(tesis_renata, agente, renata_q)
(tesis_renata, instancia_de, defensa_tesis)
```

Cuando alguien pregunta «¿qué hizo Renata este año?», la consulta es una sola: trae todas las situaciones donde `renata_q` sea `agente` y proyecta su `instancia_de`. La respuesta sale del grafo de forma natural y abarca lo académico, lo docente y lo

investigativo en una sola lista. No hubo que unir tres tablas de tres «tipos» de persona, porque nunca hubo tres tipos: hubo una persona y tres situaciones.

Caso 5 · La cadena nota → reclamo → rectificación

Llegamos al caso emblemático de la trazabilidad académica. Lucía rinde el examen final de *Cálculo I* el 15 de julio. La Dra. Bravo corrige y le pone 12. Lucía revisa, no está de acuerdo con la corrección de un ejercicio y presenta un reclamo formal el 20 de julio. La docente lo revisa, le da la razón y el 25 de julio rectifica la nota a 15.

En un sistema tradicional, la nota se **sobrescribe**. El historial (si existe) vive en una tabla auxiliar que casi nadie consulta. Cuando años después alguien quiere reconstruir el incidente (un decano que atiende una queja, un comité de calidad que audita), el rastro está borroso. En WQuestions, las tres situaciones **conviven** en el grafo, y la causa que las encadena queda explícita.

Aquí entra en juego una decisión que el libro ya estableció: el «por qué» no es un eje, sino una relación entre hechos, repartida en cuatro cables distintos.

> **D7 EL PORQUÉ, REPARTIDO EN CUATRO CABLES**
>
> No hay un eje «por qué»: la causalidad se expresa con `causado_por`, `motivado_por`, `con_finalidad` y `justificado_por`. La rectificación está `motivado_por` el reclamo (esa fue su razón) y `rectifica` la evaluación original (ese es su blanco). La graduación, más adelante, estará `justificado_por` la defensa. El audit trail no es una tabla aparte: es la propia topología del grafo.

TRIPLETAS

```
# 1. la evaluación inicial (situación reificada)
(eval_lucia_calc) ∈ O
  instancia_de: evaluar_examen
  agente:       dra_bravo
  paciente:     lucia_q
  tema:         ma_120_calculo
  momento:      2026-07-15

# 2. el reclamo: otra situación, agente = Lucía
(reclamo_lucia) ∈ O
  instancia_de: reclamar_nota
  agente:       lucia_q
  tema:         eval_lucia_calc
  momento:      2026-07-20

# 3. la rectificación: causa explícita y blanco explícito
(rectif_lucia) ∈ O
  instancia_de:  rectificar_nota
  agente:        dra_bravo
  motivado_por:  reclamo_lucia        # ← por qué se hizo
  rectifica:     eval_lucia_calc      # ← qué corrige
  momento:       2026-07-25
```

¿Y la nota vigente? Cambia con la misma vigencia D6 del Caso 3. La calificación era 12 entre el 15 y el 25 de julio; desde el 25 es 15. Las dos versiones existen, fechadas:

TRIPLETAS

```
(eval_lucia_calc, nota_vigente, n_12,
                  inicio=2026-07-15, fin=2026-07-25)
```

```
(eval_lucia_calc, nota_vigente, n_15,
                  inicio=2026-07-25, fin=hoy)
```

La consulta con `al=2026-07-18` devuelve 12; con `al=2026-07-30`, devuelve 15. El reclamo sigue en el grafo, consultable años después, y la cadena causal está a la vista: la rectificación está `motivado_por` el reclamo y `rectifica` la evaluación original. Si dentro de cinco años alguien revisa el caso de Lucía, el grafo le entrega la historia completa sin gimnasia alguna.

LA TRAMPA DEL `UPDATE` EN LAS NOTAS

Corregir una nota tienta a ejecutar `UPDATE notas SET valor=15 WHERE id=…` . Es la decisión que destruye la trazabilidad: al actualizar la celda, se borra la prueba de que alguna vez la nota fue 12 y de que un reclamo la cambió. El reflejo más natural del programador es, también aquí, el peor enemigo de la institución. El modelo lo previene: la nota anterior no se modifica, se le cierra el intervalo de vigencia y queda en el grafo.

Caso 6 · Una defensa de tesis con director y jurado

El último escenario es la defensa de tesis de Renata. Una tesis es una situación de larga duración (un año, en su caso) con varios participantes que no juegan el mismo papel:

La **tesista** es Renata, la agente principal. El **director** es el Dr. Acuña, un rol específico del dominio. Y el **jurado** lo componen tres personas con funciones distintas: la Decana Ortiz como presidenta, la Dra. Bravo como vocal y el Dr. Lima como secretario. En un sistema tradicional, todo esto pide una tabla `tesis_participantes` con una columna discriminadora de `tipo`, y cada consulta tiene que filtrar por ese código.

En WQuestions, **cada rol es una tripleta separada**. No hay tabla puente ni columna discriminadora: el rol *es* el nombre del cable. Preguntar «¿quién presidió el jurado de Renata?» es leer un solo enlace, `jurado_presidente`, sin filtrar ninguna columna de tipo. La forma del dato coincide con la forma de la pregunta.

TRIPLETAS

```
(tesis_renata,  agente,              renata_q)
(tesis_renata,  director_tesis,      dr_acuna)
(defensa_renata, parte_de,           tesis_renata)
(defensa_renata, jurado_presidente,  decana_ortiz)
(defensa_renata, jurado_vocal,       dra_bravo)
(defensa_renata, jurado_secretario,  dr_lima)
(defensa_renata, momento,            2026-12-04)

# la graduación: justificada por la defensa exitosa
(graduacion_renata, instancia_de,    graduar)
(graduacion_renata, agente,          renata_q)
(graduacion_renata, justificado_por, defensa_renata)
```

Roles como `director_tesis`, `jurado_presidente`, `jurado_vocal` y `jurado_secretario` no figuran en el catálogo canónico (ese catálogo invisible que vimos al hablar del lexicon). No importa: la política liberal del modelo los acepta sin protestar, y mientras el dominio universitario los use de forma consistente, el motor opera con ellos como si fueran canónicos. La graduación, por su parte, queda `justificado_por` la defensa: Renata se graduó *porque* defendió con éxito su tesis. El grafo deja la cadena causal escrita, sin un solo campo añadido al motor.

> *En una universidad, un expediente no es una foto del presente: es la*

película completa de una vida que el tiempo no debería poder reescribir.

LA LECCIÓN DEL EXPEDIENTE ACADÉMICO

Del expediente al tablero del rectorado

Modelar la matrícula de Tomás y la malla que arrastra *Aprendizaje de Máquina* es resolver un caso. Pero el rectorado vive en el agregado: cuántos estudiantes siguen vigentes en cada carrera, qué curso acumula la tasa de desaprobación más alta, cuántos egresan dentro de los cinco años. Ninguna de esas preguntas pide una tabla nueva. La inscripción de cada alumno ya es una situación de tipo `matricula` con su carrera y su estado; la estadística del semestre es contar esas situaciones, agrupadas por la coordenada que el decano necesite.

PYTHON

```
# Matrículas vigentes en una carrera — un corte; el tablero recorre todas las carreras
count(u, Pattern(fixed={"carrera": u.ind("carrera_cc"), "estado": u.ind("vigente")},
                 type_constraint=u.ind("matricula")))
```

El antes y el después: del esquema fragmentado al grafo único

Pongamos las dos arquitecturas frente a frente con la pregunta que abrió el capítulo, ahora en su forma más exigente: *¿qué cadena de prerrequisitos arrastra «Aprendizaje de Máquina», cuáles de ellos ya aprobó Tomás y cuál le falta?*

Antes, en el modelo relacional. La información se reparte entre `alumnos`, `cursos`, `prerequisitos`, `matriculas` y `notas`. Para expandir el árbol de dependencias de un curso hay que escribir un `WITH RECURSIVE` sobre `prerequisitos`, y luego cruzarlo con `matriculas` y `notas` mediante varios *joins*. Cada vez que la malla cambia (y cambia cada pocos años), la consulta hay que revisarla con lupa.

SQL

```
-- Antes: el árbol de prerrequisitos exige SQL recursivo.
WITH RECURSIVE cadena(curso) AS (
    SELECT prereq FROM prerequisitos WHERE curso = 'cc_420_aprendizaje'
  UNION
    SELECT p.prereq
    FROM prerequisitos p
    JOIN cadena c ON p.curso = c.curso
)
SELECT c.curso,
       CASE WHEN n.estado = 'aprobado' THEN 'ok' ELSE 'FALTA' END
FROM cadena c
LEFT JOIN notas n
       ON n.curso = c.curso AND n.alumno = 'tomas_q';
-- ...y todavía falta cruzar vigencias para la malla histórica.
```

Después, en WQuestions. La misma información vive como un solo grafo. Cada prerrequisito es una tripleta `requiere_prereq`; cada curso aprobado es un nodo alcanzable desde Tomás. La cadena completa de dependencias de *Aprendizaje de Máquina* se obtiene con un recorrido hacia atrás sobre las flechas `requiere_prereq` (sin SQL recursivo, sin procedimiento almacenado). Determinar qué le falta es restar dos conjuntos: la cadena de requisitos menos su subgrafo de aprobados. Y el grafo no distingue entre malla vigente y malla histórica: la bitemporalidad ya lo cubre, así que la misma consulta sirve para el plan de hoy y para el de hace diez años.

Qué quedó probado en este capítulo

La universidad es, probablemente, el dominio donde el **timeline largo** y los **grafos de dependencia** dejan de ser un detalle accesorio y se vuelven el problema central. El modelo absorbió ambas cosas con su maquinaria estándar, sin un solo añadido al motor:

EL VEREDICTO DEL DOMINIO DE LA FORMA

Dependencias como grafo. La malla curricular es un DAG, y cada prerrequisito un hecho atómico. La pregunta «¿qué puedo tomar?» es un recorrido sobre el grafo que, de paso, devuelve lo que falta —no un sí o no mudo.

Historial intacto de por vida. La vigencia (D6) deja matrículas, notas y estados con su trayectoria completa durante toda la vida académica del egresado. La trampa del `UPDATE` nunca llega a ser una operación disponible.

Una persona, muchos roles. La agencia contextual (D5) permite que Renata sea estudiante, jefa de práctica y tesista a la vez: el grafo no distingue tipos de personas, sino tipos de situaciones.

Cadenas causales explícitas. El porqué repartido en cables (D7) deja a la vista que una rectificación se hizo por un reclamo, o que una graduación se justifica por una defensa. El audit trail es la topología misma del grafo.

Y, una vez más, **el motor no creció ni una línea** para absorber el dominio. Lo único que creció fue el lexicon: un puñado de verbos (`matricular`, `dictar`, `evaluar`, `reclamar`, `rectificar_nota`, `graduar`, `defender_tesis`) con sus signaturas, más un dialecto de dominio que mapea «estudiante», «docente», «tesista» o «director de tesis» a sus roles. En el próximo capítulo dejamos el campus y cruzamos a lo público: una municipalidad, donde las fechas de vigencia de una norma deciden la vida de un barrio entero.

22

Una municipalidad

Hasta ahora, lo que importaba era lo que pasaba: el evento. En un gobierno local, nada pasa «porque sí». Cada trámite cuelga de una norma que lo habilita, y esa norma de otra. Aquí el «por qué» deja de ser un adorno y se vuelve la columna vertebral.

Carla Ferreyra entra a la oficina de atención al vecino un lunes a las nueve y diez, con una carpeta bajo el brazo. Quiere abrir una pequeña tienda de bicicletas en la esquina de su barrio. La empleada que la atiende no le pregunta, de entrada, qué vende ni cuánto invierte: le pide su documento, el plano del local y el certificado de zonificación. ¿Por qué esos tres papeles y no otros? Porque una ordenanza (la 142, de micromovilidad) exige que toda licencia de funcionamiento de un comercio de vehículos livianos acredite esos requisitos. Carla no lo sabe, pero acaba de tocar el primer eslabón de una cadena que va de una norma a un trámite, y de ese trámite a cada uno de sus requisitos. Reconstruir esa cadena (y poder responder, años después, *bajo qué artículo se le pidió cada cosa*) es el examen de este capítulo.

> **EL GIRO DEL CAPÍTULO**
> En el spa, el taxi o el banco, el centro de gravedad era la operación. En una municipalidad, ningún acto se entiende sin la norma que lo autoriza. El protagonista, por primera vez, no es el evento sino su *fundamento*.

En los dominios anteriores de esta parte, lo predominante era lo que ocurría: una venta, un viaje, una transferencia. El hecho mandaba. Una municipalidad introduce una tensión distinta. Aquí, lo que sucede no se sostiene sin la regla que le da pie. Una licencia de funcionamiento no se emite por voluntad de un funcionario: se emite **porque una ordenanza fija los requisitos y alguien verificó que se cumplían**. Una multa de tránsito no se aplica al capricho del inspector: se aplica **porque un artículo la habilita y el conductor cometió el supuesto que ese artículo describe**. Una resolución no revoca una multa por gusto: la revoca **porque un recurso ciudadano fue declarado fundado conforme al procedimiento**.

En todos esos casos, el «porque» del Estado tiene dos caras al mismo tiempo: una cara *fáctica* (qué ocurrió en el mundo) y una cara *normativa* (qué regla lo autoriza). Un sistema municipal serio tiene que registrar las dos y poder responder por separado a dos preguntas que suenan iguales pero no lo son:

¿ POR QUÉ SE APLICÓ ESTA MULTA?

Porque un vehículo estaba estacionado en zona prohibida a las dos y media de la tarde. Es el **hecho** que la disparó: su causa en el mundo.

§ BAJO QUÉ AUTORIDAD LEGAL SE APLICÓ?

Porque el artículo 7 de la ordenanza 142 habilita la sanción para ese supuesto. Es el **fundamento**: la norma que le da validez.

WQuestions separa esas dos caras con precisión quirúrgica, y lo hace con un reparto que ya conoces. Como argumentó el capítulo sobre el «por qué», no existe un eje «por qué»: el porqué se divide en cuatro cables distintos. Dos de ellos hacen aquí todo el trabajo: `causado_por` apunta a la causa fáctica, y `justificado_por` apunta a la norma. Conviven sin pisarse, y —lo decisivo— permiten reconstruir cualquier acto del Estado hacia atrás hasta su fundamento legal con un par de saltos en el grafo.

> **RECORDATORIO · D7**

La cuádruple partición del «por qué» (`causado_por`, `motivado_por`, `con_finalidad` y `justificado_por`) se enunció en el capítulo dedicado a por qué el porqué no es una coordenada. Aquí, `justificado_por` pasa a primer plano: es la fibra con la que se teje lo jurídico.

LO QUE PONE A PRUEBA ESTE CAPÍTULO

Que el modelo represente **lo normativo** sin extensiones especiales. La apuesta es fuerte: las ordenanzas y sus artículos viven en `0` como cualquier otra entidad, los trámites son situaciones reificadas, y un único cable (`justificado_por`) encadena norma, trámite y requisito en una estructura que se puede recorrer, auditar e impugnar. Ni una tabla nueva, ni un eje nuevo.

El barrio en siete coordenadas

El primer trabajo de quien modela un dominio es repartir sus entidades en las coordenadas correctas. Una municipalidad es generosa en variedad: tiene personas, papeles, territorio, fechas, montos y estados, y todos caben con naturalidad en los siete ejes.

El gobierno local, repartido en sus siete coordenadas

Figura 22.1. El dominio municipal sobre los siete ejes. Repara en dos detalles. La propia municipalidad vive en `0`, porque es una persona jurídica que dicta normas, emite licencias y sanciona: un agente de pleno derecho. Y las *ordenanzas*, que un sistema tradicional escondería en un repositorio de documentos aparte, aquí viven en `0` junto a las solicitudes y las multas, con identidad propia y trazables como cualquier otra entidad.

Las decisiones de diseño que más se ejercitan aquí ya están todas sobre la mesa, enunciadas en capítulos previos. Vale la pena nombrarlas, porque este dominio las pone a trabajar juntas: **D7**, las dos relaciones del «por qué» operando en paralelo (una fáctica, una normativa); **D4**, cada acto administrativo reificado como una situación con identidad propia; **D6**, los estados de un expediente que cambian en el tiempo y se preservan; y **D8**, la política liberal del lexicon, que admite roles específicos del dominio sin tocar el catálogo canónico.

Las ordenanzas viven en el grafo

El punto de partida, y el más sorprendente para quien viene del mundo relacional, es este: **una ordenanza no es un PDF guardado en una carpeta; es una entidad**. Vive en `0` con sus atributos (número, fecha de publicación, autoridad que la dictó,

materia que regula) y sus artículos cuelgan de ella por `parte_de`, exactamente la misma relación que une una línea con su venta en el spa, o una pieza con su ensamblaje en un ERP.

Tomemos la ordenanza 142, sobre micromovilidad (scooters, bicicletas y comercios afines), que será nuestro hilo conductor. La municipalidad la dictó, fija una fecha de publicación y, como toda norma, entra en vigor más tarde: a los treinta días, tal como la propia ordenanza dispone.

TRIPLETAS

```
(ordenanza_142) ∈ O
  instancia_de     : ordenanza_municipal      # K · qué clase de norma es
  numero           : 142                       # N · su número
  emitida_por      : municipalidad_lomas       # Q · qué agente la dictó
  materia          : micromovilidad            # K · sobre qué regula
  fecha_publicacion: 2026-02-10                # T · cuándo se publicó
  entra_en_vigor   : 2026-03-12                # T · a los 30 días
  estatus_factual  : vigente                   # K · su estado
```

Cada artículo es un objeto en `O` por derecho propio, vinculado a su ordenanza y —esto es lo que vuelve la estructura interesante — capaz de *habilitar* trámites y *exigir* requisitos. El artículo 4 habilita la licencia de funcionamiento para comercios de vehículos livianos; el artículo 7 habilita la sanción por estacionar un vehículo de reparto en zona prohibida.

TRIPLETAS

```
(art_4_ord_142, parte_de,      ordenanza_142)        # O→O · es parte de la norma
(art_4_ord_142, habilita,      tramite_licencia_micromov)  # qué trámite autoriza
(art_4_ord_142, exige,         requisito_zonificacion)     # qué requisito impone
(art_4_ord_142, exige,         requisito_plano_local)

(art_7_ord_142, parte_de,      ordenanza_142)
(art_7_ord_142, habilita,      sancion_estacionamiento)    # qué sanción autoriza
(art_7_ord_142, monto_base,    320)                        # N · multa, en USD
```

VIGENCIA · D6

Si mañana se modifica la ordenanza 142, la nueva versión es *otra* entidad, con su propia identidad, enlazada a la anterior por una relación de reemplazo. La regla de vigencia garantiza que las multas dictadas bajo la versión vieja sigan apuntando al texto que de verdad las habilitó: la verdad histórica no se reescribe.

Fíjate en lo que acabamos de construir sin darnos cuenta. Tenemos un *artículo* que `habilita` un *trámite* y que `exige` unos *requisitos*. Eso es ya, en germen, una cadena normativa. Y las cadenas, en este modelo, no se declaran en una tabla aparte: emergen de encadenar cables del eje `M`.

La cadena normativa

Aquí está el corazón del capítulo. Cuando Carla pide su licencia, la empleada no inventa los requisitos: los lee de la cadena que va de la norma al trámite y del trámite a cada requisito. Esa cadena es un camino en el grafo, y cada arista es un cable de `M` con su firma y su color de eje.

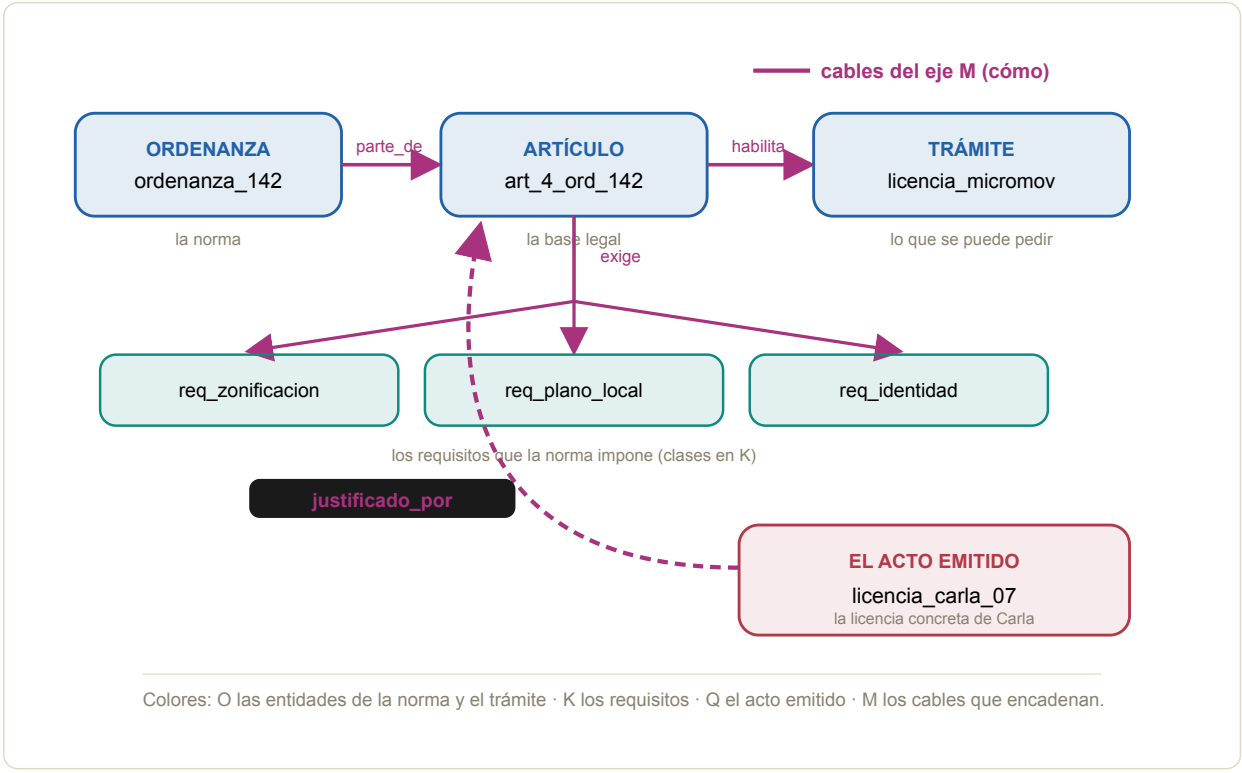
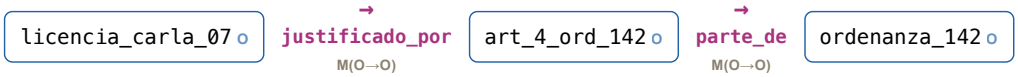


Figura 22.2. La cadena normativa. De izquierda a derecha, la *norma* contiene el *artículo*, que `habilita` el *trámite* y `exige` sus *requisitos*. Y, cuando la licencia concreta de Carla se emite, el cable `justificado_por` (la flecha punteada) cierra el círculo apuntando de vuelta al artículo. Reconstruir *bajo qué norma se le pidió cada cosa* es, literalmente, recorrer estas aristas hacia atrás.

Conviene leer la misma estructura como una tripleta, que es como el modelo la guarda. El eslabón clave (el que ata el acto a la norma) se lee así:



CADENA NORMATIVA

Camino en el grafo que conecta un acto administrativo con la norma que lo fundamenta, eslabón a eslabón: la licencia está `justificado_por` un artículo, el artículo es `parte_de` una ordenanza, la ordenanza fue `emitida_por` la municipalidad. No es una tabla ni un campo: es un *recorrido*. Su longitud puede crecer (un artículo puede a su vez remitir a una ley superior) sin que el modelo cambie: solo se añaden eslabones del mismo tipo.

El territorio, jerarquía de lugares

Antes de seguir los trámites de Carla, hace falta ubicarlos. Una municipalidad gestiona territorio, y el territorio se anida: la ciudad contiene distritos, los distritos barrios, los barrios manzanas, las manzanas lotes. Cada uno es un punto en `L`, y la jerarquía se establece con un rol de dominio, `dentro_de`:

TRIPLETAS

```
(distrito_norte, dentro_de, ciudad_capital)    # L→L
(barrio_lomas,   dentro_de, distrito_norte)
(manzana_19,     dentro_de, barrio_lomas)
(lote_19,        dentro_de, manzana_19)         # aquí abrirá Carla su tienda
```

POLÍTICA LIBERAL · D8

> El cable canónico `parte_de` está declarado como `0→0`. Para anidar *lugares* (`L→L`) usamos `dentro_de`, un rol de dominio que la política liberal admite sin validarlo contra el catálogo: el motor no protesta y la semántica queda nítida para humanos y para un LLM. La interfaz es el lexicon, no el catálogo.

La cadena territorial se recorre recursivamente. La pregunta *«¿dónde queda exactamente el lote 19?»* se responde siguiendo `dentro_de` hacia arriba: `lote_19 → manzana_19 → barrio_lomas → distrito_norte → ciudad_capital`, en un solo recorrido y sin un solo `JOIN`. Y, como veremos, el mismo lote es a la vez el *lugar* de la futura tienda y el *tema* de los trámites que la habilitan: una misma entidad territorial se cruza con la cadena normativa.

El trámite, paso a paso, como situaciones encadenadas

Volvamos a Carla. Su licencia no es un instante: es un proceso de tres actos administrativos encadenados, y cada uno (fiel a la idea de reificar lo que merece identidad propia) es una situación en `O` con su agente, su momento, su lugar y su resultado.

1. **Solicitud.** Carla, en nombre de su comercio, solicita la licencia indicando el lote donde quiere operar.
2. **Inspección.** Un inspector municipal visita el local y certifica que cumple los requisitos que la norma exige.
3. **Emisión.** La municipalidad emite la licencia, válida por un año, fundada en el artículo.

Los tres actos se enlazan con cables del «por qué». La inspección está `motivado_por` la solicitud; la emisión está `motivado_por` tanto la solicitud como la inspección (porque `motivado_por` admite varios valores); y la emisión cierra con el cable decisivo, `justificado_por`, apuntando a la norma. En el prototipo, armar la emisión no exige redactar las tripletas a mano: una llamada consulta el lexicon, desambigua el verbo, crea la situación y enchufa los cables, validando cada uno.

PYTHON

```
licencia = ingest_situation(u, lex, "emitir",
    roles={
        "agente":         municipalidad_lomas,
        "tipo_emitido":   category("licencia_funcionamiento"),
        "beneficiario":   comercio_carla,
        "lugar_de":       lote_19,
        "momento":        at("2026-04-06T16:00"),
        "justificado_por": art_4_ord_142,        # ← la base legal
    },
    extra={
        "valido_desde": at("2026-04-10T00:00"),
        "valido_hasta": at("2027-04-09T23:59"),  # un año
    },
    sit_id="licencia_carla_07",
)
u.assert_fact(licencia, "motivado_por", solicitud_carla)   # multi-valor
u.assert_fact(licencia, "motivado_por", inspeccion_carla)
```

Lo notable es la nitidez con que el grafo responde dos preguntas que un sistema clásico mezcla. *«¿Bajo qué norma se emitió la licencia de Carla?»* se resuelve con un único salto siguiendo `justificado_por`: el artículo 4. Y *«¿qué motivó la emisión?»* devuelve dos respuestas (la solicitud y la inspección) porque el cable es multi-valor. Dos preguntas distintas, dos cables distintos, dos respuestas limpias.

Mientras tanto, el estado de la solicitud cambia con el tiempo, y esos cambios se preservan en lugar de sobrescribirse, fieles a la regla de vigencia. La solicitud estuvo `en_revision` entre el 6 de marzo y el 6 de abril, y quedó `aprobada` desde entonces. Ambos hechos conviven, y la pregunta *«¿en qué estado estaba la solicitud el 20 de marzo?»* tiene una respuesta exacta.

TRIPLE

```
(solicitud_carla, estatus_factual, en_revision)   ⟦2026-03-06 … 2026-04-06⟧
(solicitud_carla, estatus_factual, aprobada)      ⟦2026-04-06 … ∞⟧
```

```
# ↑ dos hechos vigentes en rangos distintos; ninguno borra al otro (D6)
```

Una denuncia abre un expediente

Pasa un mes. Bruno Salazar, vecino de la misma manzana, denuncia que la tienda de Carla recibe repartos a deshora y bloquea la vereda con cajas. La denuncia es, otra vez, una situación: tiene un `agente` (Bruno), un `paciente` (el comercio denunciado) y un `tema` que reifica el hecho específico que se reporta, el bloqueo de la vereda.

PYTHON

```
denuncia = ingest_situation(u, lex, "denunciar",
    roles={
        "agente":   bruno,
        "tema":     hecho_bloqueo_vereda,     # el bloqueo reificado
        "paciente": comercio_carla,
        "lugar_de": lote_19,
        "momento":  at("2026-05-18T09:30"),
    },
    sit_id="denuncia_bruno_03",
)
```

La denuncia dispara un **expediente administrativo** (otra entidad en `O`) que agrupa, por `parte_de`, todas las diligencias que el municipio realice para procesarla. La inspección de verificación que el inspector hace cuatro días después es `parte_de` el expediente; y si más tarde se suman citación, descargo y resolución, todas viven dentro del mismo expediente.

TRIPLE

```
(expediente_2026_214) ∈ O
  instancia_de  : expediente_administrativo
  motivado_por  : denuncia_bruno_03          # nació de la denuncia

(insp_verificacion, parte_de, expediente_2026_214)   # una diligencia
(citacion_carla,    parte_de, expediente_2026_214)   # otra diligencia
```

El expediente como entidad articuladora no es un patrón nuevo: es el mismo que reúne un viaje de taxi con sus etapas, o una transferencia bancaria con sus pasos. Una situación superior que junta todos sus actos secundarios. La pregunta *«¿qué diligencias hubo en este expediente?»* es una proyección directa por `parte_de`.

El caso paradigmático: la multa con doble «por qué»

Y ahora, la escena que justifica el capítulo entero. Mientras la denuncia sigue su curso, ocurre algo en la avenida. El conductor del furgón de reparto de la tienda, Iván Cardozo, estaciona en zona prohibida sobre la avenida del Parque un martes a las dos y media de la tarde. Tres minutos después, una inspectora de tránsito le aplica una multa de 320 dólares.

El modelo registra la cadena entera. La **infracción** se reifica como una situación con su tema (el furgón), su lugar, su momento y su tipo categórico. Y la **multa** se reifica como otra situación con su agente, su paciente, su monto, y —esto es lo crucial— **dos relaciones del «por qué» a la vez**:

PYTHON

```
multa = ingest_situation(u, lex, "multar",
    roles={
        "agente":          inspectora_transito,
```

```
        "paciente":        ivan,
        "monto":           n(320, "USD"),
        "lugar_de":        av_del_parque,
        "momento":         at("2026-06-09T14:33"),
        "causado_por":     infraccion_furgon,    # ← CAUSA FÁCTICA  (D7)
        "justificado_por": art_7_ord_142,        # ← AUTORIDAD NORMATIVA (D7)
    },
    sit_id="multa_ivan_88",
)
```

Esta es la forma del modelo que solo la cuádruple partición del «por qué» hace posible. La multa **es causada por** la infracción (sin el furgón mal estacionado no habría razón para sancionar) y **es justificada por** el artículo 7 (sin él, la inspectora no tendría base legal para hacerlo). Las dos afirmaciones son verdaderas a la vez y dicen cosas distintas. Visto como tripletas, el doble fundamento se lee de un golpe:

TRIPLE

```
(multa_ivan_88, causado_por,     infraccion_furgon)   # M(O→O) · el hecho
(multa_ivan_88, justificado_por, art_7_ord_142)       # M(O→O) · la norma
# ↑ mismo «por qué», dos cables: uno mira al mundo, otro mira a la ley
```

LO QUE EL CAMPO «MOTIVO» DESTRUYE

En un sistema tradicional, esto se aplasta en una sola columna `motivo`: un texto libre tipo «estacionamiento prohibido, art. 7». Esa cadena de caracteres mezcla el hecho con la norma, no se puede consultar limpiamente y —lo más grave— el ciudadano no puede impugnar *solo* el fundamento legal sin tocar el hecho, ni viceversa. El doble cable de D7 mantiene ambas caras separadas, consultables e impugnables por separado.

El recurso, y un acto que no se borra

Iván no está de acuerdo. Sostiene que la señalización de la zona estaba tapada por un árbol y no era visible. El 16 de junio presenta un recurso de reconsideración, que se modela como una situación nueva apuntando a la multa que impugna:

TRIPLE

```
(recurso_ivan_01) ∈ O
  instancia_de : recurso_reconsideracion
  agente       : ivan                          # Q · quién recurre
  tema         : multa_ivan_88                 # O · la multa que impugna
  motivado_por : alegato_senalizacion          # por qué recurre
  momento      : 2026-06-16T11:00              # T
```

Tres semanas más tarde, el alcalde resuelve. Declara el recurso **fundado** (la señalización, en efecto, no era visible), lo que deja la multa sin efecto. La resolución es, como todo lo demás, una situación; lleva su propia justificación normativa (el artículo del procedimiento administrativo que faculta a resolver recursos) y dos consecuencias ocurren en paralelo:

PYTHON

```
resolucion = ingest_situation(u, lex, "resolver",
    roles={
        "agente":          alcalde_lomas,
        "tema":            recurso_ivan_01,
        "conclusion":      category("fundado"),
```

```
        "momento":         at("2026-07-07T17:00"),
        "justificado_por": art_15_proc_admin,   # base legal de la resolución
    },
    sit_id="resolucion_ivan_01",
)
u.assert_fact(resolucion, "rectifica", multa_ivan_88)        # trazabilidad
u.assert_fact(multa_ivan_88, "estatus_factual", u.ind("revocada"),
              valid_from=at("2026-07-07T17:00"))             # vigencia (D6)
```

Dos cosas pasan al unísono. La resolución `rectifica` la multa (trazabilidad explícita: este acto modificó aquel acto), y el estado de la multa cambia a `revocada` desde el instante exacto de la resolución. Pero —y aquí está la promesa— la multa **no se borra**. Sigue en el grafo, sigue siendo consultable, sigue formando parte del historial de Iván. Lo único que cambió es su validez vigente. Gracias a la regla de vigencia, todavía podemos responder con exactitud: *«¿estaba esa multa vigente el 1 de julio?»* —sí— y *«¿y el 10 de julio?»* —no, ya revocada—.

Esta es la diferencia práctica entre un grafo de hechos y una tabla que se sobrescribe. En la tabla, cuando la multa se revoca, alguien actualiza la fila: `estado = 'revocada'`, y la verdad anterior desaparece. Si meses después un tribunal pregunta qué decía esa multa el día en que se aplicó, la respuesta honesta es «ya no se sabe».

En WQuestions, en cambio, los dos hechos coexisten con sus rangos de vigencia. El acto original permanece intacto, con su monto, su fundamento legal y su fecha; el acto que lo revoca también permanece, con el suyo. La auditoría no recupera un fósil: lee un registro que nunca se mutiló.

Cómo se reconstruye un acto del Estado

Reunamos lo construido en una sola pregunta, la que de verdad le importa a un órgano de control: *«Dame todo lo que sostiene esta multa.»* Con los cables que ya tendimos, la respuesta es un puñado de saltos de grafo, no un proyecto de minería de datos.

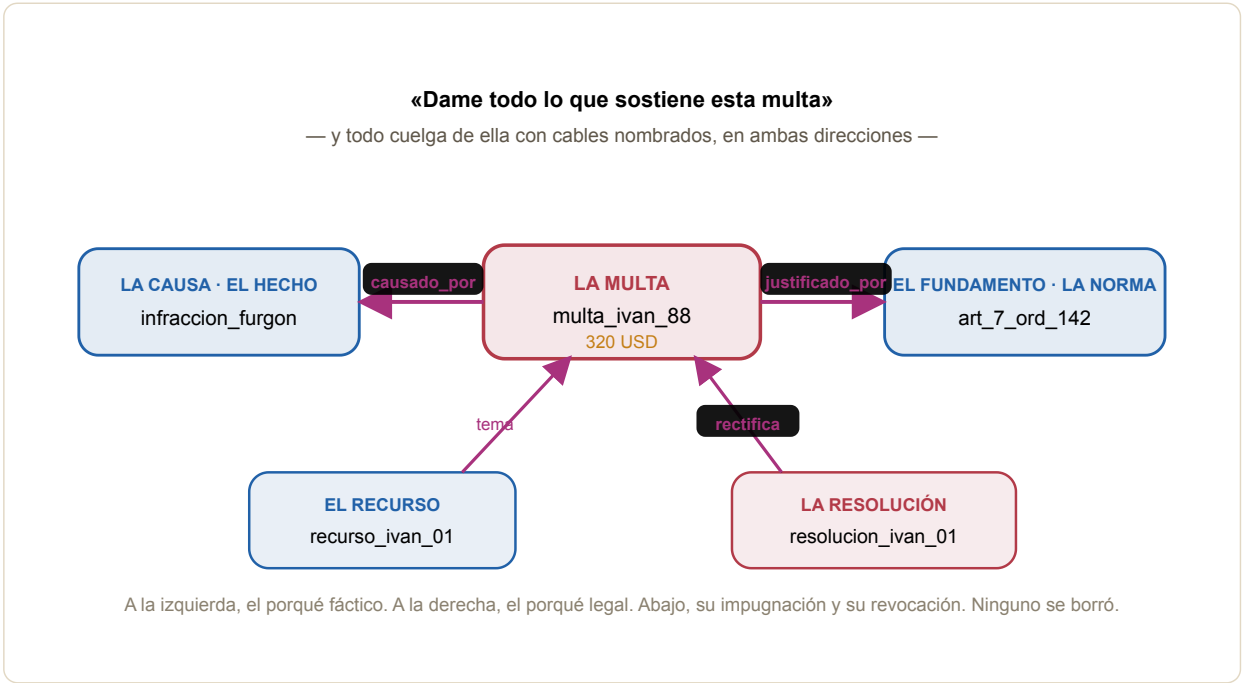


Figura 22.3. La multa al centro, reconstruida en todas sus direcciones. Hacia la izquierda, `causado_por` da la causa fáctica; hacia la derecha, `justificado_por` da el fundamento legal (los dos «por qué» de D7, verdaderos a la vez). Abajo, el recurso que la impugnó y la resolución que la `rectifica`. Cualquier auditor responde *«¿qué norma autorizó esto?»* con la misma facilidad que responde *«¿quién lo hizo?»*: un salto de grafo.

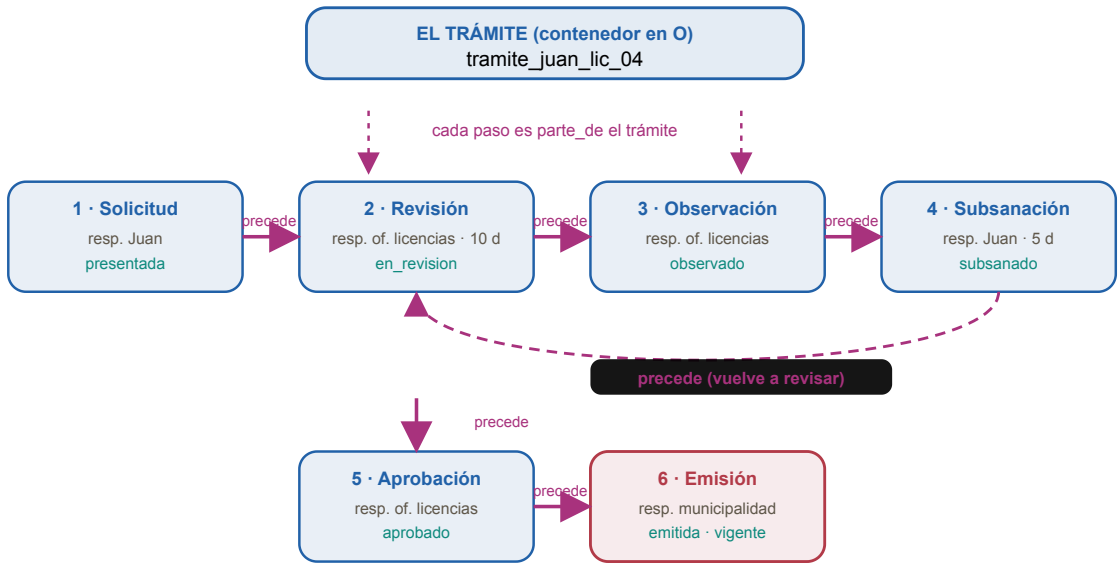
El trámite de punta a punta, como máquina de estados

Hasta aquí miramos el trámite de Carla por fuera: tres actos enlazados y una licencia al final. Pero quien trabaja en la ventanilla lo vive por dentro, y por dentro un trámite no es un acto: es una **secuencia de situaciones encadenadas** que avanza paso a paso, a veces retrocede, y solo a veces llega. Para verlo de cerca seguiremos a otro vecino, Juan (el mismo Juan cuya identidad resolvimos al hablar del principio de reconocer a una persona una sola vez), que abre su propio trámite de licencia para un taller de bicicletas en el lote contiguo.

El trámite de Juan es una entidad en `O`, un contenedor con identidad propia que nace el día que él presenta su solicitud. Pero lo interesante no es el contenedor: son sus *pasos*. Cada paso (solicitud, revisión de requisitos, observación, subsanación, aprobación, emisión) se reifica como una situación con cuatro datos que un sistema clásico suele dejar implícitos o repartidos: su **estado** (en qué punto está), su **responsable** (qué agente lo tiene en la mano), su **plazo** (cuántos días tiene para resolverlo) y su **fecha** (cuándo ocurrió de verdad). Reificar cada paso (fiel a la idea de dar identidad propia a lo que merece ser interrogado después) es lo que vuelve el trámite auditable minuto a minuto.

> **POR QUÉ REIFICAR CADA PASO · D4**
> Un paso de trámite cumple de sobra los criterios para convertirse en situación: tiene varios participantes (un responsable, un solicitante), se le harán preguntas en el futuro («¿quién observó esto y qué día?»), cambia de estado y necesita su propio plazo. No es un atributo del trámite: es una entidad por derecho propio.

Los pasos no flotan sueltos. Dos cables del eje `M` les dan estructura. Cada paso es `parte_de` el trámite (la relación de composición que ya une un artículo con su ordenanza) y cada paso `precede` al siguiente, dibujando el orden temporal sin inventar tabla alguna. El resultado es, literalmente, una máquina de estados grabada en el grafo: una cadena que se puede recorrer hacia adelante para saber qué falta, o hacia atrás para reconstruir todo lo que ocurrió.



Cada caja es una situación en O con su estado (K), su responsable (Q) y su plazo (N).

El flujo feliz va de 1 a 6; la observación abre una rama de subsanación que vuelve a la revisión antes de aprobar.
El orden no es una columna: es el cable precede; la pertenencia no es una FK: es el cable parte_de.

Figura 22.4. El trámite de Juan como máquina de estados. Cada paso es una situación reificada que lleva su propio estado en `K`, su responsable en `Q`, su plazo en `N` y su fecha en `T`. El cable `precede` dibuja el orden; el cable `parte_de`, la pertenencia al trámite. Avanzar el trámite es añadir un eslabón; auditarlo es recorrer la cadena. La rama punteada muestra que el modelo no exige un camino recto: una observación devuelve el expediente a revisión sin perder el rastro de por dónde pasó.

Veamos dos pasos como tripletas, para fijar la forma. La revisión es `parte_de` el trámite, la observación `precede` a la subsanación, y cada una carga su responsable, su plazo y su fecha sin aplastarlos en un comentario:

TRIPLE

```
(paso_revision_04) ∈ O
  instancia_de   : paso_tramite
  parte_de       : tramite_juan_lic_04        # O→O · pertenece al trámite
  responsable    : oficina_licencias          # Q · quién lo tiene en la mano
  plazo_dias     : 10                          # N · días para resolver
  momento        : 2026-04-22T09:00            # T · cuándo se inició
  estatus_factual: en_revision                 # K · su estado

(paso_observacion_04, parte_de, tramite_juan_lic_04)
(paso_observacion_04, precede,  paso_subsanacion_04)   # M(O→O) · el orden
(paso_subsanacion_04, responsable, juan)               # Q · ahora la pelota es de Juan
(paso_subsanacion_04, plazo_dias,  5)                  # N · 5 días para subsanar
```

Repara en un detalle que cambia todo para la atención al vecino: cuando una observación se emite, el `responsable` del paso siguiente deja de ser una oficina y pasa a ser Juan. El grafo sabe, en todo momento, *de quién es la pelota*. La pregunta *«¿qué trámites están esperando algo del ciudadano y cuáles esperan algo de la municipalidad?»* deja de ser un reporte artesanal y se vuelve una proyección por el rol `responsable` del paso vigente.

De un trámite a la demora de todos

Recorrimos el trámite de Juan paso a paso. Pero al alcalde nadie le pregunta por `tramite_juan_lic_04`: le preguntan por la demora. Cuántos días, en promedio, tarda una licencia de micromovilidad desde la solicitud hasta la emisión; qué paso es el cuello de botella; si el «una sola vez» de verdad recortó los tiempos este año. Cada trámite ya cargó su `momento` de inicio y su fecha de cierre, paso por paso; el indicador de gestión es promediar esa diferencia sobre todos los trámites de un mismo tipo.

PYTHON

```
# Días promedio de resolución de un tipo de trámite ya emitido — el SLA se mira por tipo
promedio(u, "dias_resolucion", Pattern(fixed={"estatus_factual": u.ind("emitida")},
                                       type_constraint=u.ind("tramite_licencia_micromov")))
```

El expediente, contenedor que todo lo agrupa

El trámite encadena los pasos; el **expediente** los guarda. Es el segundo contenedor reificado del capítulo, y conviene no confundirlo con el trámite: el trámite es el procedimiento (la máquina de estados), mientras que el expediente es el *legajo*, la carpeta única, citable y foliada, que reúne **todo** lo que el procedimiento generó: cada paso, sí, pero también cada documento, cada notificación y cada dictamen. Ya vimos un expediente nacer de una denuncia; aquí lo miramos como pieza central de archivo.

El expediente de Juan es una entidad en `O` con un número de foja (su cita oficial), una oficina que lo custodia y una fecha de apertura. Todo lo demás cuelga de él por `parte_de`, exactamente la misma relación de composición que ya conocemos. La diferencia entre «un paso del trámite» y «un documento del expediente» no exige cables nuevos: basta la clase de cada pieza en `K`.

TRIPLE

```
(expediente_2026_337) ∈ O
  instancia_de  : expediente_administrativo
  numero_foja   : "2026-337"                  # su cita oficial, citable
  custodiado_por: oficina_licencias           # Q · qué oficina lo guarda
  abierto_el    : 2026-04-20                   # T
  agrupa_a      : tramite_juan_lic_04         # O→O · el procedimiento que contiene
```

```
# todo cuelga del mismo expediente por parte_de — pasos y documentos juntos:
(paso_revision_04,    parte_de, expediente_2026_337)   # una situación
(paso_observacion_04, parte_de, expediente_2026_337)   # otra situación
(doc_plano_local,     parte_de, expediente_2026_337)   # un documento
(doc_cert_zonif,      parte_de, expediente_2026_337)   # otro documento
(notif_observacion,   parte_de, expediente_2026_337)   # una notificación
```

CITABLE Y AUDITABLE

Que el expediente tenga identidad y número de foja propios es lo que permite citarlo desde fuera («ver expediente 2026-337») y auditarlo entero con una sola proyección. Un control externo no reconstruye el legajo cruzando cinco tablas: lo lee siguiendo `parte_de` hacia dentro, y obtiene en un solo recorrido pasos, documentos y notificaciones, cada uno con su fecha y su autor.

La auditoría que antes era un proyecto se vuelve una pregunta. *«Dame el expediente completo, en orden»* es la proyección de todo lo que es `parte_de` el expediente, ordenado por su cable `momento`. Y como nada se sobrescribe (fiel a la regla de vigencia), el legajo que se lee hoy es el mismo que existió en cada instante del pasado: el expediente no es una foto del estado actual, es la película entera.

PYTHON

```
# El expediente entero, foliado y en orden cronológico: una proyección.
piezas = u.project(parte_de=expediente_2026_337)        # pasos + documentos + notificaciones
legajo = sorted(piezas, key=lambda p: u.get(p, "momento"))
# → cada pieza con su clase (K), su autor (Q) y su fecha (T). Nada quedó fuera.
```

Interoperabilidad entre oficinas y el principio «una sola vez»

Llegamos al problema que de verdad sufre el ciudadano. El trámite de Juan no lo procesa una sola oficina: la atención al vecino recibe la solicitud, la oficina de zonificación verifica el uso del suelo, la de rentas comprueba que no haya deudas, y la de licencias decide. En casi todos los municipios reales, cada oficina es una isla con su propia base de datos, y el pegamento que las une es… el ciudadano. Es Juan quien lleva el certificado de zonificación de una ventanilla a otra, quien vuelve a presentar su documento en cada oficina, quien pide la constancia de no adeudo y la camina por el pasillo. **El ciudadano es el integrador.** Y es un integrador caro, lento y falible.

EL CIUDADANO COMO MIDDLEWARE

Cuando cada oficina guarda su propia copia de «Juan», pedirle de nuevo un documento que otra oficina ya tiene no es un descuido: es la consecuencia inevitable de no tener una identidad compartida. El vecino se convierte en el bus de datos entre sistemas que no se hablan, fotocopiando y caminando papeles que el Estado ya posee. El principio «una sola vez» (*once-only*) dice exactamente lo contrario: lo que una oficina del Estado ya tiene, ninguna otra debe volver a pedirlo.

Ese principio no se sostiene con buena voluntad: se sostiene sobre la identidad resuelta. Si las cuatro oficinas reconocen al *mismo* Juan (no cuatro registros parecidos, sino una única entidad en `Q` a la que todas apuntan), entonces el certificado de zonificación que Juan entregó en una oficina es, sin más trámite, visible para las otras. Es el principio de reconocer a una persona una sola vez, llevado del expediente clínico al mostrador municipal: la misma idea, otro dominio.

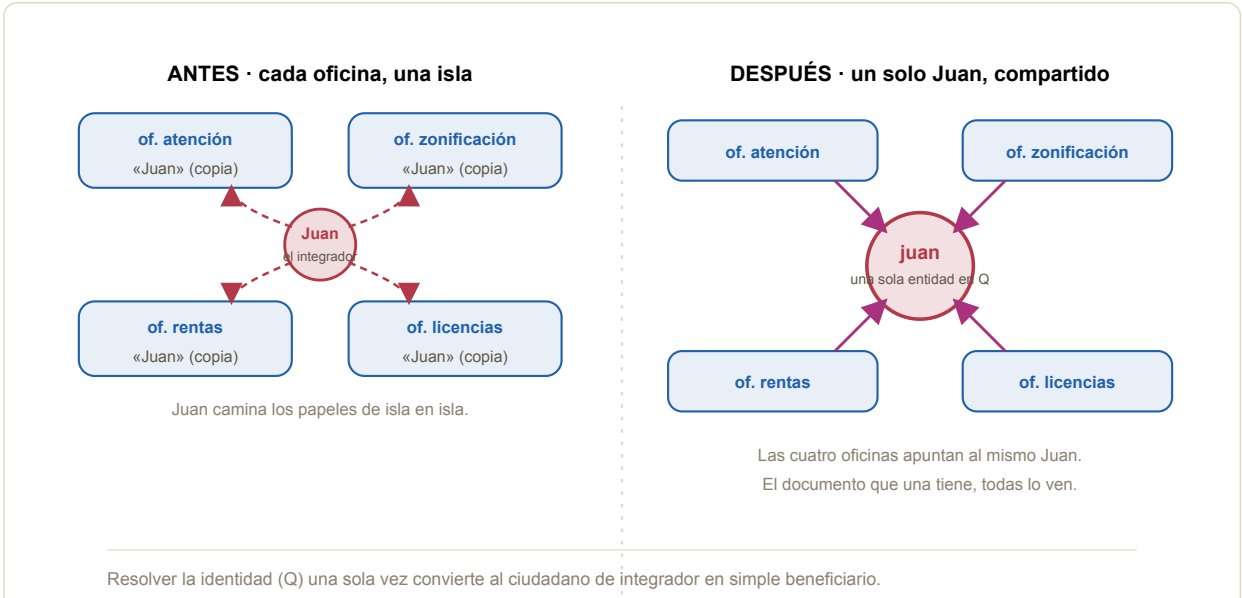


Figura 22.5. El principio «una sola vez», en una imagen. A la izquierda, cada oficina guarda su propia copia de Juan y es él quien camina los papeles: el ciudadano es el integrador. A la derecha, las cuatro oficinas apuntan a una única entidad `juan` en `Q`; el documento que una oficina recibió queda visible para todas, sin volver a pedirlo. La identidad resuelta es la condición técnica que hace posible el principio.

En tripletas, la diferencia es elocuente. El documento de zonificación es una sola entidad, `parte_de` el expediente de Juan, `presentado_por` Juan y `verificado_por` la oficina de zonificación. Cuando la oficina de licencias necesita ese documento, no lo vuelve a pedir: lo encuentra colgando del mismo Juan y del mismo expediente. La identidad compartida en `Q` es el eje sobre el que gira todo.

TRIPLE

```
(doc_cert_zonif) ∈ O
  instancia_de   : certificado_zonificacion
  presentado_por : juan                        # Q · el MISMO juan que todas ven
  verificado_por : oficina_zonificacion        # Q · quién lo dio por bueno
  parte_de       : expediente_2026_337         # O→O · vive en el expediente
  vigente_hasta  : 2027-04-19                   # T · su caducidad

# La oficina de licencias NO vuelve a pedirlo: lo consulta donde ya está.
(paso_revision_04, usa_documento, doc_cert_zonif)   # M(O→O) · lo reutiliza
```

POLÍTICA LIBERAL · D8

Roles como `presentado_por`, `verificado_por`, `custodiado_por` o `usa_documento` son de dominio: la política liberal del lexicon los admite sin tocar el catálogo canónico. El motor no protesta y la semántica queda clara para humanos y para un LLM. Lo que las cuatro oficinas comparten no es código: es un lexicon y un grafo.

Falta un actor invisible pero decisivo: **el tiempo**. La interoperabilidad no solo evita pedir dos veces lo mismo; también obliga a respetar plazos. Cada paso tiene su `plazo_dias`, y la ley administrativa establece que, si el plazo vence sin que la oficina responda, el silencio produce un efecto jurídico por sí mismo: en muchos procedimientos, *silencio positivo* (la solicitud se da por aprobada) y en otros, *silencio negativo* (se da por denegada, habilitando el recurso). Es el **silencio administrativo**.

Aquí brilla una idea que ya vimos al hablar de los estados derivados por reglas: un plazo que vence no necesita que nadie pulse un botón. La fecha límite es un dato en `T`, y una regla de vigencia compara esa fecha con el reloj y, al cruzarla, hace nacer un estado nuevo. El tiempo deja de ser un valor pasivo y se vuelve un agente: dispara hechos.

UN PLAZO QUE VENCE ES UN HECHO QUE NACE

El paso de revisión de Juan tiene `plazo_dias = 10` desde el 22 de abril: vence el 2 de mayo. Si llega esa fecha sin resolución, una regla deriva el estado `vencido_por_silencio` y, según el procedimiento, un efecto: aprobación tácita o habilitación del recurso. Nadie lo teclea; lo dispara el cruce de una fecha. Y como nada se sobrescribe, el grafo recuerda a la vez que el plazo existió, que venció y qué consecuencia tuvo (tres hechos, no una celda que se pisa).

TRIPLE

```
(paso_revision_04, vence_el, 2026-05-02T23:59)        # T · derivado de fecha + plazo

# Si el reloj cruza vence_el sin resolución, una regla hace nacer el estado:
(paso_revision_04, estatus_factual, vencido_por_silencio)  ⟦2026-05-03 … ∞⟧
(paso_revision_04, efecto_silencio, aprobacion_tacita)     # K · el efecto jurídico
# ↑ ningún funcionario lo tecleó: lo disparó el vencimiento del plazo (regla de vigencia)
```

Cerremos el círculo con una tabla que resume los seis pasos del trámite de Juan tal como el grafo los guarda: cada uno con su estado, su responsable (de quién es la pelota), su plazo y su fecha. No hace falta más para reconstruir el procedimiento entero, ni para saber, en cualquier instante, qué está esperando y a quién le toca actuar.

Tabla 22.1 · Los pasos del trámite `tramite_juan_lic_04`, reificados. Estado en K, responsable en Q, plazo en N, fecha en T.

#	PASO (SITUACIÓN EN O)	ESTADO (K)	RESPONSABLE (Q)	PLAZO (N)	FECHA (T)
1	`paso_solicitud_04`	presentada	Juan	—	2026-04-20
2	`paso_revision_04`	en_revision	of. licencias	10 días	2026-04-22
3	`paso_observacion_04`	observado	of. licencias	—	2026-04-29
4	`paso_subsanacion_04`	subsanado	Juan	5 días	2026-05-03
5	`paso_aprobacion_04`	aprobado	of. licencias	—	2026-05-08
6	`paso_emision_04`	emitida · vigente	municipalidad	—	2026-05-11

Es la misma promesa del capítulo, llevada del acto aislado al procedimiento completo: nada se borra, todo deja rastro, y el ciudadano deja de ser el integrador. Juan ya no camina papeles entre islas; las oficinas se reconocen entre sí porque reconocen, una sola vez, al mismo Juan.

El lexicon municipal

Para que la empleada de la ventanilla no tenga que aprender jerga de bases de datos, configuramos el lexicon con las palabras del gobierno local. El lexicon es el compilador del que habló la Parte IV: traduce el vocabulario humano a los roles canónicos y resuelve la polisemia. Un dialecto municipal mapea las muchas maneras de nombrar a una persona (según el trámite, la misma persona es «vecino», «solicitante», «denunciante» o «infractor») a un único rol canónico.

PYTHON

```
lex.register_domain_dialect("municipalidad_lomas", {
    "vecino":       "agente",        # quien denuncia o consulta
    "solicitante":  "agente",        # quien pide una licencia
    "denunciante":  "agente",        # quien presenta una denuncia
    "infractor":    "paciente",      # quien recibe la sanción
```

```
    "ordenanza":    "ordenanza_municipal",
    "papeleta":     "sancion_transito",
    "expediente":   "expediente_administrativo",
})
```

Gracias a esto, un funcionario puede escribirle al sistema *«multa al furgón de reparto por estacionar en zona prohibida, según el artículo 7»* y el grafo lo traduce a sus identificadores internos (con su `causado_por` y su `justificado_por` en su sitio) sin que nadie en la oficina sepa que esas etiquetas existen.

El antes y el después

Antes, en SQL. Un esquema municipal típico reparte la información en tablas como `tramites`, `pasos_tramite`, `ordenanzas`, `multas` y `ciudadanos`. Se ensamblan con `JOIN`, pero hay una pregunta que el esquema casi nunca modela: *¿qué norma autoriza cada paso?* El vínculo normativo, si existe, vive en una columna `observaciones` de texto libre, o en una clave foránea que apunta al encabezado de la ordenanza, sin distinguir qué artículo concreto habilita la acción. Cuando el auditor pregunta bajo qué artículo se declaró fundado un recurso, hay que rastrear código, leer comentarios y cruzar a mano tablas que nunca se diseñaron para responder eso juntas.

SQL

```sql
-- Antes: la multa guarda el «motivo» como texto, y la ordenanza
-- como una FK al encabezado. El artículo exacto se pierde.
CREATE TABLE multas (
  id        INTEGER PRIMARY KEY,
  infractor INT,
  monto     NUMERIC,
  motivo    TEXT,             -- «estacionar en zona prohibida, art.7» (¡texto!)
  ordenanza_id INT            -- apunta al encabezado, no al artículo
);
-- «¿Bajo qué artículo se aplicó?»  →  no hay columna que lo responda.
-- «¿Qué la causó, separado de su fundamento legal?»  →  ambos viven en 'motivo'.
```

Después, en WQuestions. La misma información es un solo grafo de hechos, tal como lo modeló este capítulo. Cada acto (solicitud, inspección, emisión, multa, resolución) es una situación reificada que lleva sus propios cables del «por qué»: `causado_por` a la causa fáctica, `justificado_por` al artículo que lo habilita. La trazabilidad normativa no es código a mano ni un `JOIN` extra: es un recorrido de un solo salto. Cualquier auditor, tribunal o periodista responde *«¿qué norma autorizó este acto?»* con la misma consulta que responde *«¿quién lo ejecutó?»* y *«¿cuándo?»*.

> *En un gobierno local, ningún acto se borra: cada uno deja rastro de su causa, su fundamento legal, su resultado y su eventual revocación. Esa promesa no es una funcionalidad añadida; es lo que queda cuando dejas de sobrescribir filas y empiezas a acumular hechos.*
>
> LA LECCIÓN DE LA MUNICIPALIDAD

El veredicto del dominio público

El dominio municipal pone a trabajar **D7 en su forma más exigente**: cada acto del Estado tiene un fundamento factual y uno normativo, y el modelo los preserva como dos relaciones distintas, consultables por separado. La pregunta *«¿por qué se aplicó esta multa?»* tiene dos respuestas válidas según qué «porque» se pida, y el modelo distingue cuál es cuál. Repasemos lo que quedó probado.

> **QUÉ DEMOSTRÓ EL PROTOTIPO AL ABSORBER UN GOBIERNO LOCAL**
>
> **Lo normativo cupo sin extensiones.** Las ordenanzas y sus artículos viven en `O` como cualquier entidad; la cadena norma → trámite → requisito es un camino de cables del eje `M`, no una tabla aparte.
>
> **El doble «por qué» (D7) funcionó.** Causa fáctica y fundamento legal conviven en cables separados (`causado_por` y `justificado_por`) y se consultan de forma independiente.
>
> **Nada se borra (D6).** Una multa revocada permanece en el grafo con su rango de vigencia; la pregunta «¿estaba vigente tal día?» siempre tiene respuesta. La auditoría lee un registro que nunca se mutiló.
>
> **El motor no creció ni una línea.** Lo único que creció fue el lexicon: un puñado de verbos administrativos (`solicitar` , `inspeccionar` , `emitir` , `denunciar` , `multar` , `resolver`) con sus firmas, y un dialecto municipal que traduce «vecino», «infractor» o «papeleta».

Y la conclusión va más allá de una sola alcaldía. Las jerarquías territoriales en `L`, los expedientes que articulan múltiples diligencias y los recursos que rectifican actos sin borrarlos son patrones que reaparecen en todo el sector público: sanitario, judicial, tributario, ambiental. Lo que probamos aquí no es solo que el modelo absorbe una municipalidad; es que **una familia entera de aplicaciones de gobierno encaja con el mismo motor**, sin extensiones, sin proyectos de integración entre módulos que nunca se diseñaron para hablarse.

El gobierno local nos exigió tomarnos en serio el «por qué». El próximo dominio nos exigirá tomarnos en serio el mundo físico: una operación minera, donde las magnitudes se miden en toneladas y leyes de mineral, los sensores son agentes que reportan sin descanso, y la trazabilidad ya no es un asunto de auditores sino de seguridad y de ley.

23

Una operación minera

Casi todo lo que modelamos hasta aquí lo hizo alguien. En una mina, la mitad de lo que importa no lo hizo nadie: la roca cede, el agua se contamina, un sensor dispara una alarma a las tres de la madrugada. Es el dominio donde las preguntas tienen que sostener lo físico.

Son las 03:14 de un martes en el Yacimiento San Marcos, a 4.250 metros sobre el nivel del mar. No hay nadie en el Frente C del Banco 04: el turno de noche extrae en el tajo contiguo y la zona está vacía hasta el amanecer. Y, sin embargo, algo ocurre. En la Estación 7, junto a la quebrada que baja del tajo, un sensor de calidad de agua toma su lectura de rutina (una cada quince minutos, sin descanso, desde que se instaló) y registra **0,34 miligramos de arsénico por litro**. El umbral que la norma ambiental tolera es 0,10. El número rojo no espera a que llegue un humano: dispara un evento, el evento engancha con una norma, la norma obliga a un reporte, y el reporte queda listo para viajar al organismo regulador antes de que el supervisor termine su turno. Nadie tomó esa decisión. La tomó el grafo, porque los hechos estaban tendidos de antemano para que la tomara.

> **EL GIRO DEL CAPÍTULO**
>
> Spa, taxi, clínica, banco, ERP, universidad y municipalidad eran mundos donde toda acción tenía un autor. La minera rompe esa premisa: introduce eventos que *nadie* ejecuta y agentes que *no* son personas. Es la prueba más dura para la noción de agencia.

Esa escena resume lo que distingue a este dominio de los siete que lo preceden. Hasta ahora, cada evento que modelamos tenía detrás una voluntad: alguien vendía, alguien pagaba, una médica diagnosticaba, un funcionario emitía una ordenanza. Aun cuando el actor no era humano (la app del taxi asignando un conductor, el motor antifraude del banco marcando una operación) había una *intención* programada empujando la acción. Una operación minera de gran escala desmonta esa comodidad. En la mina pasan continuamente cosas que ningún agente «hace»: la roca se desprende de la pared, el suelo cede, una junta geomecánica se debilita con el tiempo, una vena de agua subterránea altera la química del río. Son eventos físicos puros, que ocurren porque las condiciones materiales se cumplen, y son tan consecuentes como cualquier acto deliberado: porque cuando esos eventos lesionan a un trabajador o contaminan un cauce, la cadena causal entera tiene que quedar trazada para la auditoría, para la responsabilidad legal y para que no vuelva a ocurrir.

LO QUE PONE A PRUEBA ESTE CAPÍTULO

Un mismo yacimiento nos obliga a sostener, a la vez, cuatro exigencias que ningún capítulo anterior había pedido juntas:

- **Eventos sin agente.** Lo físico ocurre solo; el modelo debe registrarlo sin inventar un culpable.
- **Sensores como agentes.** Quien toma la medición no es una persona sino un instrumento que mide cada quince minutos, sin parar.
- **Cadenas causales largas.** Un accidente se explica retrocediendo de hecho en hecho hasta una condición que precede a todos.
- **Trazabilidad de lote y de equipo.** Un mineral hay que seguirlo de la roca a la onza; un camión, a lo largo de quince años de estados.

Por qué la mina no se parece a nada anterior

Antes de modelar conviene fijar las cuatro fricciones propias del dominio, porque cada una empuja al modelo en una dirección distinta y todas comparecen a la vez.

ESTRUCTURA ESPACIAL PROFUNDA

Una mina no es un lugar: es una *jerarquía* de lugares. Yacimiento → tajo → nivel → banco → frente. Cinco escalones físicos anidados, cada uno con sus propios metros cúbicos extraíbles y su propia condición geomecánica.

EQUIPOS DE VIDA MUY LARGA

Un camión de acarreo cuesta tres millones de dólares y opera quince años. En ese lapso atraviesa decenas de estados (opera, entra a taller, sale, falla, se repara) que hay que poder reconstruir a cualquier fecha pasada.

PRODUCCIÓN EN CASCADA DE UNIDADES

Toneladas de mineral, gramos de oro por tonelada de ley, onzas troy de oro fino. Tres unidades distintas, encadenadas por multiplicación, y cada cifra debe conservar su semántica y su trazabilidad hacia atrás.

EVENTOS AUTOMÁTICOS Y UMBRALES

Sensores miden aire y agua sin pausa; cuando un valor cruza el límite normativo, el reporte regulatorio se dispara solo. La operación corre 24/7 y los hechos se acumulan estén o no presentes las personas.

Repara en la asimetría. En el spa, los hechos llegaban cuando una clienta se acercaba al mostrador; en el banco, cuando alguien ordenaba una transferencia. Aquí los hechos llegan *aunque no haya nadie*: el sensor mide a las tres de la mañana, la pared se debilita mientras todos duermen, el camión registra horas de motor en un turno desierto. El grafo de una mina respira solo. Esa es la novedad que el modelo tiene que absorber sin doblarse.

El yacimiento sobre los siete ejes

El primer trabajo de quien modela es repartir las entidades del negocio en las coordenadas correctas. La decisión se paga después: si una cosa cae en el eje equivocado, el sistema cojea para siempre. Así se acomoda el Yacimiento San Marcos.

El Yacimiento San Marcos, repartido en sus siete coordenadas

Figura 23.1. El dominio minero sobre los siete ejes. Nota dos detalles que ningún capítulo anterior había forzado a la vez. Primero: en `Q` conviven personas y un *sensor de agua*, porque la pregunta «quién tomó esta medición» admite que la respuesta sea un instrumento. Segundo: en `O` aparecen eventos como el `desprendimiento` que carecen por completo de un actor en `Q` —son hechos físicos puros—. La banda inferior, el eje `M`, no contiene entidades sino los cables que las amarran; aquí el protagonista es `causado_por`.

RECORDATORIO · D5

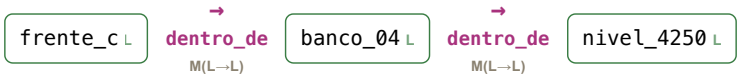
La *agencia contextual* dice que el rol `agente` lo pueden ocupar humanos, organizaciones, software o sensores, según el verbo. Se enunció en el capítulo de situaciones. La minera la lleva a su extremo: hay verbos físicos donde *nadie* ocupa ese rol, y otros donde lo ocupa una máquina que mide.

Las decisiones de diseño que más se ejercitan aquí ya las conoces de capítulos previos; este dominio no estrena ninguna, solo las combina con una intensidad nueva. La agencia contextual (**D5**) se estira hasta admitir eventos sin agente y sensores como agente. La descomposición del porqué (**D7**) reconstruye cadenas causales con `causado_por` sin intención, y justifica reportes con `justificado_por` apuntando a normas. La vigencia bitemporal (**D6**) sigue los quince años de estados de un equipo. Y el patrón de la *situación articuladora* (el mismo que organizó el viaje del taxi y la transferencia del banco) aquí organiza el turno.

Caso 1 · La estructura espacial profunda

Empecemos por el espacio, porque todo lo demás ocurre *en algún sitio*. El nivel más alto es el yacimiento entero. Dentro hay uno o más tajos a cielo abierto. Cada tajo tiene niveles definidos por altitud. Cada nivel se subdivide en bancos (los escalones físicos de la pared del tajo). Y cada banco tiene varios frentes de trabajo activos a la vez. Cinco escalones anidados.

No hace falta inventar nada: es exactamente la subdivisión territorial que ya modelamos en la municipalidad, con un rol de dominio `dentro_de` que la política liberal del catálogo acepta sin protestar. Un cable `L→L` que encadena lugar dentro de lugar.



PYTHON

```
u.assert_fact(tajo_norte,  "dentro_de", yacimiento)
u.assert_fact(nivel_4250,  "dentro_de", tajo_norte)
```

```
u.assert_fact(banco_04,    "dentro_de", nivel_4250)
u.assert_fact(frente_c,    "dentro_de", banco_04)
```

La consulta *«¿dónde ocurrió exactamente este accidente?»* devuelve la cadena entera con un solo recorrido ascendente sobre el grafo: `Frente C → Banco 04 → Nivel 4250 → Tajo Norte → Yacimiento San Marcos`. Toda la geografía jerárquica disponible para cualquier reporte de seguridad, cualquier auditoría regulatoria, cualquier análisis de productividad por zona —sin una sola columna `id_tajo`, `id_nivel`, `id_banco` repetida en cada tabla que mencione un lugar.

Caso 2 · El camión que vive quince años

Un camión de acarreo CAT 793F entra en servicio en abril de 2018. Su vida útil planeada es de quince años. Durante ese tramo atraviesa muchos estados: opera, entra a mantenimiento programado, sale, vuelve a operar, sufre una falla, se repara, sigue operando. El modelo registra todo ese ciclo con vigencia temporal (**D6**, la misma maquinaria bitemporal que en el banco siguió la vida de un préstamo y en la universidad la historia de un salario). Cada estado es una tripleta con su rango de validez.

PYTHON

```
u.assert_fact(camion_007, "estado", operativo,
              valid_from=t_alta_2018, valid_to=t_mant_2024)
u.assert_fact(camion_007, "estado", mantenimiento_prog,
              valid_from=t_mant_2024, valid_to=t_post_mant)
u.assert_fact(camion_007, "estado", operativo,
              valid_from=t_post_mant, valid_to=t_falla_2025)
u.assert_fact(camion_007, "estado", mantenimiento_correctivo,
              valid_from=t_falla_2025, valid_to=t_reparado)
u.assert_fact(camion_007, "estado", operativo,
              valid_from=t_reparado)   # sigue vigente
```

Cinco tripletas. El historial completo del camión a lo largo de ocho años de operación. La consulta *«¿el camión 007 estaba operativo el 15 de marzo de 2024?»* devuelve `mantenimiento_programado`; *«¿y el 1 de junio de 2026?»* devuelve `operativo`. La trazabilidad para auditoría es nativa, no un módulo añadido: el rango `valid_from` / `valid_to` es parte del propio hecho, no una columna que alguien tuvo que recordar mantener.

POR QUÉ ESTO IMPORTA EN UN ACTIVO DE TRES MILLONES DE DÓLARES

Cuando un regulador o una aseguradora pregunta *«¿en qué estado estaba este equipo el día del incidente?»*, la respuesta no puede ser «depende de qué tabla mires». El estado vigente a una fecha es una proyección directa del grafo: se filtran las tripletas `estado` de `camion_007` por la fecha consultada y se devuelve la única cuyo rango la contiene. Sin lógica de aplicación, sin riesgo de que dos sistemas satélite discrepen sobre el pasado.

Caso 3 · La producción en cascada de unidades

La minería de oro registra producción en una cascada de unidades convertibles. Una extracción se mide en **toneladas métricas** de mineral. Ese mineral tiene una *ley* expresada en gramos de oro por tonelada. Multiplicando ambos se obtiene el oro contenido; tras el proceso metalúrgico quedan las **onzas troy** de oro fino vendible. Tres unidades distintas, cada una con su semántica, ninguna intercambiable con la otra.

El turno de noche del 19 de mayo extrae 2.480 toneladas de mineral con una ley de 8,6 g/t en el Frente A. La extracción se reifica como una situación en `0` (lo merece: agrupa varios roles, tiene momento y participa en otras relaciones) y se describe con sus magnitudes tipadas, cada una con su unidad.

```python
extraccion = ingest_situation("extraer", roles={
    "agente":      operador_quispe,
    "extraido":    mineral_oro,
    "monto":       n(2480, "toneladas"),
    "unidad":      tonelada_metrica,
    "lugar_de":    frente_a,
    "ley_mineral": n(8.6, "g/t"),
    "ley_unidad":  gramo_por_tonelada,
})
```

La producción de oro fino no es un campo más de la extracción: es una entidad propia, modelada como **sub-situación** `parte_de` la extracción y calculada de ella. El cálculo (2.480 t × 8,6 g/t ÷ 31,1 g/oz ≈ 685,8 oz) se asienta como hecho, y un cable `calculado_de` apunta hacia atrás, a la extracción de la que salió.

```python
u.assert_fact(produccion_oro, "parte_de",     extraccion)
u.assert_fact(produccion_oro, "monto",        n(685.8, "onzas_troy"))
u.assert_fact(produccion_oro, "unidad",       onza_troy)
u.assert_fact(produccion_oro, "calculado_de", extraccion)
```

Esto es lo que en un dominio físico se llama *trazabilidad de lote*: el grafo nunca pierde de vista de qué roca salió cada onza. Si mañana alguien pregunta *«¿de qué extracción específica vinieron estas 685,8 onzas?»*, la respuesta es directa, un solo salto por `calculado_de`. Y si se descubre que la ley estaba mal medida y hay que recalcular, la producción corregida se modela como un hecho nuevo que `rectifica` al anterior (el mismo patrón de auditoría que vimos en el rediagnóstico de la clínica y la rectificación de multa de la municipalidad). El dato viejo no se borra: queda en el grafo con su historia, porque en minería, igual que en banca, sobrescribir el pasado es perder la auditoría.

Caso 4 · El turno como entidad articuladora

Una operación minera trabaja 24 horas divididas en turnos de doce. Cada turno reúne un puñado de operadores, varios equipos y toda la producción y los incidentes que ocurran en ese lapso. Modelar el turno como entidad articuladora (exactamente como el `viaje_001` del taxi o la `transferencia_001` del banco) hace que todo lo que pasa quede colgado del mismo nodo.

```python
turno = u.add_individual(Individual(id="turno_noche_2026_05_19", axis=Axis.O))
u.assert_fact(turno, "inicio",           at("2026-05-19T18:00"))
u.assert_fact(turno, "fin",              at("2026-05-20T06:00"))
u.assert_fact(turno, "lugar_de",         tajo_norte)
u.assert_fact(turno, "supervisor",       supervisor_mamani)
u.assert_fact(turno, "operador_asignado", operador_quispe)
u.assert_fact(turno, "operador_asignado", operador_calle)
u.assert_fact(turno, "operador_asignado", operador_rojas)
```

Y la extracción, la producción derivada, el accidente, las mediciones ambientales: todos quedan `parte_de` el turno.

```python
```

```
u.assert_fact(extraccion, "parte_de", turno)
u.assert_fact(accidente,  "parte_de", turno)
u.assert_fact(medicion_alta, "parte_de", turno)
```

La consulta *«¿qué pasó en el turno de noche del 19 de mayo?»* devuelve la cadena completa en una sola proyección por `parte_de`: tres operadores asignados, una extracción de 2.480 toneladas, un accidente en el Frente A, una medición ambiental sobre el umbral. Sin tablas, sin *joins*, sin un esquema `turno_operadores` con claves foráneas. Solo el grafo, recorrido hacia abajo desde un nodo.

Caso 5 · La cadena causal sin agente

Aquí llega el caso emblemático del dominio, el que justifica todo el capítulo. A las 23:40 de ese mismo turno, mientras el operador Quispe trabaja en el Frente A del Banco 04, una roca se desprende de la pared y lo golpea. Sufre una contusión en el brazo derecho. La pregunta que la auditoría hará primero es brutal en su simpleza:

> *¿Quién causó el accidente?*
>
> LA PREGUNTA FORENSE

La respuesta (y la que el grafo registra fielmente) es: **nadie**. La roca se desprendió. Nadie la tiró. Pero tampoco se desprendió «de la nada»: hubo una condición previa, un debilitamiento estructural de la pared del Banco 04 que la inspección geomecánica anterior debió haber detectado. Dos hechos físicos encadenados, ninguno con autor, y entre ellos el cable que lo explica todo: `causado_por`.

> **RECORDATORIO · D7**
> No existe un eje «por qué». El porqué se reparte en cuatro cables distintos: `causado_por` (la causa física), `motivado_por` (la razón que impulsa), `con_finalidad` (el objetivo) y `justificado_por` (la norma que autoriza). Aquí trabaja el primero, sin ninguna intención detrás.

El modelo lo registra con dos eslabones causales. Lo decisivo es lo que *no* está: en ninguno de los dos eventos aparece el rol `agente`.

PYTHON

```
# El debilitamiento: una condición geomecánica preexistente
debilitamiento = u.add_individual(Individual(
    id="debilitamiento_pared_b04", axis=Axis.0,
    label="Debilitamiento estructural Banco 04"))
u.assert_fact(debilitamiento, "instancia_de", category("condicion_geomecanica"))
u.assert_fact(debilitamiento, "lugar_de",     banco_04)

# El desprendimiento: evento físico SIN AGENTE, causado por el debilitamiento
desprendimiento = u.add_individual(Individual(id="evento_desprendimiento_07", axis=Axis.0))
u.assert_fact(desprendimiento, "lugar_de",    frente_a)
u.assert_fact(desprendimiento, "momento",     at("2026-05-19T23:40"))
u.assert_fact(desprendimiento, "causado_por", debilitamiento)
# Nota: NO existe  u.assert_fact(desprendimiento, "agente", ...)

# El accidente que sufre Quispe, causado_por el desprendimiento
u.assert_fact(accidente, "paciente",    operador_quispe)
u.assert_fact(accidente, "causado_por", desprendimiento)
u.assert_fact(accidente, "parte_de",    turno)
```

La agencia contextual (**D5**) absorbe esto sin gimnasia: el rol `agente` simplemente *no aparece* en las situaciones físicas, porque la signatura del verbo no lo exige. Y la descomposición del porqué (**D7**) deja que `causado_por` reconstruya la cadena hacia atrás hasta su origen (la condición que el sistema geomecánico no detectó). Veámoslo dibujado.

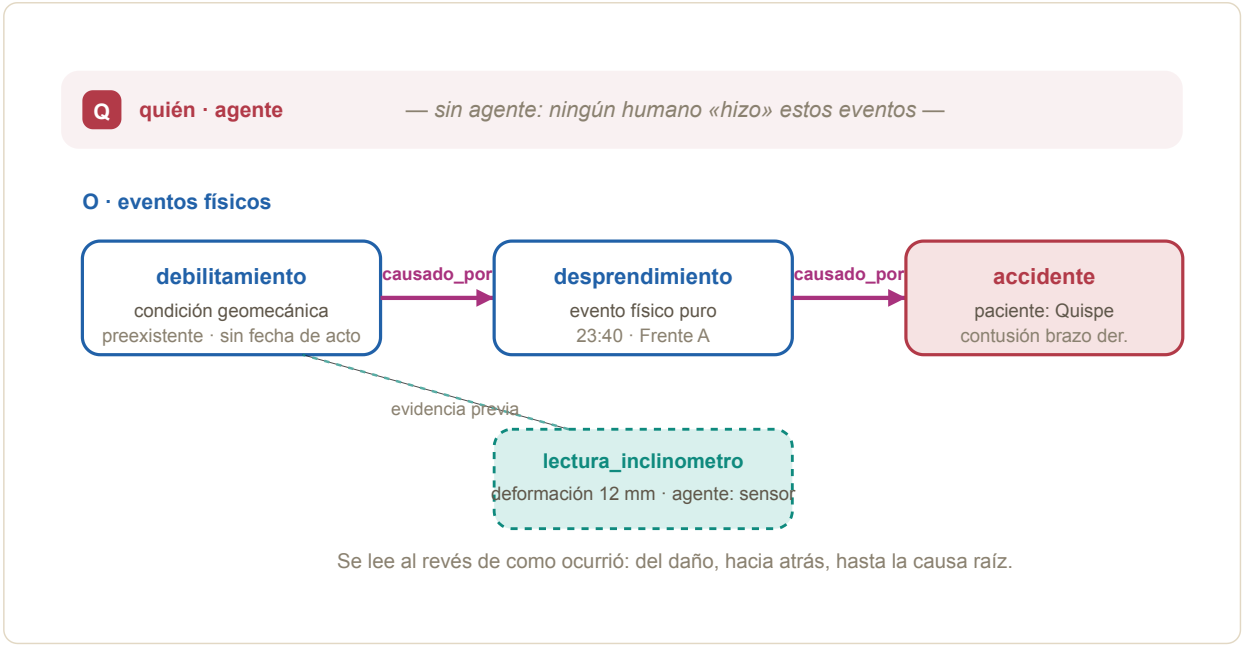


Figura 23.2. Una cadena causal sin culpable. El accidente apunta por `causado_por` al desprendimiento, y este al debilitamiento estructural. Ninguno de los tres tiene agente en `Q` (el carril superior está deliberadamente vacío) y, aun así, el grafo permite reconstruir la responsabilidad completa. La lectura de inclinómetro (en `K`, con un sensor por agente) cuelga del debilitamiento como evidencia previa: la deformación de 12 mm que *sí* se midió y que la inspección debió interpretar.

Cualquier auditor de seguridad recorre esta cadena hacia atrás con un solo barrido sobre el grafo. *«¿Qué causó el accidente?»* → el desprendimiento. *«¿Y qué causó el desprendimiento?»* → el debilitamiento estructural. *«¿Y por qué no se detectó ese debilitamiento?»* → ahí empieza el siguiente capítulo de la investigación, que cruzará la lectura del inclinómetro con el informe geomecánico. La pregunta forense (*¿quién es responsable?*) se vuelve precisa justamente porque el grafo entrega la cadena exacta, no una explicación diluida en un párrafo de texto libre dentro de un campo `observaciones`.

LA TRAMPA QUE EL MODELO EVITA

En un esquema relacional, la tentación es resolver «un evento sin autor» poniendo `NULL` en la columna `id_responsable` (o, peor, inventando un registro ficticio «la naturaleza» para que la clave foránea no se queje). Ambas salidas mienten: la primera confunde «no hubo agente» con «no sabemos quién»; la segunda contamina el catálogo de personas con una entrada que no es una persona. WQuestions no necesita ninguna: el rol `agente` simplemente no se asienta, y la ausencia del cable *es* el dato. No es lo mismo un agente desconocido que un evento que nadie causó, y el modelo sabe distinguirlos.

Caso 6 · El sensor como agente y el reporte por umbral

Volvamos a la alarma con la que abrió el capítulo. Las operaciones mineras grandes están bajo regulación ambiental estricta. Sensores automáticos vigilan la calidad del aire y del agua en estaciones repartidas por el yacimiento, midiendo concentraciones de contaminantes cada cierto intervalo. Aquí el agente de la medición no es una persona: **es el sensor**. Y eso, lejos de ser una excepción incómoda, es precisamente lo que la agencia contextual (**D5**) anticipó.

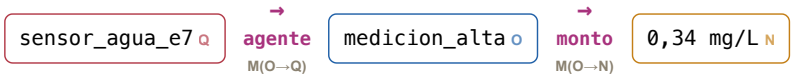
AGENCIA NO HUMANA

Un *sensor* es, para el modelo, un habitante de pleno derecho de `Q`: ocupa el rol `agente` en el verbo «medir» igual que el vendedor lo ocupa en «vender» una camiseta o la cajera en «cobrar». No es una metáfora ni un comodín. La pregunta «¿quién tomó esta lectura?» tiene una respuesta legítima que no es una persona, y el grafo la guarda como tal (con su identificador, su estación, su intervalo de muestreo) para que más tarde se pueda auditar el propio instrumento.

A las 14:00 del 15 de mayo, el sensor de la Estación 7 mide 0,34 mg/L de arsénico en la quebrada que baja del tajo (por encima del umbral normativo de 0,10 mg/L que fija la norma de calidad ambiental). La medición se reifica como cualquier otra situación, con el sensor en el rol de agente.

PYTHON

```
medicion_alta = ingest_situation("medir_calidad", roles={
    "agente":    sensor_agua_e7,        # un sensor en Q (D5)
    "medido_de": arsenico,
    "monto":     n(0.34, "mg/L"),
    "unidad":    miligramo_por_litro,
    "lugar_de":  estacion_07,
    "momento":   at("2026-05-15T14:00"),
})
```



La norma ambiental vive en `0`, con su umbral declarado como un hecho más. No es un valor escondido en el código de validación: es un dato del grafo, citable y versionable.

PYTHON

```
norma_ambiental = u.add_individual(Individual(
    id="eca_agua_cat3", axis=Axis.O,
    label="Estándar de calidad ambiental · agua, categoría 3"))
u.assert_fact(norma_ambiental, "umbral_arsenico", n(0.10, "mg/L"))
```

Cuando el motor de reglas (que el prototipo aún no trae, y que el capítulo 30 declara como trabajo pendiente) detecta que el valor medido supera el umbral, dispara automáticamente el reporte regulatorio. Y ese reporte ejercita las dos caras del porqué que **D7** separa.

PYTHON

```
reporte = ingest_situation("reportar", roles={
    "agente":         minera_san_marcos,   # la empresa reporta
    "tema":           medicion_alta,       # qué se reporta
    "beneficiario":   ente_regulador,      # a quién
    "momento":        at("2026-05-15T18:00"),
    "motivado_por":   medicion_alta,       # la medición concreta que disparó
    "justificado_por": norma_ambiental,    # la norma que lo exige
})
```

El reporte está `motivado_por` la medición específica que cruzó el umbral, y `justificado_por` la norma que obliga a reportar. El grafo registra ambas vinculaciones por separado, sin confundirlas. Si dentro de cinco años un auditor pregunta *«¿por qué se emitió este reporte ambiental?»*, hay dos respuestas precisas y distintas: por la medición de 0,34 mg/L del 15 de mayo, y porque el estándar de calidad ambiental lo exigía. La causa concreta y la justificación normativa son cosas diferentes, y el modelo las mantiene diferenciadas: exactamente el patrón que la municipalidad usó para sus ordenanzas, ahora con normas ambientales en lugar de decretos.

El pulso del yacimiento

Un dominio intensivo en sensores no produce hechos de a uno: produce *series*. El sensor de la Estación 7 no tomó una sola lectura el 15 de mayo; tomó una cada quince minutos durante todo el día, y la mayoría fueron perfectamente normales. La alarma es el

pico que rompe la rutina. Verlo en el tiempo aclara por qué el umbral importa y por qué el disparo tiene que ser automático: a las 14:00, ningún humano estaba mirando ese número.

Figura 23.3. Concentración de arsénico medida por el sensor de la Estación 7 a lo largo del 15 de mayo, contra el umbral normativo (línea verde, constante en 0,10 mg/L). Casi todo el día transcurre por debajo del límite; el pico de las 14:00 (0,34 mg/L) es el que dispara el reporte. Cada punto de la serie roja es una medición reificada con el sensor por agente. Pasa el cursor sobre los puntos para ver las cifras.

Cada uno de esos puntos es, en el grafo, una situación `medir_calidad` con su agente, su monto, su unidad, su momento y su lugar. La serie no es una tabla aparte con un esquema propio de telemetría: es el mismo tipo de hecho atómico que todo lo demás, repetido en el tiempo. Y porque comparte forma con el resto del grafo, la pregunta *«dame todas las mediciones de arsénico de la Estación 7 que superaron el umbral este mes»* es una proyección sobre los mismos cables que ya usamos para todo, no una consulta a un subsistema especializado.

Comisionamiento: poner en marcha como secuencia de pruebas

Hasta aquí miramos el yacimiento *en operación*. Pero antes de que el camión 007 mueva su primera tonelada o de que el sensor de la Estación 7 tome su primera lectura, hubo un proyecto de ingeniería (diseño, procura, construcción, lo que en la industria se abrevia EPC) y, al final de ese proyecto, un momento crítico: el **comisionamiento**. Comisionar es entregar un equipo o una planta a operaciones, pero no de golpe: es una secuencia ordenada de pruebas y protocolos de aceptación que prueban, uno a uno, que lo construido hace lo que el diseño prometía.

> **NOTA · QUÉ ES EPC**
> EPC (*engineering, procurement, construction*) es el modelo de contrato bajo el que se construyen las grandes plantas: una sola empresa responde por la ingeniería, la compra de equipos y el montaje, y entrega la obra «lista para operar». El comisionamiento es la frontera donde esa entrega se vuelve verificable hecho por hecho.

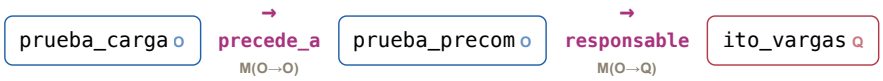
La tentación, en un sistema convencional, es tratar el comisionamiento como un campo de estado del equipo: `comisionado = true`. Pero eso tira a la basura justo lo que la auditoría necesita: *cómo* se llegó a ese sí, qué pruebas se corrieron, quién las firmó y cuándo. Cada etapa del comisionamiento es, en realidad, una **situación reificada** (**D4**): merece nodo propio porque tiene estado, responsable y fecha, y participa en otras relaciones. La planta de procesamiento del Yacimiento San Marcos recorre cuatro fases.

Figura 23.4. El comisionamiento de la planta de procesamiento como secuencia de cuatro fases reificadas. Cada fase es un nodo en `O` con su propio estado (`cerrada`, `en_curso`, `pendiente`), su responsable en `Q` y su fecha en `T`, y se encadena con la siguiente por `precede_a`. El estado no se aplana a un booleano «comisionado»: se conserva el camino completo, que es justo lo que la entrega contractual exige poder mostrar.

Cada fase se asienta con la misma maquinaria de cualquier situación: un identificador fresco, un estado, un responsable que puede ser una persona o el cliente, y una fecha. El cable `precede_a` ordena la secuencia, de modo que ninguna fase abre antes de que cierre la anterior.

PYTHON

```python
# Fase 1 · pre-comisionamiento (cerrada)
prueba_precom = ingest_situation("pre_comisionar", roles={
    "objeto_de":   planta_procesamiento,
    "estado":      cerrada,
    "responsable": supervisor_mamani,
    "momento":     at("2026-03-02T17:00"),
})

# Fase 2 · pruebas en vacío y con carga (en curso)
prueba_carga = ingest_situation("probar_aceptacion", roles={
    "objeto_de":   planta_procesamiento,
    "estado":      en_curso,
    "responsable": ito_vargas,           # inspección técnica de obra
    "momento":     at("2026-03-18T09:00"),
    "precede_a":   prueba_precom,        # va después de la fase 1
})

# Fase 3 · aceptación (protocolo de aceptación, pendiente)
prueba_aceptacion = ingest_situation("aceptar_entrega", roles={
    "objeto_de":   planta_procesamiento,
    "estado":      pendiente,
    "responsable": minera_san_marcos,    # el cliente firma la aceptación
    "precede_a":   prueba_carga,
})
```



La pregunta de cualquier gerente de proyecto (*«¿en qué fase está el comisionamiento de la planta y quién la tiene a su cargo?»*) se contesta con una proyección sobre las situaciones que tienen a `planta_procesamiento` como `objeto_de`, ordenadas por `precede_a`. Y cuando la fase de aceptación cierre, la planta podrá recibir su primer estado `operativo` —exactamente el mismo cable `estado` con vigencia (**D6**) que ya usamos para el camión 007—. El comisionamiento no es un mundo aparte: es la antesala bitemporal de la vida operativa.

Punchlist: las deudas pendientes antes de la entrega

Ninguna obra se entrega perfecta. Entre las pruebas de carga y la firma de aceptación, el inspector levanta un **punchlist**: la lista de pendientes, los defectos y remates que faltan resolver antes de que el cliente reciba la planta. Una baranda sin pintar, una bomba que vibra de más, un manómetro descalibrado. Cada ítem del punchlist bloquea —o no— la aceptación según su gravedad, y eso obliga a clasificarlos.

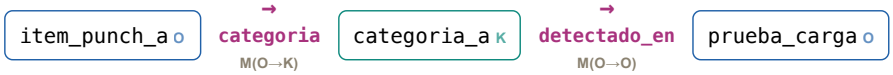
LAS TRES CATEGORÍAS DE UN ÍTEM DE PUNCHLIST

- **Categoría A.** Impide la operación segura. Bloquea la aceptación: hasta que no se cierre, la planta no se entrega.
- **Categoría B.** No impide operar, pero debe resolverse en un plazo corto pactado. No bloquea la aceptación; sí la condiciona.
- **Categoría C.** Remate menor o estético. Se cierra cuando se pueda, sin afectar la entrega.

Cada ítem es una **situación reificada**: un defecto concreto, con su categoría, su responsable de cierre, su estado abierto o cerrado y (cuando se cierra) su fecha de cierre. No es una fila muerta en una hoja de cálculo: es un nodo del mismo grafo, ligado a la fase de comisionamiento que lo destapó y al equipo o lugar donde vive el defecto.

PYTHON

```python
# Ítem A: bloqueante, todavía abierto
item_punch_a = ingest_situation("registrar_pendiente", roles={
    "defecto":      "fuga en sello de bomba P-201",
    "categoria":    categoria_a,           # bloquea la aceptación
    "objeto_de":    planta_procesamiento,
    "lugar_de":     sala_bombas,
    "responsable":  ito_vargas,
    "estado":       abierto,
    "detectado_en": prueba_carga,          # nació en la prueba con carga
    "momento":      at("2026-03-18T11:20"),
})

# Ítem C: estético, ya cerrado
item_punch_c = ingest_situation("registrar_pendiente", roles={
    "defecto":      "baranda nivel 2 sin pintura de seguridad",
    "categoria":    categoria_c,
    "objeto_de":    planta_procesamiento,
    "responsable":  supervisor_mamani,
    "estado":       cerrado,
    "detectado_en": prueba_carga,
    "fecha_cierre": at("2026-03-20T16:00"),
})
```



La gracia es que el punchlist no es una tabla con un esquema propio: es una consulta. «Dame los ítems abiertos de categoría A de la planta» se proyecta sobre las situaciones `registrar_pendiente` con `objeto_de = planta_procesamiento`, `categoria = categoria_a` y `estado = abierto`. Y la regla de negocio (*no se firma la aceptación mientras quede un ítem A abierto*) es una pregunta al grafo, no un disparador escondido en código de aplicación. Veámoslo como tabla de control.

Punchlist de la planta de procesamiento · estado al 20 de marzo

ÍTEM	DEFECTO	CAT.	RESPONSABLE	ESTADO	CIERRE
`item_punch_a`	Fuga en sello de bomba P-201	A	ITO Vargas	**abierto**	—
`item_punch_b`	Vibración en motor de faja FA-3	B	ITO Vargas	**abierto**	—
`item_punch_c`	Baranda nivel 2 sin pintura	C	Mamani	cerrado	20 mar 16:00

Figura 23.5. El punchlist de la planta de procesamiento como proyección del grafo. Cada fila es una situación `registrar_pendiente` con su categoría en `K`, su responsable en `Q`, su estado y (si está cerrado) su fecha en `T`. Mientras la fila `item_punch_a` siga **abierta**, la fase de aceptación de la Figura 23.4 no puede cerrar: la regla «sin ítems A abiertos no hay entrega» es una consulta, no un candado oculto.

POR QUÉ EL PUNCHLIST VIVE MEJOR EN EL GRAFO

En el flujo habitual, el punchlist nace como un Excel que viaja por correo y muere desconectado de todo lo demás: nadie puede cruzar «¿qué pendientes destapó la prueba de carga?» con «¿en qué equipo?» sin copiar y pegar. Aquí cada ítem ya está enlazado a la prueba que lo detectó (`detectado_en`), al equipo afectado (`objeto_de`) y al lugar (`lugar_de`). El historial de cierres queda en el mismo tejido que el comisionamiento y, más tarde, que el mantenimiento.

Mantenimiento: órdenes de trabajo sobre el historial del activo

Comisionada la planta y entregada a operaciones, empieza la vida larga del activo (y con ella el mantenimiento). La industria distingue tres modos, y los tres caben en el modelo como **órdenes de trabajo** ligadas al historial bitemporal (**D6**) del equipo, exactamente el mismo historial que en el Caso 2 siguió los quince años del camión 007.

Preventivo. Programado por calendario o por horas de uso: cambiar el aceite cada 500 horas, sin que nada haya fallado. La orden nace de un plan, no de un síntoma.

Correctivo. Reacciona a una falla ya ocurrida: la bomba se rompió, hay que repararla. La orden nace de un evento —y ese evento puede ser, como vimos, sin agente.

Predictivo. Se adelanta a la falla leyendo señales: el inclinómetro o el sensor de vibración avisa que algo se degrada antes de romperse. La orden nace de una medición que cruza un umbral: el mismo patrón del sensor del Caso 6.

Una orden de trabajo es una situación más, con su tipo, su activo, su disparador y su estado. La predictiva es la más interesante porque enlaza con la telemetría que ya modelamos: una lectura de vibración alta `motiva` la orden, igual que la medición de arsénico motivaba el reporte ambiental.

PYTHON

```
# Orden PREDICTIVA: la dispara una lectura de vibración del propio camión
orden_trabajo_88 = ingest_situation("dar_mantenimiento", roles={
    "tipo":         predictivo,
    "activo":       camion_007,
    "motivado_por": lectura_vibracion_alta,   # señal del sensor (D5)
    "responsable":  taller_mecanico,
    "estado":       programada,
    "momento":      at("2026-06-10T08:00"),
})

# Al ejecutarse, abre un tramo de estado en el HISTORIAL del activo (D6)
u.assert_fact(camion_007, "estado", mantenimiento_predictivo,
    valid_from=t_inicio_ot88, valid_to=t_fin_ot88)
u.assert_fact(orden_trabajo_88, "resultado", "rodamiento sustituido")
```

La orden no flota suelta: cada ejecución abre un tramo en el historial bitemporal del activo, el mismo `estado` con `valid_from` / `valid_to` del Caso 2. Así, la pregunta de la aseguradora (*«¿cuántas horas estuvo el camión 007 en mantenimiento este año, y por qué entró cada vez?»*) se contesta cruzando los tramos `mantenimiento_*` del historial con las órdenes que los abrieron, sin un sistema de gestión de mantenimiento aparte.

Y ese historial cubre toda la existencia del activo, no solo su operación. El equipo no aparece de la nada ni desaparece sin rastro: **se comisiona, opera, se mantiene y, al final, se da de baja**. Cuatro etapas que son cuatro tramos de un mismo eje temporal.

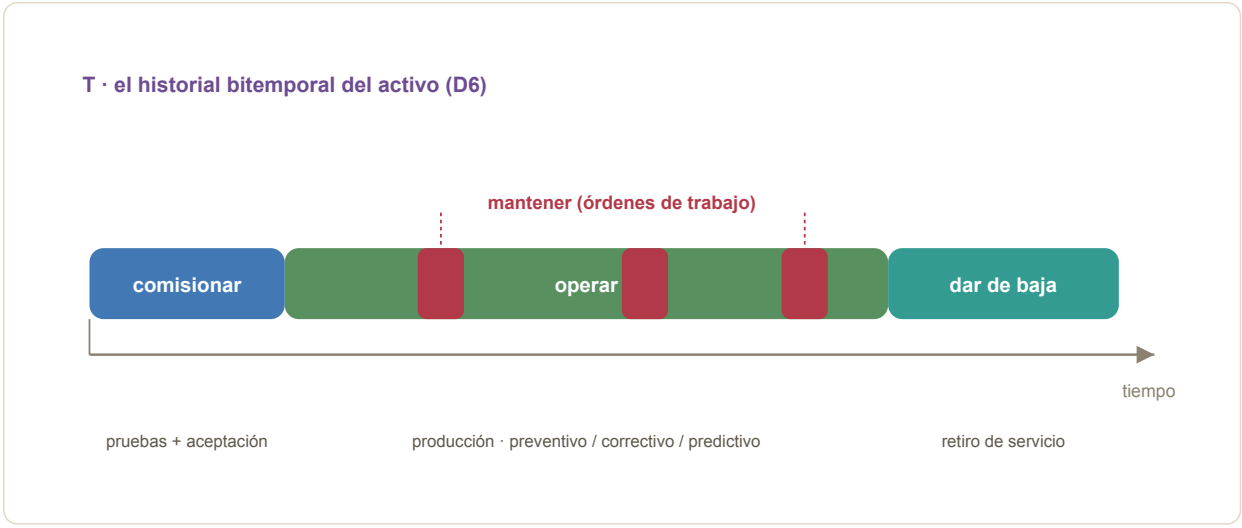


Figura 23.6. El ciclo de vida del activo: `comisionar → operar → mantener → dar de baja`. No son cuatro tablas ni cuatro sistemas, sino cuatro clases de tramo sobre el mismo historial bitemporal (**D6**) del equipo. El mantenimiento (en azul) se intercala dentro de la operación como pulsos (cada uno, una orden de trabajo), y el retiro final cierra el último tramo con su fecha. El estado vigente a cualquier fecha pasada sigue siendo una proyección directa del grafo, como en el Caso 2.

Dar de baja no borra al equipo: cierra su último tramo de estado y abre uno nuevo (`fuera_de_servicio`) con su fecha. El camión 007 seguirá respondiendo, dentro de diez años, a la pregunta *«¿qué estado tenías el día del incidente?»*, aunque para entonces ya no exista físicamente. El ciclo de vida completo queda en el grafo, del primer protocolo de comisionamiento a la última orden de trabajo, sin que ninguna pieza nueva del modelo haya tenido que inventarse para sostenerlo.

Del camión 007 al parque entero

Reconstruimos la vida de un activo: el camión 007, sus quince años de estados, sus órdenes de trabajo. Pero quien planifica el mantenimiento no decide sobre un camión, decide sobre el parque. Y la pregunta que ordena el presupuesto no es «¿en qué estado está este equipo?», sino «¿qué activo acumula más intervenciones correctivas?» o «¿qué causa raíz reaparece trimestre tras trimestre?». Esas no son consultas de otro sistema: cada falla quedó registrada con su activo y su disparador, y el reporte de confiabilidad es contar esas órdenes correctivas, activo por activo.

PYTHON

```
# Órdenes correctivas sobre un activo — un corte; el reporte recorre todo el parque
count(u, Pattern(fixed={"activo": u.ind("camion_007"), "tipo": u.ind("correctivo")},
                 type_constraint=u.ind("dar_mantenimiento")))
```

Una operación entera, vista como hechos

Conviene detenerse y mirar el conjunto. Lo que en una arquitectura tradicional serían cinco o seis subsistemas con sus propios esquemas (flota, producción, seguridad, ambiental, mantenimiento) aquí es un único grafo de hechos donde todo se cruza por los mismos cables. El turno de noche del 19 de mayo, leído de punta a punta, es una sola constelación.

El **turno** articula: reúne a Quispe, Calle y Rojas como operadores asignados, ocurre en el Tajo Norte entre las 18:00 y las 06:00, y lo supervisa Mamani. De él cuelga, por `parte_de`, todo lo que sucede.

La **extracción** aporta la producción: 2.480 toneladas con ley de 8,6 g/t en el Frente A, de las que se calculan 685,8 onzas de oro fino (trazables por `calculado_de` hasta la roca original).

El **accidente** aporta la cadena causal: Quispe como paciente, causado por un desprendimiento, causado a su vez por un debilitamiento estructural —tres eventos físicos sin ningún agente humano.

Y lo **ambiental** aporta el pulso automático: el sensor de la Estación 7 midiendo arsénico sin descanso, el pico que cruza el umbral y el reporte que se dispara solo, justificado por la norma y motivado por la lectura.

Cuatro mundos que un sistema convencional mantendría en bases separadas, hablándose a duras penas por exportaciones nocturnas, conviven aquí en una sola estructura porque todos responden a las mismas siete preguntas. Quién, qué, dónde, cuándo, cuánto, cuál y cómo no cambian de significado al pasar de la flota a la seguridad o de la producción al medio ambiente. Esa invariancia es justamente lo que permite que la pregunta del supervisor (*«¿qué pasó anoche?»*) se conteste de una vez, sin saber de antemano en qué subsistema vivía la respuesta.

El antes y el después: del esquema fragmentado al grafo único

Antes (relacional). Una operación minera tradicional reparte su información entre tablas separadas (`equipos`, `estados_equipo`, `eventos`, `inspecciones_geomecanicas`, `mediciones_sensor`) y la primera grieta aparece enseguida: reconstruir la cadena causal de un accidente que no tuvo autor obliga a cruzar al menos cuatro tablas con *joins* parciales, y aun así el esquema no tiene dónde guardar la relación entre el debilitamiento previo y el desprendimiento posterior sin inventar columnas *ad hoc*. El historial de estados de un camión a lo largo de quince años tampoco cabe limpio: la tabla `estados_equipo` acumula filas sin semántica de vigencia, y cualquier consulta (*¿cuál era el estado exacto del camión el 15 de marzo de 2024?*) exige lógica extra que el esquema no provee. Y la medición del sensor, ¿de quién es? La columna `id_operador` no admite que el autor sea una máquina sin estirarse o mentir.

Después (WQuestions). La misma información vive en un único grafo de hechos. Cada evento físico (el debilitamiento, el desprendimiento, el accidente) es un nodo conectado al siguiente por `causado_por`, y la cadena se recorre hacia atrás de un tirón, sin agente humano en ningún punto. Los quince años de estados del camión son vigencia nativa (**D6**): cada tripleta `(camion_007, estado, operativo)` lleva su propio `valid_from` y `valid_to`, y la consulta por fecha devuelve el estado correcto sin lógica añadida. El sensor es un agente legítimo en `Q` (**D5**): la pregunta «quién midió» se responde con un instrumento, no con un rodeo. El modelo no necesita tablas especiales para la causalidad sin culpable, ni historiales improvisados para equipos de larga vida, ni un esquema de telemetría aparte para los sensores —los absorbe a todos con la maquinaria estándar.

Qué quedó probado

La minería pone a trabajar a **D5 en su forma más radical**: hay eventos físicos puros (el desprendimiento de roca, el debilitamiento de la pared) que ningún agente humano hizo, y son tan consecuentes como cualquier acto deliberado. El modelo los acepta sin esfuerzo porque la signatura del verbo físico no exige agente; basta `causado_por` para reconstruir la cadena entera hasta su origen. Y el mismo D5, por el otro lado, admite que el autor de una medición sea un sensor: agencia no humana de pleno derecho.

L ESTRUCTURA ESPACIAL PROFUNDA

Cinco escalones anidados se modelan con el mismo `dentro_de` de la municipalidad. La política liberal admite el rol de dominio `L→L` sin declararlo en el catálogo.

T EQUIPO DE VIDA LARGA

Quince años de estados quedan trazables con la vigencia bitemporal de D6, la misma que el banco usó para préstamos y la universidad para salarios y notas.

N CASCADA DE UNIDADES

Toneladas, gramos por tonelada y onzas troy conviven, cada cifra con su unidad; la producción derivada se ancla por `calculado_de` a la extracción que la originó. Trazabilidad de lote nativa.

M CAUSALIDAD SIN AGENTE

El cable `causado_por` de D7 reconstruye la responsabilidad forense sin necesidad de un culpable en ningún eslabón. La ausencia del rol `agente` es, ella misma, un dato.

En suma: **el dominio minero no exigió ninguna pieza nueva del modelo**. Lo absorbió combinando piezas que ya conocíamos. El motor sigue siendo el mismo que el del spa más simple. Lo único que crece es el lexicon (cinco verbos nuevos: `extraer`,

`operar_equipo`, `dar_mantenimiento`, `medir_calidad`, `reportar`) y la lista de roles de dominio que la política liberal acepta sin chistar: `dentro_de`, `ley_mineral`, `operador_asignado`, `calculado_de`, `umbral_arsenico`, `medido_de`, entre otros. La arquitectura conceptual no movió una línea entre el comercio más inofensivo y el yacimiento más exigente.

Cierre de la serie de dominios profundos

Con la minera cerramos la **serie completa de ocho dominios industriales** modelados en profundidad: spa, taxi, clínica, banco, ERP, universidad, municipalidad y minera. Cada uno empujó al modelo en una dimensión distinta (el impuesto del comercio, la concurrencia del taxi, la bitemporalidad del banco, la causalidad sin agente de la mina) y cada uno fue absorbido sin extensiones especiales. La arquitectura no cambió ni una línea entre el primero y el último; lo único que cambió fue el lexicon y los roles de dominio. Cuando ocho problemas independientes convergen en la misma respuesta estructural, esa respuesta deja de ser una hipótesis.

Pero el modelo tiene que medirse también con los casos incómodos. Conviene ahora cambiar el ritmo y someter al modelo a cuatro dominios cualitativamente distintos (música, química, fútbol y contratos) buscando explícitamente dónde se resiste: algunos se resuelven con elegancia; otros exigen extensiones al catálogo o dejan pendientes que documentaremos sin disimular. Antes de eso, sin embargo, vale la pena un desvío revelador: tomar un sistema real, viejo y vivo (no un ejemplo diseñado para lucirse) y excavar su esquema para ver qué de todo esto ya estaba ahí, enterrado bajo nombres de columna. Es la arqueología del próximo capítulo.

24

Arqueología de un sistema real

Todos los dominios anteriores partieron de una pizarra en blanco. Pero casi nadie tiene una pizarra en blanco: tiene un sistema viejo, vivo y en producción. Este capítulo excava uno real para mostrar que el modelo se aplica sin reescribir nada.

Son las 19:40 de un viernes. En la recepción de un pequeño complejo que reúne un sauna, un hostal, un gimnasio y un cafetín, una empleada teclea en una pantalla que lleva años igual: fondo gris, botones cuadrados, una rejilla de productos con códigos cortos. El sistema se llama **yaku** y corre sobre MySQL desde hace una década. En la última media hora ha registrado una socia que entró a entrenar sin pagar nada, un huésped que cargó una cerveza a su habitación, un cliente de paso que pagó veinticinco soles por una sauna, y una cortesía: una sauna regalada a una clienta que cumple años. Cuatro hechos del mismo turno, cada uno tecleado en segundos, cada uno guardado en filas que nadie volverá a mirar individualmente. yaku no sabe que está modelando agencia, vigencia ni cobertura. Solo registra ventas. Y, sin embargo (esa es la tesis del capítulo), debajo de sus nombres de columna ya están enterradas casi todas las coordenadas que este libro lleva veintiún capítulos construyendo.

> **EL GIRO DEL CAPÍTULO**
> Los ocho dominios industriales se modelaron *desde cero*: la pizarra limpia, el modelo aplicado sin obstáculos. yaku invierte el ejercicio. No vamos a *modelarlo*: vamos a *mapearlo* sin tocar una fila. La diferencia es categórica.

El capítulo 16 modeló el spa Serena Termas de punta a punta: servicios, comprobantes, impuestos, todo limpio sobre una hoja en blanco. Es un ejercicio valioso para entender qué se *puede* hacer. Pero el libro no puede quedarse ahí, porque la pregunta operativa de la mayoría de sus lectores no es «¿cómo modelaría esto desde cero?», sino otra, mucho más incómoda: **«¿cómo aplico WQuestions sin romper lo que ya funciona?»**. El lector típico no tiene una pizarra: tiene un ERP de diez años, un CRM heredado, una base relacional con tres mil tablas o un sistema artesanal en PHP que mueve la empresa desde antes de que él llegara. yaku es exactamente ese escenario, con la ventaja de que es real, está vivo y podemos mirarle las entrañas.

LO QUE DEMUESTRA ESTE CAPÍTULO

Un sistema heredado se puede traducir al modelo de preguntas *sin reescribirlo*. El trabajo no es una migración: es un diagnóstico. Y arroja tres hallazgos que ningún ejercicio de pizarra limpia revela:

- Los sistemas viejos ya se aproximan a WQuestions **a pedazos, sin nombrarlo**. Mapearlos consiste en detectar qué pedazos están bien, cuáles a medias y cuáles faltan.
- «¿Dónde viven los datos?» tiene **tres respuestas**, no una; elegir bien evita meses de retrabajo.
- El modelo **no inventa información**: lo que el negocio no captura, no aparece en el grafo por arte de magia.

yaku en media página

yaku corre sobre MySQL y administra, con un solo esquema, cuatro líneas de negocio que comparten recepción: el sauna, el hostal, el gimnasio y el cafetín. Sus tablas centrales son las que cualquier negocio de servicios termina teniendo, llámense como se llamen:

`cliente`: quiénes consumen. `persona`: los empleados: recepción, masajistas, instructores. Entre ambas tablas vive todo lo que el modelo llamaría el eje **Q**, repartido en dos según un criterio puramente operativo (quién cobra y quién paga), no semántico.

`producto`: todo lo vendible bajo un mismo techo: habitaciones, sesiones de sauna, jugos, gaseosas, planes mensuales. `venta` más `ventadet`: las comandas y sus líneas. `asistencia`: el registro de entrada y salida del gimnasio, con banderas anexas para sauna y ducha. `socio` más `plan`: las membresías mensuales vigentes.

Y aquí aparece el primer detalle revelador, el que justifica la palabra «arqueología». El campo `producto.MARCA` no guarda lo que su nombre promete (la marca comercial del producto). Guarda su **familia de negocio**: `HOSTAL`, `SAUNA`, `GYM`, `RESTAURANT`, `STOCK`. Es decir, yaku *ya* clasifica sus productos en categorías (ya tiene un eje **K** en funcionamiento) solo que escondido tras un nombre de columna que disimula su función real. Excavar yaku es, literalmente, esto: levantar la capa de nombres engañosos y reconocer debajo las coordenadas que el modelo predice.

> **ARQUEOLOGÍA SEMÁNTICA**
>
> Llamamos *arqueología semántica* al ejercicio de recorrer un esquema heredado preguntándose, tabla por tabla y columna por columna, **a qué eje del modelo responde su contenido**, sin alterar una sola fila. No es ingeniería inversa para reescribir: es un diagnóstico para descubrir cuánto del modelo de preguntas ya estaba implementado de forma tácita, qué partes están incompletas y qué partes faltan por entero.

Qué de WQuestions ya estaba enterrado en yaku

El mapeo arranca con una pregunta inocente que se le hace a cada tabla: *¿en qué eje vive su contenido?* La respuesta, puesta sobre los siete ejes, produce el mapa de la figura 24.1. Conviene leerlo despacio, porque tres de sus observaciones sostienen el capítulo entero.

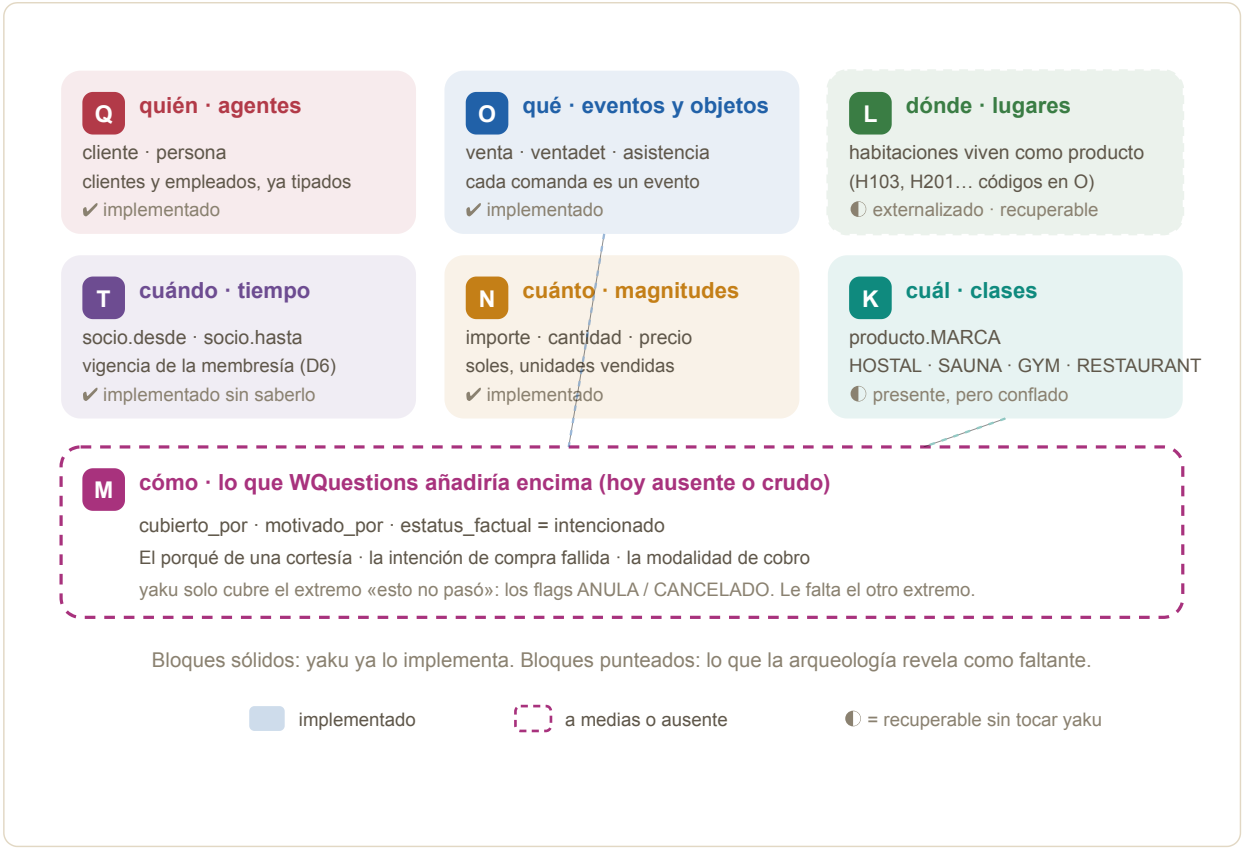


Figura 24.1. Arqueología de yaku sobre los siete ejes. Cada tabla del MySQL existente proyecta su contenido a uno o varios ejes; los bloques sólidos marcan lo que yaku *ya* implementa (aunque nunca lo haya bautizado así), y el bloque punteado inferior marca lo que la excavación revela como faltante o crudo. Repara en tres cosas: en `Q` el modelo ya tiene clientes y empleados tipados; en `T` los campos `socio.desde` / `socio.hasta` son vigencia bitemporal de manual; y en `K` el campo `producto.MARCA` es un eje categórico disfrazado de marca comercial.

Primera observación · yaku ya implementa D6 sin saberlo

Los campos `socio.desde` y `socio.hasta` son, exactamente, el patrón que el capítulo de situaciones prescribe para los hechos que cambian con el tiempo: la membresía no se guarda como un atributo del cliente, sino como un *evento con inicio y fin*. Cuando un socio renueva, yaku no sobrescribe el registro anterior: crea uno nuevo. Esa decisión (tomada hace años por alguien que probablemente solo quería un historial de pagos) es la vigencia bitemporal que el modelo formaliza como **D6**. Y tiene la consecuencia exacta que D6 promete: la pregunta *«¿qué socios estaban activos el 15 de marzo del año pasado?»* se responde con precisión, sin reconstrucciones aproximadas ni «depende de qué tabla mires».

> **RECORDATORIO · D6**
> La *vigencia* dice que las propiedades que cambian se reifican con un rango `inicio` / `fin` (bitemporalidad). Se enunció en el capítulo de situaciones; el banco la usó para préstamos y la minera para quince años de estados de un camión. yaku la tenía implementada de forma tácita en dos columnas de la tabla `socio`.

Esto se traduce al modelo sin mover una fila de yaku. La membresía de una socia es un hecho con su rango de validez, y el sistema heredado ya guarda los dos extremos del rango:

Segunda observación · yaku ya tiene un embrión de `estatus_factual`

Las banderas `venta.ANULA` y `venta.CANCELADO` son la versión cruda de algo que el spa del capítulo 16 modeló con cuidado: el `estatus_factual`, ese campo en `K` que distingue `{real, intencionado, planeado, hipotético, cancelado}`. yaku cubre, sin problemas, el extremo negativo («esto no pasó») con sus dos flags. Lo que le falta es el extremo opuesto: el `intencionado`. El cliente que estuvo a punto de comprar el plan trimestral y se fue sin firmar deja, en yaku, una huella pobrísima: una nota en prosa

libre dentro de `cliente.NOTAS`, no consultable, no agregable, invisible para cualquier reporte. La intención de compra fallida (oro puro para marketing) se evapora.

LA GRIETA DEL CAMPO DE TEXTO LIBRE

Casi todo sistema heredado tiene su `cliente.NOTAS`: un campo de texto donde el negocio vuelca lo que el esquema no supo prever (intenciones, motivos, excepciones, promesas). Es el cementerio de los hechos que el modelo de datos no anticipó. La arqueología semántica consiste, en buena parte, en **rescatar de ese campo lo que debería ser estructura** y darle un eje. El `intencionado` que hoy vive como prosa en `NOTAS` es, en WQuestions, una tripleta consultable como cualquier otra.

Tercera observación · el eje L está externalizado, pero es recuperable

yaku no tiene tabla de lugares físicos. Las habitaciones del hostal viven como filas de `producto` (códigos `H103`, `H201`, etcétera). A primera vista parece una carencia; en realidad es **una decisión de modelado** que el capítulo 16 ya preveía: una misma entidad puede vivir en O *y* en L a la vez. La cámara de vapor del spa era, simultáneamente, una máquina (objeto) y un lugar. yaku eligió priorizar la cara `O` de la habitación (para que entrara sin fricción en el flujo de ventas) y dejó implícita la cara `L`. La buena noticia: WQuestions puede recuperar esa cara *sin cambiar yaku*. Basta declarar, en el lexicon, que cada código de habitación es además un individuo del eje `L`. El dato ya está; solo le faltaba el segundo sombrero.

> *Los sistemas heredados se aproximan a WQuestions a pedazos, sin nombrarlo. La arqueología no migra: diagnostica.*
>
> EL MÉTODO DEL CAPÍTULO

Un concepto nuevo que el ejercicio regala: las modalidades de cobertura

El spa introdujo el `estatus_factual` para responder a una pregunta única: *¿esto pasó o no?*. Es una distinción importante, pero unidimensional. yaku, por ser un negocio real con cuatro líneas conviviendo, hace aflorar otra dimensión **ortogonal** que el libro no había formalizado hasta aquí: *¿cómo se cobró (o se cubrió) este servicio?*. No es lo mismo que «si pasó»; un servicio puede haber ocurrido con total certeza y, aun así, haberse cobrado de cuatro maneras distintas. En la operación diaria de yaku conviven, el mismo viernes, las cuatro:

PAGO DIRECTO

El cliente de paso entra al sauna, paga veinticinco soles, sale. La venta cierra en sí misma: el servicio y su cobro son el mismo hecho.

CUBIERTO POR PLAN

La socia mensual entra a entrenar; su asistencia se registra, pero con `IMPORTE = 0`. El derecho viene del contrato vigente, no del pago del día.

CARGO A ESTANCIA

El huésped pide una cerveza en el cafetín y la cargan a la habitación. El consumo es real, pero el cobro queda agrupado bajo la estancia, para liquidarse al hacer el *check-out*.

CORTESÍA

La recepcionista regala una sauna a la clienta que cumple años. El servicio ocurre, no hay cobro, y —dato clave— *hay un motivo* que merece quedar registrado.

Las cuatro modalidades comparten el mismo hecho factual («se usó el sauna») pero responden de forma distinta a la pregunta del cobro. En yaku, distinguirlas obliga a cruzar tres señales frágiles: el valor de `asistencia.IMPORTE`, la presencia de un `socio` activo y la existencia de una `venta` asociada. Tres consultas que se rompen en cuanto alguien renombra un producto o cambia una regla de caja. WQuestions las colapsa en un único cable explícito del eje M, que llamaremos `cubierto_por`:

TRIPLETAS

```
(uso_sauna_001, instancia_de, servicio_sauna)
(uso_sauna_001, cliente,      juan)
(uso_sauna_001, cubierto_por, pago_directo)               # caso 1: paga y sale

(uso_sauna_002, cubierto_por, contrato_plan_maria_2026)   # caso 2: apunta al contrato
(uso_sauna_003, cubierto_por, estancia_carlos_5234)       # caso 3: apunta a la venta-padre
(uso_sauna_004, cubierto_por, promo_cumpleanos_ana)       # caso 4: apunta a la promoción
```

Cada `uso_sauna` es, en su esencia, un hecho idéntico: alguien usó el sauna. La modalidad de cobertura, por separado, explica *por qué no se cobró como pago directo*. Lo elegante del diseño es que `cubierto_por` apunta a entidades de naturaleza distinta (un contrato, una estancia, una promoción) sin que el cable tenga que saber de antemano cuál; el destino lleva su propia clase. Y es generalizable mucho más allá de un complejo de sauna: cualquier club deportivo, hotel o clínica con cobertura de seguros vive la misma heterogeneidad de orígenes de cobro.

DOS PREGUNTAS ORTOGONALES, DOS CABLES

El `estatus_factual` del spa y el `cubierto_por` de yaku no compiten: son **perpendiculares**. Uno responde *«¿el hecho es real?»*; el otro, *«¿cómo se financió ese hecho real?»*. Una sauna puede ser real y cortesía a la vez. `cubierto_por` pertenece a la familia del porqué que **D7** descompone (apunta al origen que justifica la ausencia de cobro), y se suma al repertorio de cables de ese eje sin tocar la maquinaria del modelo. Es un concepto que la pizarra limpia nunca habría hecho aflorar: lo regaló un sistema real.

RECORDATORIO · D7

No hay un eje «por qué»: el porqué se reparte en `causado_por`, `motivado_por`, `con_finalidad` y `justificado_por`. La cortesía de cumpleaños se ata con `motivado_por` a la razón comercial; el `cubierto_por` general vive en esa misma vecindad: explica el origen del derecho.

¿Dónde viven los datos? Las tres arquitecturas de convivencia

Decir «vamos a aplicar WQuestions» deja sin responder una pregunta crítica que en la pizarra limpia ni siquiera existía: **¿dónde viven los datos?**. Cuando ya hay un MySQL en producción con años de ventas, no hay una sola respuesta —hay tres—, y elegir bien evita mucho retrabajo. La figura 24.2 las ordena de menor a mayor compromiso.

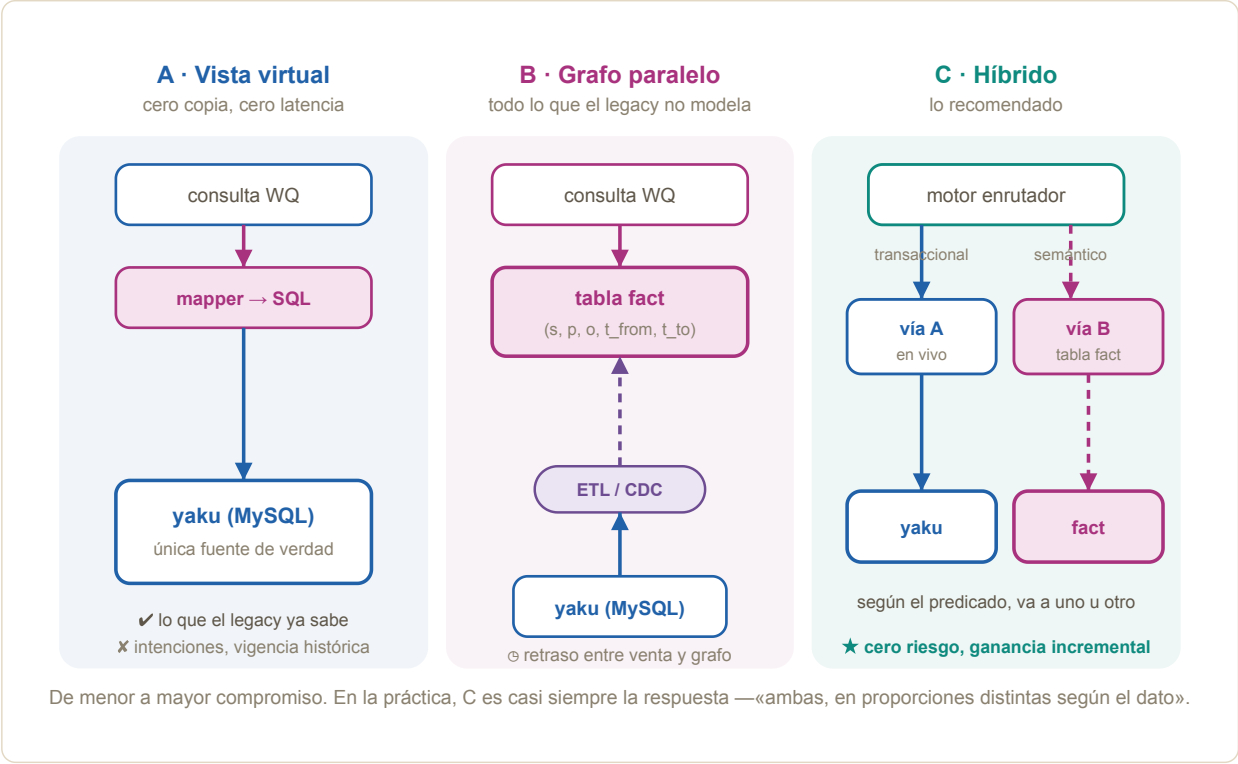


Figura 24.2. Tres arquitecturas para hacer convivir el modelo con un sistema heredado. **A** traduce cada consulta a SQL contra las tablas vivas (cero copia, cero latencia, pero limitada a lo que el *legacy* ya modela). **B** proyecta las filas a una tabla `fact(s, p, o, t_from, t_to)` alimentada por ETL (expresiva, pero con retraso). **C** enruta cada consulta según su predicado: lo transaccional en vivo vía A, lo semántico-puro vía B. La estrella marca la opción recomendada.

Arquitectura A · la vista virtual

El motor WQuestions no almacena datos propios. Cuando recibe una consulta, su *mapper* la traduce a SQL contra las tablas heredadas; yaku sigue siendo la única fuente de verdad. No hay sincronización, no hay copia, no hay latencia. Una pregunta del estilo «¿cuántas saunas tomó Juan este mes?» se convierte, internamente, en algo que el MySQL ya sabe responder:

```sql
-- El mapper traduce el patrón WQ a SQL contra las tablas vivas de yaku.
-- (uso_sauna, cliente, ?)  +  (uso_sauna, durante, este_mes)
SELECT COUNT(*)
FROM   ventadet d
JOIN   venta    v ON v.id = d.id_venta
JOIN   producto p ON p.id = d.id_producto
WHERE  v.id_cliente = 'juan'
  AND  p.MARCA      = 'SAUNA'
  AND  v.fecha BETWEEN '2026-06-01' AND '2026-06-30'
  AND  v.ANULA = 0;
```

Funciona de maravilla para las preguntas que el sistema heredado ya sabe contestar con sus tablas actuales («¿qué huéspedes están alojados ahora?», «¿cuánto facturó el cafetín ayer?»). Y fracasa, por construcción, para todo lo que el *legacy* no modela: la vigencia histórica de la dirección de un cliente, las intenciones de compra fallidas, la justificación de una cortesía. Si el dato no está en una columna, ningún SQL lo inventa.

Arquitectura B · el grafo paralelo

Una tabla separada — `fact(subject, predicate, object, t_from, t_to)`, en MySQL, Postgres, SQLite o donde sea— recibe, vía un proceso ETL o de captura de cambios, una proyección de las filas de yaku convertidas en hechos atómicos. El motor consulta exclusivamente esta tabla `fact`. Su virtud es la libertad: todo lo que el *legacy* no modela (las intenciones, el porqué, la vigencia bitemporal completa, las modalidades de cobertura) cabe limpio, porque la tabla está diseñada justo para alojarlo.

SQL

```sql
-- El grafo paralelo: una sola tabla aloja cualquier hecho, incluso los
-- que yaku no sabe representar.
CREATE TABLE fact (
  subject    VARCHAR(64),
  predicate  VARCHAR(64),
  object     VARCHAR(128),
  t_from     DATETIME,
  t_to       DATETIME NULL
);

-- Una intención de compra fallida: imposible de guardar como tal en yaku.
INSERT INTO fact VALUES
  ('intencion_noa_01', 'instancia_de',    'intencion_compra', NULL, NULL),
  ('intencion_noa_01', 'cliente',         'noa',              NULL, NULL),
  ('intencion_noa_01', 'sobre_producto',  'plan_trimestral',  NULL, NULL),
  ('intencion_noa_01', 'estatus_factual', 'intencionado',     NULL, NULL);
```

El proceso ETL que alimenta esa tabla es, en esencia, la arqueología hecha código: lee una fila relacional de yaku y la *descompone* en las tripletas que el modelo predice. Una fila de `socio` (que para yaku es un renglón con columnas) se convierte en cuatro hechos atómicos, uno por cada coordenada que esa fila escondía:

PYTHON

```python
def proyectar_socio(fila, u):
    """Descompone una fila de la tabla `socio` en hechos atómicos.
    La columna se vuelve cable; el valor, objeto; el eje queda implícito."""
    contrato = u.add_individual(f"contrato_plan_{fila.id}")          # un nodo en O
    u.assert_fact(contrato, "instancia_de", clase_de(fila.nplan))    # K · plan_sauna / gym
    u.assert_fact(contrato, "cliente",      fila.id_socio)           # Q · el cable hacia quién
    u.assert_fact(contrato, "desde",        fila.desde)              # T · vigencia (D6)…
    u.assert_fact(contrato, "hasta",        fila.hasta)              # T · …con inicio y fin
    return contrato
```

Repara en lo que ocurre: el *nombre de la columna* se vuelve el cable, el *valor* se vuelve el objeto, y el eje queda determinado por el lexicon. La fila plana de yaku, leída así, revela las coordenadas que siempre tuvo. Eso es, en una función, todo el capítulo.

Su límite es la latencia. Entre que una venta entra a yaku y aparece en el grafo hay un retraso. Para muchas aplicaciones eso es irrelevante (consultas analíticas, campañas de marketing, tableros que se miran una vez al día); para otras es un bloqueador insalvable (una decisión que se toma en la caja, frente al cliente, ahora mismo).

Arquitectura C · el híbrido pragmático

Los datos transaccionales (clientes, productos, ventas, asistencias) se consultan en vivo contra el *legacy*, vía A. Los datos puramente semánticos que WQuestions agrega (intenciones, modalidades de cobertura, justificaciones, vigencias históricas) viven en la tabla `fact` paralela, vía B. El motor decide a qué *backend* ir según el predicado de la consulta. Es la que recomendaría para casi cualquier adoptante real: aprovecha el sistema operativo existente para lo que ya hace bien y le añade encima, sin tocarlo, la capa semántica que el modelo aporta. Cero riesgo sobre la operación diaria, ganancia incremental en cada consulta nueva.

> **LO QUE EL CAPÍTULO 30 NO DICE SOBRE LA PERSISTENCIA**
>
> El capítulo 30 lista la persistencia industrial como uno de los frentes pendientes, y la plantea como una disyuntiva de tecnología: ¿Postgres o RDF? Este capítulo añade una dimensión que aquella no menciona: persistencia industrial es **también** decidir si la capa WQuestions vive *con* el *legacy* o *aparte* de él. Y la respuesta práctica casi nunca es una sola: es «ambas, en proporciones distintas según el dato». La elección de tecnología viene después de la elección de topología.

El costo real de construir un lexicon

El capítulo 14 presentó el lexicon como una pieza ya construida. El capítulo 30 reconoce que falta poblarlo *a nivel idioma* —miles de verbos del español, los recursos de FrameNet[14], AnCora—. Pero entre el verbo genérico de un idioma y la entrada usable en producción hay un trabajo intermedio que el libro no había puesto sobre la mesa: **construir el lexicon de un negocio concreto**. Para yaku, ese trabajo se desglosa en cinco tareas, y ninguna de ellas es programación.

LAS CINCO TAREAS DEL LEXICON DE UN NEGOCIO

1. **Inventariar el dialecto del personal.** ¿La recepcionista dice «tomar sauna», «usar sauna» o «hacer sauna»? ¿Llama «huésped» al cliente del hostal o «el señor de la habitación»? Esto sale de horas con el equipo, no de leer el SQL. En yaku afloraron los verbos `tomar`, `entrenar`, `hospedar`, `consumir`, `contratar`, más los modificadores `cargar a`, `regalar`, `cancelar`.
2. **Descubrir las polisemias locales.** El verbo `tomar` cubre dos situaciones según el complemento: «tomar sauna» es un *servicio*; «tomar una cerveza» es un *consumo*. yaku no las distingue (escribe ambas en la misma `ventadet`); el lexicon sí, vía el patrón sintáctico del capítulo 14.
3. **Detectar campos que conflan dimensiones.** El campo `producto.MARCA` codifica *dos* cosas ortogonales en una: la categoría semántica (`HOSTAL`, `SAUNA`…) y el control de inventario (`STOCK` = se cuenta; lo demás = no se cuenta). Un jugo de naranja es semánticamente `RESTAURANT` y, a la vez, operativamente no-`STOCK` (no hay jugos en almacén, se preparan al pedido). El lexicon separa las dos dimensiones por construcción.
4. **Clasificar los artefactos en subclases.** yaku tiene casi un centenar de planes en `plan.nplan`, texto libre. Hay que rotular cada uno como `plan_sauna`, `plan_gym` o `plan_mixto`; y otro tanto con los productos `HOSTAL` (matrimonial, doble, simple) y `SAUNA` (adulto, niño, de paso). Una pasada manual o asistida por un modelo de lenguaje resuelve esto *una sola vez*, no por consulta.
5. **Inventariar las modalidades de cobertura** y sus señales en los datos (el `cubierto_por` de la sección anterior). Es el hallazgo ortogonal que el propio ejercicio regala.

La inversión total para yaku es del orden de **dos a cinco días** de trabajo etnográfico, clasificación y escritura del lexicon. Es labor arqueológico-lingüística, no ingeniería. Y se hace una sola vez: desde ese punto, toda consulta futura sobre el negocio *reúsa* el mismo lexicon. Esa amortización es el retorno que justifica el esfuerzo inicial: un costo fijo y pequeño contra un beneficio que se cobra en cada pregunta de los años siguientes.

El gap del gimnasio · el modelo no inventa información

Toca ahora la observación de fondo, una que la euforia de los ejercicios de pizarra limpia tiende a callar: **el modelo no es alquimia**. Si un dato no se captura en ningún lado, no aparece en el grafo por arte de magia.

En yaku, el uso del gimnasio *solo* se registra cuando el cliente es socio en plan (entra al sistema por la tabla `asistencia`). Los días sueltos comprados como producto `MARCA=GYM` se cobran en `venta`, pero no generan registro de entrada y salida: no existe una tabla `asistencia_walkin_gym`. Por tanto, la pregunta «¿cuántas horas usó alguien el gimnasio este mes?» hoy *no se puede* responder para los clientes de paso (ni con yaku puro, ni con WQuestions encima).

LO QUE NO SE REGISTRA, NO EXISTE

El modelo se vende a veces como una solución mágica para preguntas que las bases heredadas no respondían. No lo es. WQuestions *explicita* estructura, conexiones, vigencia y porqué (pero solo de lo que el negocio decidió capturar). Cualquier capacidad nueva exige captura nueva: un lector de huella en la puerta del gimnasio, un QR escaneado, una anotación en recepción. Confundir «el modelo es expresivo» con «el sistema lo sabe todo» es un error costoso. El lector que aplique la arqueología sobre su propio *legacy* descubrirá, igual que con yaku, gaps que el modelo no rellena por sí solo. Eso es parte del contrato.

Una consulta real, de punta a punta, sobre yaku

Cerremos con un ejercicio concreto que ata todo lo anterior. La recepcionista pregunta: *«¿qué huéspedes han usado sauna durante su estancia este mes, y cuánto extra han generado?»*. Esa consulta no tiene una vista preparada en yaku. Hoy obliga a cruzar `asistencia` con `venta` (filtrada por `MARCA=HOSTAL`) y con `ventadet` (filtrada por `MARCA=SAUNA` o por la bandera `asistencia.Sauna`), y aún hay que desambiguar las saunas anexas del gimnasio de las saunas de paso. Es factible, pero frágil: un cambio en los nombres de productos rompe el reporte. Así de enredado se ve en SQL crudo:

SQL

```
-- Hoy, en yaku puro: frágil y atado a los nombres de los productos.
SELECT  h.id_cliente, SUM(d.importe) AS extra
FROM    venta h
JOIN    producto ph ON ph.id = h.id_producto AND ph.MARCA = 'HOSTAL'
JOIN    venta v     ON v.id_cliente = h.id_cliente
JOIN    ventadet d  ON d.id_venta   = v.id
JOIN    producto ps ON ps.id = d.id_producto AND ps.MARCA = 'SAUNA'
WHERE   v.fecha BETWEEN h.fecha AND h.fecha_checkout
  AND   v.fecha >= '2026-06-01'
  AND   v.ANULA = 0
GROUP BY h.id_cliente;
```

En WQuestions (en la arquitectura híbrida C, contra el propio yaku) la misma pregunta se vuelve una composición de roles canónicos, legible y estable:

TRIPLETAS

```
filter(0,
  instancia_de        = uso_sauna,
  durante.instancia_de = estancia_hostal,
  durante.fin         >= inicio_mes,
  cubierto_por.tipo   = pago_directo
)
GROUP BY durante.huesped
PROJECT  SUM(importe)
```

El motor traduce el patrón al SQL apropiado; el modelo de lenguaje, si lo hay, traduce la frase en español a este patrón. Pero el patrón mismo es la pieza estable: tres roles canónicos (`instancia_de`, `durante`, `cubierto_por`) que cualquier negocio de servicios va a reutilizar tal cual. Y aquí está la prueba de fuego. La pregunta de seguimiento (*«¿alguno de esos huéspedes era además socio del gimnasio?»*) se contesta agregando un solo filtro: `huesped IN socios_activos_gym(inicio_mes, fin_mes)`. Sin código nuevo. Sin reporte nuevo. El lexicon ya enseñó qué es un socio del gimnasio; el grafo ya conecta los huéspedes con sus contratos; la consulta compuesta sale de combinar piezas que ya existen.

> *La promesa sobre un sistema en producción: convertir cada consulta nueva en una composición de piezas pre-construidas, en vez de un reporte ad hoc más.*

EL RETORNO DE LA ARQUEOLOGÍA

El antes y el después · del cruce frágil a la composición estable

Antes (yaku puro). Cada pregunta que el negocio no había anticipado se convierte en un reporte nuevo, escrito a mano, que cruza tablas por convenciones de nombres (`MARCA='SAUNA'`, banderas booleanas, fechas comparadas a ojo) y que se rompe en silencio el día en que alguien renombra un producto o cambia una regla de caja. El conocimiento del negocio vive en la cabeza de quien escribió cada consulta, y se pierde cuando esa persona se va. La intención fallida, la cortesía, la vigencia histórica, sencillamente no caben.

Después (WQuestions, híbrido C). Las mismas preguntas se contestan componiendo conceptos. El lexicon (construido una sola vez, en dos a cinco días) enseña el dialecto del negocio; el grafo conecta los hechos por sus cables canónicos; cada consulta nueva es una combinación de roles que ya existen. Lo que yaku no modelaba (el `intencionado`, el `cubierto_por`, la vigencia bitemporal completa) vive en la tabla `fact` paralela sin tocar una sola fila del sistema operativo. Y nada de esto requirió migrar yaku: el MySQL sigue corriendo exactamente igual que el viernes con el que abrimos.

Del viernes en recepción al consolidado del negocio

Mapear yaku no fue el objetivo final: fue la condición para poder preguntar. Y lo que el dueño quiere preguntar no es por la sauna de paso de las 19:40, sino por el negocio entero: cuánto factura cada línea —sauna, hostal, gimnasio, cafetín—, cuál crece y cuál se estanca, qué día de la semana llena el local. Antes esas respuestas vivían repartidas entre `venta`, `ventadet` y los valores de `producto.MARCA`, cruzados a mano cada vez; ahora son cortes de un mismo grafo de ventas, uno por familia de servicio.

PYTHON

```
# Facturación del sauna — un corte; el consolidado recorre las cuatro familias
suma(u, "importe", Pattern(fixed={"familia": u.ind("sauna")},
                           type_constraint=u.ind("venta")))
```

Qué quedó probado

La arqueología de yaku deja tres lecciones que ningún dominio de pizarra limpia podía dejar, y que valen para cualquier sistema heredado, no solo para un complejo de sauna.

D6 **YA ESTABA AHÍ**

Los sistemas viejos implementan trozos del modelo sin nombrarlos: `socio.desde` / `hasta` *es* vigencia bitemporal, `producto.MARCA` *es* un eje categórico. Mapear es reconocer, no reescribir.

C **TOPOLOGÍA ANTES QUE TECNOLOGÍA**

«¿Dónde viven los datos?» tiene tres respuestas. El híbrido (legacy en vivo para lo transaccional, grafo paralelo para lo semántico) da cero riesgo y ganancia incremental.

⌀ **SIN MAGIA**

El modelo no inventa información. El gap del gimnasio walk-in no se cierra con expresividad, sino con captura nueva. Lo que no se registra, no existe.

Y queda un cuarto saldo, distinto de los tres anteriores porque es una *ganancia* y no una lección: el ejercicio regaló un concepto nuevo, las **modalidades de cobertura** (`cubierto_por`), ortogonal al `estatus_factual` y reutilizable en cualquier negocio donde un servicio admita orígenes de cobro heterogéneos. Un sistema real, por el solo hecho de ser real, empujó al modelo a un lugar donde la pizarra limpia nunca lo habría llevado. En suma: **la arquitectura conceptual no movió una línea** entre los ocho dominios diseñados desde cero y este sistema heredado. Lo único que cambió fue el lexicon, la topología de persistencia y un cable nuevo en la familia del porqué.

Tres caminos de salida y la conexión con lo que sigue

El capítulo 16 demostró que el modelo *absorbe* un negocio diseñado desde cero. Este demostró que el modelo *se aplica* sobre un negocio ya digitalizado, sin migrar nada. El lector que llega hasta aquí puede tomar uno de tres caminos, según cuánto quiera arriesgar:

Solo ver si vale la pena. Aplicar la arqueología semántica a sus propias tablas: tabla por tabla, ¿a qué eje responde? Es trabajo de un fin de semana y devuelve un diagnóstico claro de cuánto del modelo ya está implementado y cuánto falta.

Probar el modelo de verdad. Montar la arquitectura híbrida C y un único *POC* de una consulta crítica, de punta a punta. Si esa consulta sobrevive, el resto es repetir el patrón sobre todas las demás.

Ir más lejos. El capítulo 26 toma el paso lógico siguiente: el lexicon que se construyó aquí se vuelve, directamente, el esquema de funciones que un modelo de lenguaje consume —y entonces el personal de yaku puede preguntar en español llano, sin saber que debajo hay un grafo.

El sistema yaku que dio origen a este capítulo sigue funcionando como siempre. Ninguna fila se movió de lugar. Lo que cambió es que ahora, con dos a cinco días de inversión en el lexicon, cualquier consulta del negocio se puede responder componiendo conceptos en vez de programando reportes. Esa diferencia, multiplicada por años de operación, es lo que justifica el esfuerzo —y es la prueba más terrenal de que WQuestions no exige empezar de nuevo: exige, apenas, aprender a leer lo que ya estaba escrito. Con eso cerramos los dominios reales; el próximo capítulo cambia de registro y somete al modelo a cuatro mundos cualitativamente distintos (música, química, fútbol y contratos) buscando, esta vez a propósito, dónde se resiste.

25

Música, química, fútbol, contratos

Un modelo que solo se prueba en lo cómodo no se ha probado. Aquí lo llevamos a cuatro mundos que no se parecen entre sí ni se parecen a un negocio: una partitura, una llama, una cancha y un contrato. Cada uno empuja un eje distinto hasta que cruje. Veremos dónde cede con elegancia y dónde deja una grieta anotada.

En una sala de ensayo, a las once de la mañana, una pianista repite por séptima vez los primeros nueve compases de una sonata. La obra que toca fue escrita hace más de dos siglos y no le pertenece a nadie; el sonido que llena la sala, en cambio, existe solo ahora, sale de sus manos y se apagará en segundos. A trescientos kilómetros, en un banco de pruebas, un inyector descarga una bocanada de metano que se enciende sin que nadie la encienda: no hay autor, solo condiciones que se cumplieron. Esa misma tarde, en un estadio, veintidós personas corren a la vez y, de toda esa simultaneidad, un sistema deberá conservar apenas lo que importa. Y en una notaría, un papel firmado dispone qué está permitido y qué no durante los próximos doce meses, mucho antes de que ocurra ninguno de los hechos que regula.

Cuatro escenas, cuatro mundos. Ninguno es una venta, una historia clínica ni un expediente administrativo. Y, sin embargo, todos pretendemos describirlos con las mismas siete preguntas. Si el modelo de WQuestions es lo que dice ser (un sistema de coordenadas universal) tiene que aguantar aquí. Este capítulo lo somete a los cuatro a propósito, buscando la grieta.

Por qué los dominios incómodos

Hasta este punto, la Parte V recorrió ocho dominios industriales (del spa al banco, de la universidad a la mina) y el modelo aguantó con poca fricción. Eso es alentador, pero también es sospechoso: todos esos dominios, por exigentes que fueran, compartían un aire de familia. Son comerciales, clínicos o administrativos; tienen personas que deciden, transacciones que se registran, fechas que se anotan. Un modelo puede encajar bien en ocho variantes de la misma forma y romperse en la novena que rompe el molde.

> **EL SESGO DEL DOMINIO CÓMODO**
> Que un modelo describa ocho negocios no prueba universalidad: prueba que es bueno con negocios. La prueba dura es el dominio que *no* se parece a un negocio. Por eso este capítulo elige, a propósito, cuatro mundos hostiles al molde comercial.

Por eso elegimos cuatro dominios **cualitativamente** distintos, cada uno escogido porque tensa una parte distinta del modelo hasta el punto de fractura. No buscamos modelar cada uno en profundidad (eso ya lo hicimos con los ocho industriales); buscamos, de cada mundo, su fricción más característica, ponerla bajo la lupa y ver qué pasa. La estrategia es la de un ingeniero de estructuras que no admira el puente terminado, sino que lo carga con camiones hasta oír dónde rechina.

LOS CUATRO EJES DE ESTRÉS

Cada dominio empuja una zona distinta de la arquitectura:

- **Música** · recursión categórica, K denso, sin agente humano del lado de la obra
- **Química** · agencia contextual al extremo (sin agente), plantilla + instancia con escala
- **Fútbol** · concurrencia masiva, doble reloj, estado derivado
- **Contratos** · lo normativo por encima de lo factual, vigencia y argumento explícito

Conviene adelantar la conclusión para leer el capítulo sin suspenso falso: **el esqueleto no se quiebra en ninguno de los cuatro**. Algunos casos se resuelven con limpieza; otros piden un *rol de dominio* que extiende el catálogo; unos pocos dejan un pendiente que dejamos señalado. Ese balance (mucha cobertura, pocas grietas y todas conocidas) es justo lo que distingue una propuesta probada de una promesa.

Figura 25.1. Los cuatro dominios incómodos y la zona del modelo que cada uno tensa. La música estira el eje K hasta volverlo un árbol recursivo; la química vacía el rol del agente (Q tachado) y exige la pareja plantilla–instancia; el fútbol duplica el reloj (T + N) y convierte el marcador en estado derivado; los contratos hacen del eje M (los predicados del porqué) el centro de gravedad.

Música: cuando la categoría se vuelve un árbol

Empecemos por la sala de ensayo. El objeto a modelar es una **composición musical** (tomemos la *Sonata para piano n.º 14, «Claro de luna», op. 27 n.º 2*, de Beethoven) junto con una **interpretación concreta**: la que da la pianista Hélène Renaud el 12 de junio de 2026 en la Sala Devereux. Queremos poder preguntar tanto por la obra abstracta («¿cuántos movimientos tiene?») como por esa noche particular («¿quién la tocó y dónde?»). Son dos preguntas sobre dos cosas distintas que comparten un nombre.

Y ahí está la primera tensión. Una composición es una entidad **atemporal**: fue creada en 1801, sí, pero no *ocurre* en ningún instante; persiste como objeto cultural, idéntica a sí misma se toque o no se toque. Una interpretación, en cambio, **ocurre**: tiene fecha, lugar, intérprete y se desvanece. El reflejo ingenuo es colocar la obra en el eje del tiempo (un patrón temporal abstracto) pero el modelo lo rechaza: el eje T es para momentos y rangos, no para entidades estables. La obra vive en K, lo categórico y

atemporal; la interpretación vive en O, lo situado. Es exactamente la separación que el capítulo de la clase fijó como **D1** (los conceptos atemporales habitan K, las entidades creadas y situadas habitan O) y aquí encuentra su caso límite más nítido.

> **REFERENCIA · D1**
> La frontera entre la plantilla (en K) y la instancia (en O) se enunció en el capítulo de la clase. La música no inventa esa decisión: es el primer dominio donde se ve *por qué* hacía falta. Una obra que nunca «ocurre» pero que se interpreta mil veces solo cabe si K y O son ejes distintos.

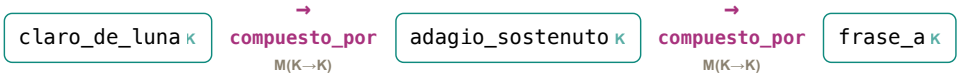
TRIPLETAS

```
(claro_de_luna)        ∈ K
  subtipo_de:          sonata_para_piano
  compositor:          beethoven
  anio_composicion:    1801
  movimientos:         [adagio_sostenuto, allegretto, presto_agitato]   ← otros K

(interpretacion_5512)  ∈ O
  instancia_de:        accion_interpretar
  agente:              helene_renaud          ∈ M(O→Q)
  obra_interpretada:   claro_de_luna          ∈ M(O→K)
  lugar_de:            sala_devereux          ∈ M(O→L)
  momento:             2026-06-12T20:30Z      ∈ M(O→T)
  duracion_min:        16                     ∈ M(O→N)
```

Mira la línea `obra_interpretada`. Su signatura es `M(O→K)`: conecta un evento situado con una categoría atemporal. El rol genérico `tema` del catálogo no servía, porque su signatura es `O→O` y rechaza valores en K. El prototipo obligó a usar un **rol de dominio** (un predicado declarado para este caso) que apunta de O hacia K. No es un parche aislado: es exactamente la forma que reaparecerá con el medicamento que se prescribe (un evento O apuntando al fármaco K) y con el plan que se contrata. La música no es excepcional; es la primera vez que el patrón *«evento que refiere a una categoría»* asoma, y por eso lo nombramos aquí.

La recursión categórica

La segunda tensión es más estructural. Una sonata se compone de movimientos; cada movimiento, de secciones; cada sección, de frases; cada frase, de motivos; cada motivo, de notas. Son cinco niveles de jerarquía anidada, y todos viven en K. Esto rompe una expectativa cómoda: que el eje de las categorías sea una *lista plana* de etiquetas. No lo es. Es un **árbol denso**, con dos relaciones internas que conviene distinguir: `subtipo_de` (una sonata *es un tipo de* obra) y `compuesto_por` (una sonata *se arma con* tres movimientos). La primera es taxonómica; la segunda, de composición.



Con esa estructura, una consulta como *«¿qué obras de Beethoven contienen un movimiento en forma de adagio?»* deja de ser una búsqueda por igualdad y pasa a ser un **recorrido transitivo**: bajar por `compuesto_por` hasta encontrar un nodo cuyo `subtipo_de` sea `movimiento_adagio`. El eje K, que en el spa o el taxi parecía una simple etiqueta de clasificación, aquí muestra su otra cara: la de una ontología navegable con profundidad arbitraria.

> **LO NUEVO QUE APORTÓ LA MÚSICA**
> El eje K no es una lista de etiquetas: es un grafo de categorías con jerarquía interna. Dos relaciones lo estructuran — `subtipo_de` (taxonomía) y `compuesto_por` (composición)— y las consultas interesantes son recorridos transitivos, no comparaciones de igualdad. Cualquier dominio con todo–partes conceptuales (un plan de estudios, una norma con artículos, una receta con subrecetas) hereda esta forma.

LO QUE LA MÚSICA DEJA PENDIENTE

El *tiempo musical* (los compases, los pulsos, el tempo) no es tiempo absoluto y no entra en T. Por ahora se modela como K (`compas_4_4` , `figura_negra`) o se reifica en O cuando hace falta hablar de un compás concreto de una interpretación. Funciona, pero es una solución de compromiso: un sistema de patrones temporales relativos, finos y periódicos, es trabajo documentado y pendiente en la hoja de ruta. No rompe nada operativamente; el tratamiento de primera clase está por construir.

Química: una reacción que nadie hace

Pasemos al banco de pruebas. El objeto es la **combustión del metano**: la reacción que ocurre cuando hay metano, oxígeno y energía de activación suficiente, y que produce dióxido de carbono y agua. Modelarla somete al sistema a presiones que ningún dominio comercial ejerce, y la primera es brutal en su simplicidad.

¿Quién hizo la reacción?

LA PREGUNTA QUE NO TIENE RESPUESTA

La respuesta honesta es: **nadie**. Una reacción química no la *hace* ningún agente en sentido intencional; ocurre porque las condiciones físicas se cumplieron. No hay vendedor, ni conductor, ni motor antifraude detrás. Esto pone a prueba, sin red de seguridad, la decisión que el capítulo de las situaciones llamó **D5**: la *agencia contextual*. Esa decisión establece que el rol `agente` no es obligatorio (solo lo es si el verbo lo exige) y que, cuando lo es, puede ocuparlo un humano, una organización, un programa o un sensor. La química es el caso donde el rol queda sencillamente **vacío**, y el grafo lo acepta sin marcar un error.

REFERENCIA · D5

La agencia contextual se enunció en el capítulo de las situaciones, y en el taxi vimos a un *programa* ocupar el rol de agente. La química cierra el otro extremo del abanico: aquí no hay agente en absoluto. Entre el software que decide y la llama que arde, D5 cubre todo el rango sin un campo obligatorio de más.

La plantilla y la instancia

La segunda presión es más rica. La química distingue con naturalidad dos cosas que el lenguaje cotidiano confunde. Por un lado, *la combustión del metano en general*: una entidad categórica que describe la estequiometría invariante (un mol de CH_4 más dos moles de O_2 producen un mol de CO_2 y dos moles de H_2O). Por otro, *esta combustión concreta* que ocurrió en el cilindro número tres a las 14:32, con cantidades reales y a una escala determinada. El modelo necesita capturar las dos y, sobre todo, **conectar la instancia con su plantilla**.

TRIPLETAS

```
(combustion_metano)    ∈ K          ← la plantilla, atemporal
  reactivos_esteq:     [{ch4: 1}, {o2: 2}]
  productos_esteq:     [{co2: 1}, {h2o: 2}]
  energia_activacion:  kj_por_mol

(reaccion_3187)        ∈ O          ← la instancia, situada
  instancia_de:        combustion_metano    ∈ M(O→K)
  factor_escala:       0.5                  ← la mitad de la estequiometría
  insumo:              n_0_5mol_ch4         (multi)   ∈ M(O→N)
  insumo:              n_1mol_o2            (multi)   ∈ M(O→N)
  lugar_de:            cilindro_3           ∈ M(O→L)
  momento:             2026-05-16T14:32Z    ∈ M(O→T)
  — agente:            ø  (D5: el verbo no lo exige)
```

El puente entre ambas es la línea `instancia_de` más el `factor_escala` . La plantilla guarda las proporciones puras; la instancia guarda cuánto de esas proporciones se consumió realmente. Con eso, todas las cantidades concretas se derivan multiplicando la

estequiometría por el factor, y un evaluador puede verificar el balance de masa sin que nadie lo haya tecleado dos veces.

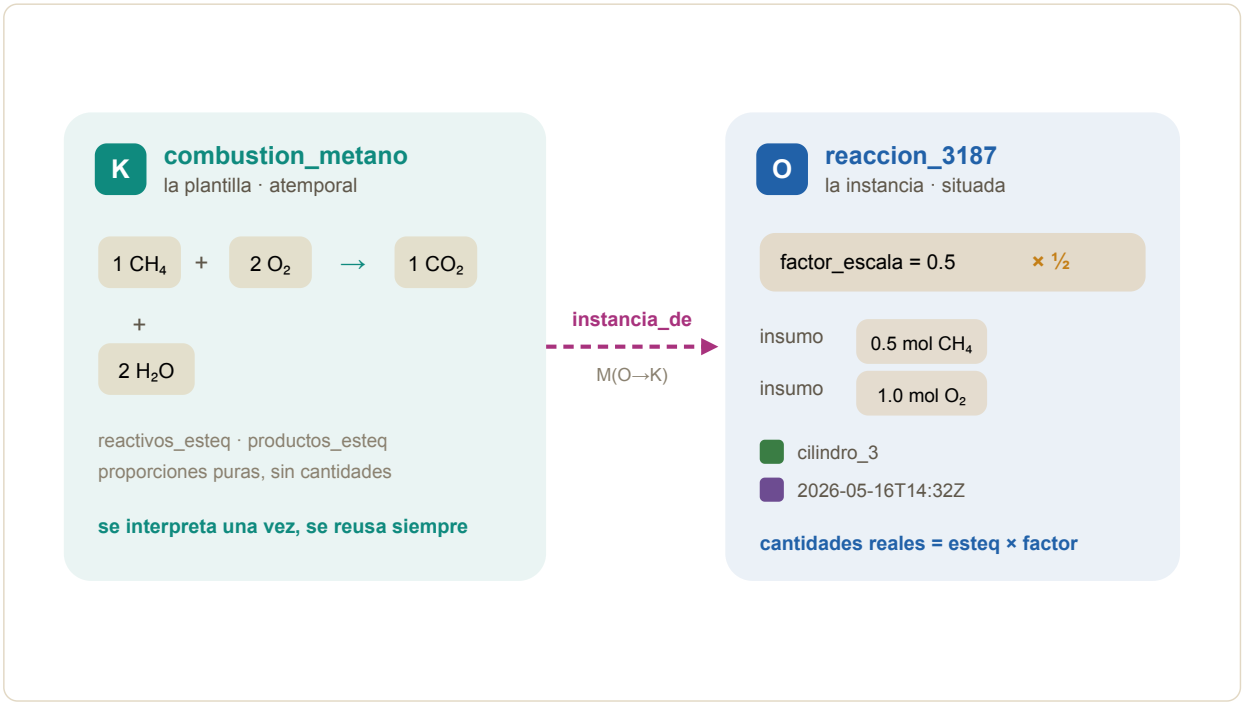


Figura 25.2. El patrón plantilla–instancia. La reacción genérica vive en K con su estequiometría invariante (las proporciones, en tono N); cada reacción concreta vive en O con cantidades reales obtenidas multiplicando la plantilla por un `factor_escala`. La flecha `instancia_de` (un cable del eje M) las une. El mismo molde sirve para sonata→interpretación, receta→preparación y regla→aplicación.

Durante el modelado quedaron registradas dos fricciones que aún persisten, y conviene nombrarlas con precisión:

Uno · el rol `paciente` es demasiado estrecho. En química, «el paciente» de la reacción es el metano: una cantidad N ligada a una categoría K, no un agente Q. El rol canónico `paciente` tiene signatura `0→Q` y rechaza esto. La solución por ahora es un rol de dominio `insumo: 0→N` (múltiple), que cubre el caso con limpieza. Pero deja en evidencia que la signatura canónica podría tener que ensancharse.

Dos · `reactivo` y `producto` no están en el catálogo. El catálogo canónico (lo que el capítulo del lexicon llamó **D8**) no los incluye. La política liberal del modelo permite registrarlos como roles de dominio sin pedir permiso, así que el dominio funciona hoy. Pero queda anotado el parche de canonizarlos algún día, porque el patrón es universal: *toda* transformación tiene entradas y salidas, no solo las químicas.

REFERENCIA · D8

El catálogo canónico de roles vive «bajo el capó»: el usuario nunca lo toca, escribe en su propio vocabulario y el lexicon traduce. Que `reactivo` no esté canonizado no impide modelar química hoy; solo significa que aún viaja como rol de dominio en vez de como término del catálogo común.

LO NUEVO QUE APORTÓ LA QUÍMICA

La pareja **plantilla en K + instancia en O + factor de escala** resultó ser un patrón general, no un truco químico. La sonata es plantilla y la interpretación su instancia; la receta es plantilla y la preparación su instancia; el procedimiento operativo estándar de una fábrica es plantilla y cada corrida su instancia; la guía clínica es plantilla y cada atención su instancia. La química fue, sencillamente, donde el molde se hizo nítido.

Fútbol: dos relojes y un marcador que no se afirma

Vamos al estadio. El objeto es un **partido concreto**: en nuestra reconstrucción, un clásico de eliminatorias en el que la delantera Mara Solís marca al minuto 67 con un remate de zurda, tras una asistencia de Tania Ojeda. Modelar un partido trae complicaciones que el spa y el taxi jamás presentaron, y la primera es la concurrencia.

En cada instante del partido hay veintidós jugadoras haciendo cosas a la vez, más la árbitra, los cuerpos técnicos y el público. Registrarlo todo sería absurdo e inútil. Aquí conviene una aclaración importante: esto *no* es una fricción del modelo, es una decisión de **granularidad**. Lo que se modela es solo lo significativo (los goles, las tarjetas, los cambios, las jugadas clave) y el resto queda implícito. El mérito del modelo es que **no obliga** a registrar lo que no importa: no exige llenar un campo por cada jugadora en cada segundo. Y, en efecto, no lo hace.

El doble reloj

La segunda complicación es más sutil: en un partido conviven **dos tiempos**. El tiempo absoluto (la hora del reloj de pared, `2026-09-14T21:14:07Z`) y el minuto de partido (el «minuto 67», que vive en una escala relativa al pitazo inicial y que se detiene en el entretiempo). No son el mismo eje. El modelo los absorbe con un doble registro: cada evento lleva un `momento: T` absoluto y, opcionalmente, un `minuto_partido: N` relativo. Así, dos consultas distintas conviven sin pisarse: *«¿qué pasó a las 21:14 hora local?»* recorre el eje **T**, y *«¿qué pasó al minuto 67?»* recorre el eje **N**.

TRIPLETAS

```
(gol_4471)            ∈ O
  instancia_de:       evento_gol
  agente:             mara_solis          ∈ M(O→Q)
  asistente:          tania_ojeda         ∈ M(O→Q)
  equipo:             seleccion_sur       ∈ M(O→O)
  momento:            2026-09-14T21:14:07Z ∈ M(O→T)   ← reloj absoluto
  minuto_partido:     67                  ∈ M(O→N)   ← reloj de juego
  parte_de:           clasico_4471        ∈ M(O→O)
```

El marcador es estado derivado

La tercera complicación es la más instructiva del capítulo entero. El «Selección Sur 2 – Selección Norte 1» **no es un hecho que alguien afirma**: es la *conclusión* de agregar todos los goles registrados. Nadie escribe el marcador como dato; el marcador se **deriva**. Es exactamente la misma forma estructural que la fidelidad del spa (siete sesiones acumuladas dan una gratis) y que el costo total de un viaje en el taxi: el modelo almacena los **hechos primitivos** (cada gol como evento reificado) y un evaluador externo recorre el grafo y agrega. El modelo no calcula marcadores; los **prepara** para que cualquier evaluador lo haga.

PYTHON

```python
goles_sur = count(u, Pattern(
    fixed={"equipo": seleccion_sur},
    type_constraint=u.ind("evento_gol"),
))
goles_norte = count(u, Pattern(
    fixed={"equipo": seleccion_norte},
    type_constraint=u.ind("evento_gol"),
))
marcador = f"{goles_sur} – {goles_norte}"   # el estado, derivado al vuelo
```

Durante el modelado apareció una fricción concreta, una sola. El rol `partes` aplicado a un partido (sus dos equipos) traía la signatura `T→Q`, que asume que las partes son agentes. Pero un *equipo* es más cómodo como **O**: un colectivo compuesto, no un agente individual. El parche propuesto es generalizar `partes` a `O→V` (admitiendo cualquier eje de valor como rango). El prototipo confirmó la fricción y la sorteó con un rol de dominio (`parte: O→O`), pero la canonización del parche es trabajo pendiente real, no resuelto.

HECHOS PRIMITIVOS Y ESTADO DERIVADO

Hecho primitivo: algo que ocurrió y se afirma directamente (un gol, un pago, un diagnóstico). Se almacena. **Estado derivado:** una conclusión obtenida agregando hechos primitivos (un marcador, un saldo, una fidelidad acumulada). No se almacena: se calcula por consulta cuando se necesita. WQuestions guarda la historia de los cambios y deriva el estado; no guarda el estado y olvida la historia.

LO NUEVO QUE APORTÓ EL FÚTBOL

La distinción entre hechos primitivos y estado derivado quedó nítida aquí, y rompe una expectativa que casi todos los sistemas relacionales arrastran: la de almacenar el «estado actual» como tal. El precio de derivar es que el evaluador externo *debe* existir. La ganancia es enorme: **el pasado nunca se pierde**. Siempre se puede reconstruir el marcador al minuto 30, porque los goles siguen ahí, fechados.

Contratos: cuando lo normativo manda sobre lo factual

Última escena, la notaría. El objeto es un **contrato de alquiler** a doce meses, con cláusulas, obligaciones recíprocas, una cláusula de rescisión por impago y (llegado el caso) una rescisión efectiva. Es el dominio donde lo *factual* (lo que pasó) queda en segundo plano frente a lo *normativo* (lo que está permitido, exigido o prohibido). El contrato existe y obliga antes de que ocurra ningún hecho que regule.

La primera tensión es la **vigencia** (del contrato entero y de cada cláusula por separado). Algunas cláusulas valen durante todo el plazo; otras solo en ventanas específicas (la de actualización del alquiler aplica únicamente en el aniversario). El modelo absorbe esto con la decisión que el capítulo de las situaciones llamó **D6** (la vigencia: las propiedades que cambian se reifican con un rango de `inicio` / `fin`). Al implementarlo se vuelve evidente que la *bitemporalidad completa* sería lo ideal: no solo registrar *cuándo algo es cierto*, sino también *cuándo lo afirmamos*. Esa segunda mitad queda documentada como pendiente.

REFERENCIA · D6

La vigencia se enunció en el capítulo de las situaciones y reapareció en el banco, en el ERP y en la universidad: el pasado no se sobrescribe, se acumula con rangos. Los contratos la llevan a su forma más exigente, porque cada cláusula puede tener su propia ventana de validez dentro de la del contrato.

Condición, consecuente y argumento

La segunda tensión son los **condicionales**. Una cláusula tiene la forma *«si X, entonces Y»*: si el inquilino impaga dos meses, el arrendador puede rescindir. El modelo la trata reificando la cláusula como una situación con dos cables, `condicion` y `consecuente`:

TRIPLETAS

```
(clausula_14)          ∈ O
  instancia_de:        clausula_contrato
  parte_de:            contrato_alq_2208    ∈ M(O→O)
  condicion:           impago_2_meses
  consecuente:         rescision_autorizada
  vigencia:            [2026-01-01 .. 2026-12-31]   (D6)
```

Cuando una rescisión efectivamente ocurre, no se dispara sola. Un evaluador (humano o algorítmico) verifica que la condición se cumpla y emite la rescisión como un hecho nuevo, que apunta a la cláusula con el cable `justificado_por`:

Ese cable no es decorativo. El capítulo sobre el porqué mostró que no existe un eje «por qué»: el porqué se reparte en cuatro cables distintos (`causado_por`, `motivado_por`, `con_finalidad` y `justificado_por`), y esa decisión es **D7**. En lo normativo, el cable que importa es el último: la *justificación*. La auditoría legal (*«¿bajo qué autoridad se rescindió este contrato?»*) se reduce a seguir `justificado_por` hasta la cláusula, y de la cláusula al contrato. La cadena de razonamiento entera está en el grafo, no en la cabeza de un abogado.

> **REFERENCIA · D7**
> El porqué no es un eje: es cuatro cables del eje M. En lo normativo domina `justificado_por` (la apelación a una norma); en una reacción química dominaría `causado_por` (la apelación a una causa física). El mismo «por qué» del lenguaje, repartido según a qué apela.

La tercera tensión es la **mutabilidad**. ¿Qué ocurre cuando una cláusula se renegocia? La regla del modelo es severa y deliberada: **los hechos son inmutables**. Una renegociación no edita el hecho viejo; es una **situación nueva** que lo `rectifica` o lo `cancela`. El historial se preserva entero y el «estado actual» se reconstruye por consulta (exactamente lo que vimos en la clínica con un rediagnóstico y en el taxi con una cancelación).

> *En lo normativo, el sistema no debe producir solo el resultado; debe producir la cadena de razones que lo justifica.*
>
> LA LECCIÓN DEL CONTRATO

> **LO QUE EL CONTRATO DEJA PENDIENTE**
>
> **Uno · bitemporalidad completa.** Tenemos el tiempo de validez (*cuándo algo es cierto*) pero no el de transacción (*cuándo lo afirmamos*). Para un litigio que pregunte *«¿qué sabía el sistema en mayo de 2024?»* hace falta agregar `tx_inicio` / `tx_fin` a cada hecho. El prototipo ya guarda un `tx_time`, pero todavía no lo expone como parámetro de consulta.
>
> **Dos · reglas de derivación versionadas.** Si la ley cambia y un contrato firmado antes debe leerse bajo la ley vieja, el evaluador necesita saber qué versión de qué regla aplicar. La regla misma se versiona con D6, pero la *mecánica* de evaluación versionada queda como trabajo de la capa superior, el motor de inferencia.

> **LO NUEVO QUE APORTÓ EL CONTRATO**
>
> La importancia de la **estructura argumentativa explícita**. En un dominio normativo, el producto valioso no es el veredicto («rescindido» o «no rescindido»), sino la *cadena de razones* que lo sostiene. Al exigir un `justificado_por` que apunte a una cláusula, el modelo **obliga** a producir esa cadena. No es opcional: un hecho normativo sin justificación queda visiblemente incompleto.

Lo que emergió de los cuatro

Vistos en serie, los cuatro dominios incómodos dejan tres lecciones que trascienden a cada uno y que conviene grabar.

🧩

EL MOLDE PLANTILLA–INSTANCIA ES UNIVERSAL

Apareció en química (reacción genérica frente a concreta), en música (sonata frente a interpretación) y antes en clínica (protocolo frente a atención) y en taxi (tarifa estándar frente a aplicada). No pertenece a ningún dominio: es un patrón de modelado tan central como la reificación, y merece figurar como convención explícita en el lexicon de cualquier proyecto.

⚙️

EL EVALUADOR EXTERNO ES ESTRUCTURAL

Marcador del fútbol, fidelidad del spa, precio dinámico del taxi, evaluación de cláusulas del contrato, conteo de visitas para un diagnóstico: todo el estado derivado es trabajo del evaluador. El modelo guarda los hechos primitivos y prepara el grafo; el razonamiento se construye encima. Es un principio arquitectónico, no una simplificación postergada.

🔧

LAS GRIETAS SON POCAS Y CONOCIDAS

Tras ocho dominios (los cuatro de aquí más los cuatro previos), las fricciones reales son cuatro: roles canónicos del catálogo D8 algo estrechos (`paciente`, `partes`, `tema`); bitemporalidad completa; tiempo musical y patrones temporales finos; y reglas de derivación versionadas. Cuatro pendientes documentados, ningún punto de quiebre.

Esa última lección es la que más importa para un lector escéptico. Un modelo que nunca falla en público suele ser un modelo que no se ha probado en serio; un modelo que declara con precisión sus cuatro grietas (y muestra que ninguna lo derrumba) es un modelo que ha pasado por el banco de pruebas. Las cuatro pendientes son el mapa de hasta dónde llega hoy la arquitectura y qué le falta para llegar más lejos.

Y como cada gol, cada reacción y cada cláusula son situaciones del mismo tipo, la pregunta que de verdad importa deja de ser «¿quién marcó el gol 4471?» para volverse «¿quién encabeza la tabla de goleadores?» o «¿cuántos contratos siguen sin entrar en vigencia?»: el ranking no es una pieza aparte, es ordenar lo que ya contamos.

EL BALANCE DE LA PARTE V

El modelo se sostuvo a lo largo de los doce dominios del libro (ocho industriales y cuatro de estrés) apoyado en un prototipo en Python de unas 2.250 líneas y un catálogo canónico de 38 roles. Donde hubo fricción, hubo parche o un pendiente documentado. En ninguno de los doce el esqueleto se quebró. Esa es, en una frase, la evidencia que la Parte V se propuso reunir.

Con esto cierra la sala de máquinas. Hemos cargado el puente con camiones hasta oírlo rechinar en cuatro lugares distintos, y los cuatro chirridos están anotados en el plano. Queda una pregunta que recorre todo el libro por debajo y que la siguiente parte pone al frente: si las preguntas son un vocabulario común, ¿qué cambia para las máquinas que hoy aprenden a hablar (los grandes modelos de lenguaje) cuando ese vocabulario existe? A eso vamos.

26

WQuestions y los modelos de lenguaje

Si esta arquitectura hubiera aparecido hace cinco años, no habría tenido con quién hablar. En 2026 sí: la era del function calling, los agentes y MCP encontró, sin buscarlo, la pieza que le faltaba. Las preguntas son ese vocabulario común.

Son las 08:11 de un martes. La doctora Paredes abre el asistente de su sistema clínico antes de la primera consulta y escribe una sola línea: «¿qué medicación venía tomando María Gonzales y desde cuándo?». Tres segundos después la pantalla responde, en español llano, que María tomaba enalapril 10 mg cada mañana desde el 14 de mayo de 2026, sin cambios hasta el 10 de enero de 2027, cuando se ajustó la dosis al rediagnosticarla con hipertensión de grado 2. Para la doctora la escena es trivial: una pregunta en castellano, una respuesta en castellano. Pero entre el instante en que pulsó *Enter* y el instante en que apareció el texto ocurrió un viaje de ida y vuelta entre dos lenguajes (el natural y el estructurado) que vale la pena ver desde adentro.

> **HILO DEL LIBRO**
> Esta es la misma sesión que apareció como `sesion_ia_5521` en los capítulos 3, 4 y 7: el modelo `modelo_lumen_2026`, la usuaria `paredes`, las herramientas `busqueda_web` y `consulta_grafo`. Aquí, por fin, se ve trabajar.

Vista por dentro, esa interacción no fue un acto único sino una secuencia de seis pasos. El modelo de lenguaje (da igual cuál: `modelo_lumen_2026` en nuestros ejemplos, pero la mecánica es idéntica para cualquiera) recibió el texto y lo leyó como una *intención de consulta*. Eligió, del catálogo de funciones que el sistema le expone, la adecuada: `consultar_tratamiento(paciente, periodo)`. Tradujo el castellano a argumentos estructurados: «María Gonzales» se volvió el identificador `maria_g`; «desde cuándo» se volvió un periodo que abarca todo el histórico. Invocó la función, que por dentro corrió un patrón sobre el grafo. El grafo respondió con la prescripción vigente y su historia, usando la vigencia bitemporal (D6) para devolver exactamente lo válido en cada momento. Y, al final, el modelo recompuso todo en una frase fluida, con sus fechas y su conexión causal.

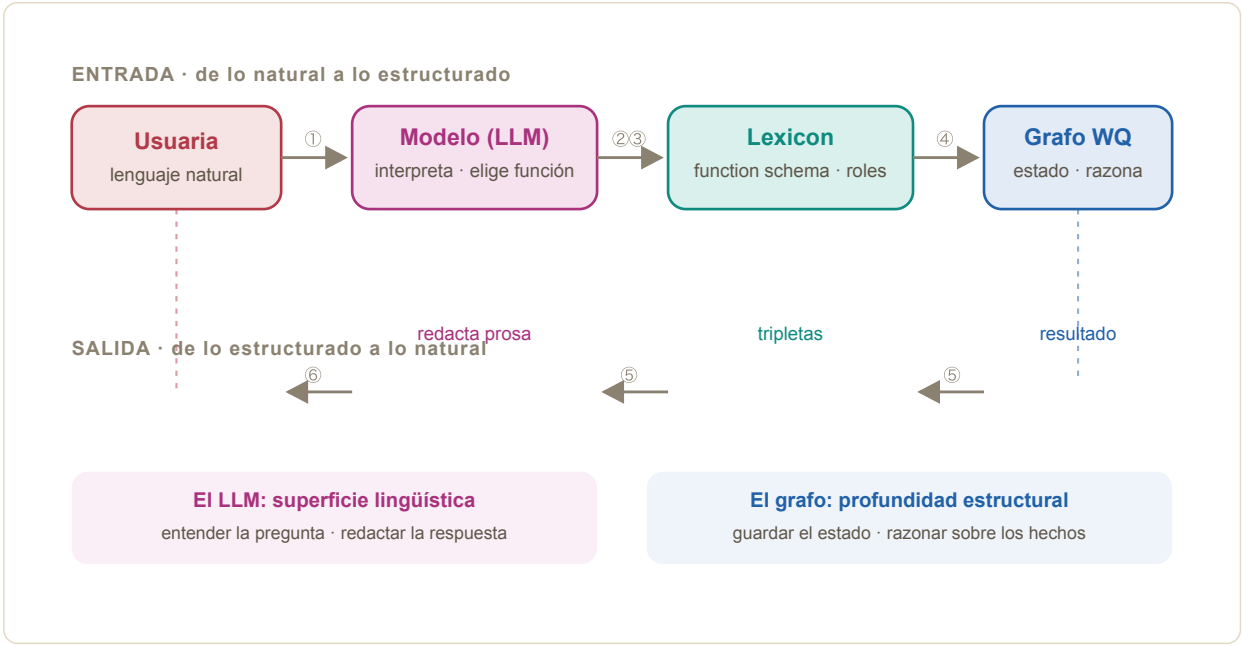


Figura 26.1. El circuito completo. La pregunta entra como lenguaje natural (①), el modelo la interpreta y elige una función del lexicon (②③), el lexicon la convierte en un patrón sobre el grafo (④), el grafo razona y devuelve hechos (⑤), y el modelo los redacta de vuelta en prosa (⑥). El bloque de abajo nombra el reparto: el LLM se ocupa de la M superficie lingüística; el grafo, de la O profundidad estructural.

Esa secuencia (usuario natural, modelo, función estructurada, grafo persistente, respuesta natural) es la forma canónica del momento tecnológico de 2026. Y es exactamente aquello para lo que WQuestions, sin habérselo propuesto, resulta estructuralmente diseñado. Este capítulo defiende una tesis de tiempo: **el momento de esta arquitectura es ahora**, porque solo ahora existe la pieza que traduce con soltura entre el lenguaje de las personas y el conocimiento estructurado. Veremos por qué la pareja es complementaria y no casual, cómo se expone el modelo a un LLM mediante *function calling* y MCP, y qué se vuelve posible que antes era engorroso o sencillamente imposible.

Por qué este es el momento, y no otro

Los modelos de lenguaje de la generación 2024–2026 son fluidos como ninguna tecnología anterior. Pero traen, dentro del mismo paquete, tres debilidades estructurales. No son fallos de implementación que se corrijan con una versión más: son límites de lo que un modelo generativo *es*.

NO RETIENEN ESTADO

Cuando la conversación termina, el modelo olvida. Si la doctora Paredes vuelve mañana, él no recuerda qué pacientes mencionó hoy ni qué se decidió. La continuidad se finge metiendo todo el historial en el *prompt*: caro en *tokens* y difícil de mantener coherente.

CONFUNDEN LO AFIRMADO CON LO VEROSÍMIL

Un LLM puede ser convincentemente incorrecto. Ante una consulta sobre María Gonzales puede *alucinar* un tratamiento plausible que jamás existió. La frontera entre «el sistema lo registró» y «al modelo le suena bien» se borra en la salida.

NO AUDITAN SU RAZONAMIENTO

Si un asistente recomienda algo, el médico necesita rastrear *por qué*: qué hecho lo disparó, qué regla lo justifica, qué evidencia lo respalda. Texto fluido no es lo mismo que cadena trazable.

Ahora mira la otra columna del balance. WQuestions tiene, casi por construcción, las tres propiedades complementarias. Persistencia inequívoca: el grafo guarda y no olvida. Distinción entre lo afirmado y lo conjeturado: el cable `estatus_factual` marca cada hecho como confirmado, previsto o hipotético. Y trazabilidad: las cuatro relaciones del «por qué» —`causado_por`, `motivado_por`, `con_finalidad`, `justificado_por` (D7)— dejan el razonamiento por escrito. A cambio, al grafo le falta justo lo que el LLM hace mejor: la fluidez lingüística, la lectura de matices, la prosa que un humano lee con gusto.

IDEA CLAVE · UNA SIMBIOSIS, NO UNA COMPETENCIA

El LLM se ocupa de la **superficie lingüística** (entender la pregunta, redactar la respuesta) y el grafo de la **profundidad estructural**: almacenar el estado, razonar sobre los hechos, dejar trazas. Cada uno hace aquello en lo que es fuerte; ninguno carga con lo que se le da mal. El error de la última década fue pedirle a una sola tecnología las dos cosas.

De ahí la respuesta a la pregunta del título de la sección. Hace cinco años esta arquitectura no tenía interlocutor: traducir entre datos estructurados y lenguaje natural exigía *parsers* frágiles y reglas a mano. Hoy ese interlocutor existe, habla con fluidez y —el punto decisivo— sabe invocar funciones por su cuenta. La simbiosis pasó de ser una aspiración a ser operativa.

El lexicon es un function schema

Hay una observación que ya hizo el capítulo 14 y que conviene retomar en limpio, porque es la **bisagra** entre las dos mitades del sistema. Lo recordarás así: una entrada del lexicon tiene exactamente la forma que los protocolos de *function calling* esperan. Aquí no la repetiremos con el verbo `vender` de aquel capítulo; tomemos el caso clínico que abre este, `prescribir`, y pongámoslo al lado de su esquema.

Primero, la entrada del lexicon, tal como vive por dentro del modelo: vocabulario natural en los `aliases`, roles canónicos en las claves:

LEXICON · ENTRADA INTERNA

```
verbo: prescribir
  tipo_situacion: accion_prescribir
  obligatorios: [agente, paciente, medicamento_prescrito]
  opcionales:   [frecuencia, duracion, momento]
  aliases:
    agente:               ["doctor", "médico", "doctora"]
    paciente:             ["paciente"]
    medicamento_prescrito: ["fármaco", "medicación", "medicina"]
    frecuencia:           ["cada cuánto", "tomas por día"]
```

Y ahora el mismo objeto, serializado como el *tool schema* que un modelo recibe al arrancar la conversación. No hay traducción manual: es la **misma información** en otra envoltura.

TOOL SCHEMA · FUNCTION CALLING / MCP

```
{
  "name": "accion_prescribir",
  "description": "Registrar una prescripción médica.",
  "input_schema": {
    "type": "object",
    "properties": {
      "agente":               { "type": "string", "eje": "Q", "description": "doctor, médico,
doctora" },
      "paciente":             { "type": "string", "eje": "Q", "description": "paciente" },
      "medicamento_prescrito": { "type": "string", "eje": "K", "description": "fármaco, medicación,
medicina" },
      "frecuencia":           { "type": "string", "eje": "K", "description": "cada cuánto, tomas
por día" },
      "duracion":             { "type": "string", "eje": "K" },
      "momento":              { "type": "string", "eje": "T" }
    },
    "required": ["agente", "paciente", "medicamento_prescrito"]
```

```
  }
}
```

La correspondencia es uno a uno, y vale enumerarla para que no quede como una intuición difusa. El `verbo` se vuelve el `name`. El `tipo_situacion` es el tipo de objeto que la función devolverá al asentarse en el grafo. Los `aliases` de cada rol alimentan la `description`, que es precisamente la pista que el modelo lee para reconocer cómo se expresa ese rol en lenguaje natural. Los ejes de origen (Q, O, K, T) se convierten en metadatos del esquema y aportan una validación semántica extra. Y la lista de `obligatorios` es, literalmente, el campo `required`.

> **D8 · CAPA 4**
> El capítulo 14 describió el lexicon en tres capas: vocabulario del usuario, roles canónicos, identificadores internos. El *tool schema* es la **cuarta capa** de D8: la cara que el lexicon le presenta a una máquina. La fachada mira ahora también hacia los modelos, sin que el catálogo interno cambie un ápice.

Lo que esa identidad implica en la práctica es contundente. Para exponer WQuestions a un LLM **no hay que escribir un adaptador a medida**: hay que *exportar el lexicon* como un catálogo de funciones. Las herramientas que ya existen para *function calling* consumen ese catálogo sin tocar una línea. Lo que en otras arquitecturas es un proyecto de integración, aquí es una función de serialización.

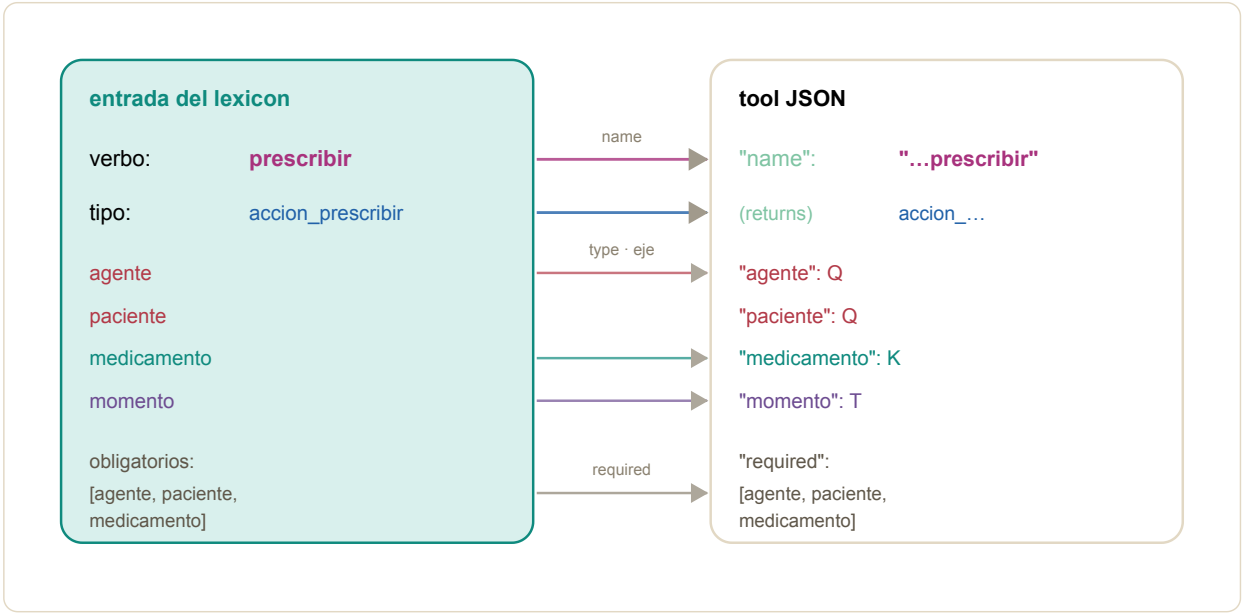


Figura 26.2. La misma información, dos envolturas. Cada campo del lexicon tiene su gemelo en el *tool* que el modelo recibe: el verbo es el `name`, el tipo de situación es lo que la función retorna, cada rol es un parámetro tipado por su eje y los obligatorios son el `required`. La correspondencia es bidireccional: cada verbo del lexicon es una función invocable por un LLM, y cada función expuesta al LLM corresponde a un verbo del lexicon.

MCP: el cable estándar

En 2024, Anthropic publicó la especificación de **MCP** (*Model Context Protocol*), un estándar abierto para que herramientas externas se expongan a los modelos de lenguaje de forma uniforme. La idea es modesta y por eso poderosa: en vez de que cada aplicación invente su propia manera de conectar un LLM a sus datos, MCP define un protocolo común. Un *servidor MCP* declara qué funciones ofrece, qué argumentos acepta y qué tipo de respuesta devuelve; cualquier modelo que «hable MCP» puede usar ese servidor sin una línea de pegamento adicional.

Para WQuestions la consecuencia es casi una tautología: **el servidor MCP de un sistema WQuestions es el lexicon expuesto como funciones**. Cada verbo se vuelve una herramienta de ingesta; cada consulta-WH habitual se vuelve una herramienta de consulta; cada explicación del «por qué» se vuelve una herramienta de auditoría. Mira cómo se ve el servidor del spa (el mismo `Spa Oasis` de la Parte V) una vez publicado:

SERVIDOR MCP · SPA OASIS

```
Tools expuestas:

  ingesta:
    - registrar_sesion(cliente, lugar, inicio, fin)
    - aplicar_promocion(cliente, promocion, momento)
    - redimir_beneficio(cliente, beneficio)

  consulta:
    - sesiones_de_cliente(cliente, desde, hasta)   -> Lista[Sesion]
    - cliente_califica_fidelidad(cliente)          -> bool
    - clientes_con_intencion_de_contratar()        -> Lista[Cliente]

  auditoria:
    - por_que_este_cobro(pago)                      -> Explicacion
    - estado_de_cliente_en(cliente, momento)       -> Estado
```

Un asistente conversacional corporativo (piensa en un «asistente del director financiero», un «asistente del médico de cabecera», un «asistente del periodista de investigación») se construye conectando un LLM a ese servidor. No hay *glue code* enmarañado: hay un catálogo declarativo de funciones y un modelo que las elige según la conversación.

> *El servidor MCP de un sistema WQuestions no es un componente que haya que escribir; es el lexicon, mirado desde afuera.*
>
> LA ECONOMÍA DE LA INTEGRACIÓN

Tres casos de uso, ya posibles en 2026

Conviene aterrizar la idea en escenarios concretos (no especulación, sino cosas técnicamente factibles hoy) donde WQuestions encaja justo en el hueco que el LLM no llena solo.

1 · Asistente operativo de un negocio

El dueño del Spa Oasis, en lugar de revisar reportes, conversa con su sistema. Cada turno se traduce a una o más llamadas a las funciones del lexicon del spa:

CONVERSACIÓN · DUEÑO ↔ ASISTENTE

```
— ¿Cuántos clientes nuevos vinieron este mes?
  → 23 clientes nuevos en mayo, 30 % más que abril. Los miércoles
    concentraron el 40 % del crecimiento.

— ¿Y cuáles de los nuevos están cerca del beneficio de fidelidad?
  → Cinco están a una o dos sesiones del beneficio. Tres reservaron
    para esta semana; los otros dos no aparecen hace más de diez días.

— Mándales un mensaje a los que no vinieron.
  → Borrador listo para Mariana C. y Tomás R. ¿Lo envío?
```

El modelo **no sabe de spas**: sabe del lexicon. El grafo guarda el estado real; la conversación es la *interfaz*, no la base de datos. Lo notable es lo que el dueño no hace: jamás escribe SQL, jamás abre un tablero, jamás aprende un lenguaje de consulta. El lexicon

del spa *es* la API y el LLM es el traductor. El sistema se vuelve **operable conversacionalmente** sin que el negocio renuncie a un gramo de su rigor estructural.

2 · Ingesta automática desde texto libre

Un hospital tiene veinte años de historias clínicas en prosa. Migrarlas a un sistema estructurado tradicional es un proyecto millonario que exige reescribir cada nota a mano. Con WQuestions y un LLM el flujo se invierte. Toma una nota cualquiera:

> *«Paciente femenina de 54 años, refiere cefalea persistente de 3 días de evolución, presión arterial 145/92 mmHg al examen. Se diagnostica HTA grado 1 y se inicia tratamiento con enalapril 10 mg cada mañana. Control en 30 días.»*

UNA NOTA CLÍNICA TÍPICA

Se procesa con un *prompt* que le entrega al modelo el lexicon clínico y le pide: «extrae todas las situaciones de esta nota». El modelo emite (usando *function calling* una vez por situación) los hechos atómicos ya tipados:

INGESTA · LLAMADAS GENERADAS

```
ingest("consultar", roles={
    "agente": dra_paredes, "paciente": maria_g,
    "momento": fecha_nota, "motivo": cefalea,
})
ingest("medir", roles={
    "agente": dra_paredes, "paciente": maria_g,
    "medida_de": presion_arterial,
    "monto": "145/92", "unidad": mmHg,
})
ingest("diagnosticar", roles={
    "agente": dra_paredes, "paciente": maria_g,
    "diagnosticado_como": hta_g1,
})
ingest("prescribir", roles={
    "agente": dra_paredes, "paciente": maria_g,
    "medicamento_prescrito": enalapril,
    "frecuencia": cada_manana,
})
ingest("controlar", roles={
    "paciente": maria_g, "agente": dra_paredes,
    "momento": fecha_nota_mas_30,
    "estatus_factual": previsto,
})
```

Cinco situaciones reificadas (D4), todas validadas contra el catálogo. La prosa original queda como nota de respaldo; la estructura entra al grafo, donde se vuelve consultable, agregable y auditable. Fíjate en la última llamada: el control futuro entra con `estatus_factual: previsto`, no como un hecho consumado: el grafo distingue lo que pasó de lo que está planeado, justo la distinción que un LLM por sí solo no sostiene.

EN LA PRÁCTICA · EL NÚMERO QUE IMPORTA

Una nota de 240 palabras (≈ 320 *tokens* en prosa) genera 24 hechos atómicos cuya serialización pesa unos 80 *tokens*. **Compresión de 4 a 1**, sin pérdida semántica y ganando estructura. Ahora multiplícalo por veinte años de hospital.

3 · Razonamiento entre dominios

Una periodista económica (la misma `paredes` de nuestra sesión, pero ahora en su faceta de investigación) estudia el efecto de una reforma tributaria sobre las ventas minoristas. Tradicionalmente tendría que cruzar bases inconexas: el registro de ventas con un esquema, los datos macro con otro, las noticias políticas en texto libre. Con WQuestions como **sustrato común**, los tres dominios viven en el mismo grafo, cada uno con su dialecto. Ella escribe una sola petición; el modelo la parte en dos consultas, una por dominio, cada cual con su `type_constraint`:

CONSULTA · DOS DOMINIOS, UN GRAFO

```
noticias = query(Pattern(
    fixed={"tema_categorico": impuesto_consumo},
    type_constraint="noticia_politica",
    ask={"agente": Var(), "momento": Var(), "contenido": Var()},
))

ventas = query(Pattern(
    fixed={"sector": retail},
    type_constraint="agregado_ventas_trimestral",
    ask={"momento": Var(), "monto": Var()},
))
```

Las dos listas vuelven al modelo, que las correlaciona por cercanía temporal y compone una respuesta narrativa. Es posible **porque ambos dominios comparten el mismo modelo subyacente**: la periodista no necesita saber que detrás hay dos bases, dos esquemas y dos sistemas. El grafo absorbió los dos, y el LLM razona sobre uno solo.

La economía de tokens, en limpio

Los modelos de 2026 manejan ventanas de contexto enormes: un millón de *tokens*, dos, cuatro. El instinto fácil es volcar todo en prosa y dejar que el modelo encuentre lo que necesita. Funciona, sí, pero gasta el presupuesto a una velocidad ingenua. Tres cifras, tomadas de los dominios que modelamos en la Parte V:

Figura 26.3. El mismo contenido, contado en *tokens*: en prosa (rojo) y serializado como grafo WQuestions (verde). La nota clínica comprime 4×; un mes de operación del spa con tres clientes, 4,4×; un contrato de alquiler con sus cláusulas, 3,6×. Pasa el cursor sobre las barras para ver las cifras exactas.

Pero la compresión, con ser importante, no es lo que más cambia el cálculo. El segundo factor pesa más: **la ambigüedad**. La prosa obliga al modelo a reinterpretar coreferencias («él», «el doctor», «la paciente mencionada»), a inferir relaciones implícitas, a reconstruir la estructura desde cero. Todas esas operaciones consumen atención: *tokens* efectivos que no se invierten en razonar sobre el contenido sino en rearmar el esqueleto.

El grafo WQuestions entrega ese esqueleto **ya reconstruido**. El modelo puede dedicar su capacidad a lo suyo: razonar, conectar, redactar con fluidez. La economía es doble (menos *tokens y* mejor calidad de respuesta) y por eso no se reduce a una partida en una factura de cómputo.

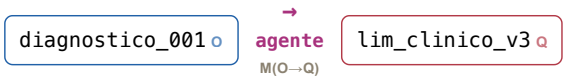
Multi-agente, de forma nativa

Recuerda la agencia contextual (D5): el rol `agente` lo puede ocupar un humano, una organización, una pieza de software o un sensor, según el verbo. Aquello no era adorno; es justo la propiedad que un futuro multi-agente reclama. Imagina, a tres años vista, un sistema empresarial donde varios LLMs especializados (uno clínico, uno financiero, uno legal) colaboran sobre el mismo grafo. Cada uno trae su sub-lexicon; cada uno entra al grafo como **agente** cuando asienta un hecho:

TRIPLETAS · AGENTES DE SOFTWARE

```
(diagnostico_001, agente, lim_clinico_v3)     ∈ M(O, Q)
(asiento_001,     agente, lim_financiero_v2)  ∈ M(O, Q)
(opinion_001,     agente, lim_legal_v1)       ∈ M(O, Q)
```

El grafo registra **quién dijo qué y cuándo**, exactamente como registraría a un médico de carne y hueso. Y conviene verlo como la tripleta que es: un cable del eje M que sale del evento y aterriza en el agente, sea humano o software:



La trazabilidad es uniforme: el sistema audita a los humanos y a los modelos con la misma maquinaria, sin un canal aparte para cada naturaleza de agente. Es D5 cobrando un sentido que, cuando se enunció en el capítulo 9, todavía sonaba abstracto.

WQuestions como infraestructura

Cerremos con la idea más ambiciosa del libro, y la única que justifica todo el aparato conceptual de las cinco primeras partes. **WQuestions, expuesto vía MCP, es una pieza de infraestructura para la IA conversacional empresarial.** No un producto, no un *framework*: una capa estructural sobre la que se construyen aplicaciones.

La distinción no es retórica. Un producto se compra; un *framework* se aprende; una **infraestructura** se da por sentada. Cuando alguien escribe una aplicación web no piensa en TCP/IP; cuando un sistema mueve dinero no piensa en SWIFT; cuando una conversación con un asistente fluye, nadie piensa en el grafo de abajo. La infraestructura es justamente lo que se vuelve invisible porque funciona.

El proyecto, en su forma actual, todavía no está listo para serlo. Le faltan piezas (persistencia industrial, motor de inferencia, bitemporalidad completa, un lexicon poblado con miles de entradas verbales en varios idiomas) y el capítulo 30 las enumera sin disimulo. Pero la *arquitectura* está completa, y el momento histórico empuja a favor.

Tres fuerzas se alinearon a la vez. Los modelos de lenguaje volvieron *operativa* la simbiosis entre prosa y estructura. Protocolos como MCP *estandarizaron el cable*. Y el costo de cómputo bajó hasta el punto en que un asistente conversacional con grafo persistente es asequible para una empresa mediana. Hace cinco años faltaban las tres; hoy están las tres. De ahí la tesis de tiempo con la que abrimos: el momento de WQuestions es ahora.

Lo que el capítulo 27 explora es el lado expansivo de esta misma idea: si WQuestions fuese infraestructura, ¿qué aplicaciones se vuelven posibles que antes no lo eran? Algunas suenan a ciencia ficción; otras resultan obvias en cuanto se las nombra. Las dejaremos sobre la mesa, sin disimular la especulación, porque ahí —y solo ahí— es donde este libro se permite mirar hacia adelante.

27

Aplicaciones futuras

Si la arquitectura funciona, la pregunta deja de ser «¿cómo se modela esto?» y pasa a ser otra, más vértigo: ¿qué se vuelve posible que antes, sencillamente, no lo era?

Es un martes cualquiera de un año que aún no llega. El dueño de una pequeña cadena de spas (cuatro sedes, tres ciudades, ningún equipo de analítica) abre el teléfono mientras espera el café y escribe una sola línea a su asistente: *«¿Qué pasó esta semana?»*. No es una consulta a un panel de control; es una pregunta, en español, como la haría a un gerente de confianza. La respuesta llega en segundos, y no es un número suelto, sino una historia con causa y consecuencia.

> *Las ventas de la sede Centro cayeron un 12 % frente a la semana anterior. La causa principal: un desplome en los planes mensuales el miércoles. Ese mismo día se dispararon las cancelaciones de sesiones (siete en total, seis por el mismo motivo: una ducha averiada). La orden de mantenimiento ya está abierta, pero lleva tres días sin atenderse. Sugiero priorizarla.*

Detrás de esa respuesta tan sosegada hay un trabajo que, para un equipo humano, habría sido media tarde de cruzar planillas: el asistente recorrió cinco fuentes (ventas, planes, sesiones, cancelaciones, órdenes de mantenimiento) siguiendo cadenas de `causado_por` y `motivado_por` a través de un grafo. Lo notable no es que un programa cruce cinco tablas; eso se hace desde hace décadas. Lo notable es que nadie tuvo que programar *esa* consulta. La pregunta no estaba prevista. Simplemente cabía en la estructura.

> **ADVERTENCIA**
> Todo este capítulo es prospectivo. Describe lo que la arquitectura *permite*, no lo que el prototipo ya ejecuta. El cierre, y el capítulo 30, separan lo construido de lo pendiente.

Cambiemos de escena sin cambiar de principio. Un hospital con quinientos médicos y veinte años de historias clínicas registradas como hechos (no como prosa libre atrapada en un campo `observaciones`). Una cardióloga pregunta: *«Muéstrame los pacientes con hipertensión de grado 2 que iniciaron enalapril en los últimos seis meses y no lograron control a los tres.»* Recibe la lista. Refina: *«De esos, ¿cuántos tenían diabetes comórbida, y cuál fue el segundo fármaco que se les añadió?»* Recibe una tabla cruzada. Lo que ayer era un proyecto de tesis (extraer a mano quinientas historias en prosa, codificarlas, analizarlas) hoy es una conversación de tres minutos. Porque las historias estuvieron en el grafo desde el primer día.

Y una tercera, más sombría. Una empresa enfrenta una investigación regulatoria. El fiscal no pregunta qué es cierto hoy, sino algo mucho más difícil: *«¿Qué sabía el directorio sobre la cláusula de garantía el 15 de noviembre de 2024?»* El sistema reconstruye el estado del grafo en esa fecha exacta (no solo qué era verdad entonces, sino qué estaba *afirmado en el sistema* en ese instante) y devuelve los hechos accesibles, los actos formales registrados, las reglas vigentes. La pregunta «¿qué sabía la organización, y cuándo?» deja de ser testimonial y se vuelve consultiva.

> *Las tres escenas no son tres inventos. Son el mismo cambio arquitectónico mirado desde tres ángulos.*

LA TESIS DEL CAPÍTULO

Aquí está el punto que cuesta ver y conviene subrayar. Analítica conversacional, búsqueda longitudinal de pacientes y auditoría retrospectiva parecen tres productos distintos, que tres empresas distintas venderían por separado. Pero las tres descansan en una sola base: un **grafo persistente** como sustrato, un **modelo de lenguaje** como interfaz, y un conjunto de **relaciones canónicas** como infraestructura de razonamiento. Cuando esa base existe, las tres aplicaciones (y muchas más) dejan de ser proyectos a medida y pasan a ser consecuencias. Lo que sigue es el inventario de esas consecuencias: cinco familias de aplicaciones que se desbloquean a la vez.

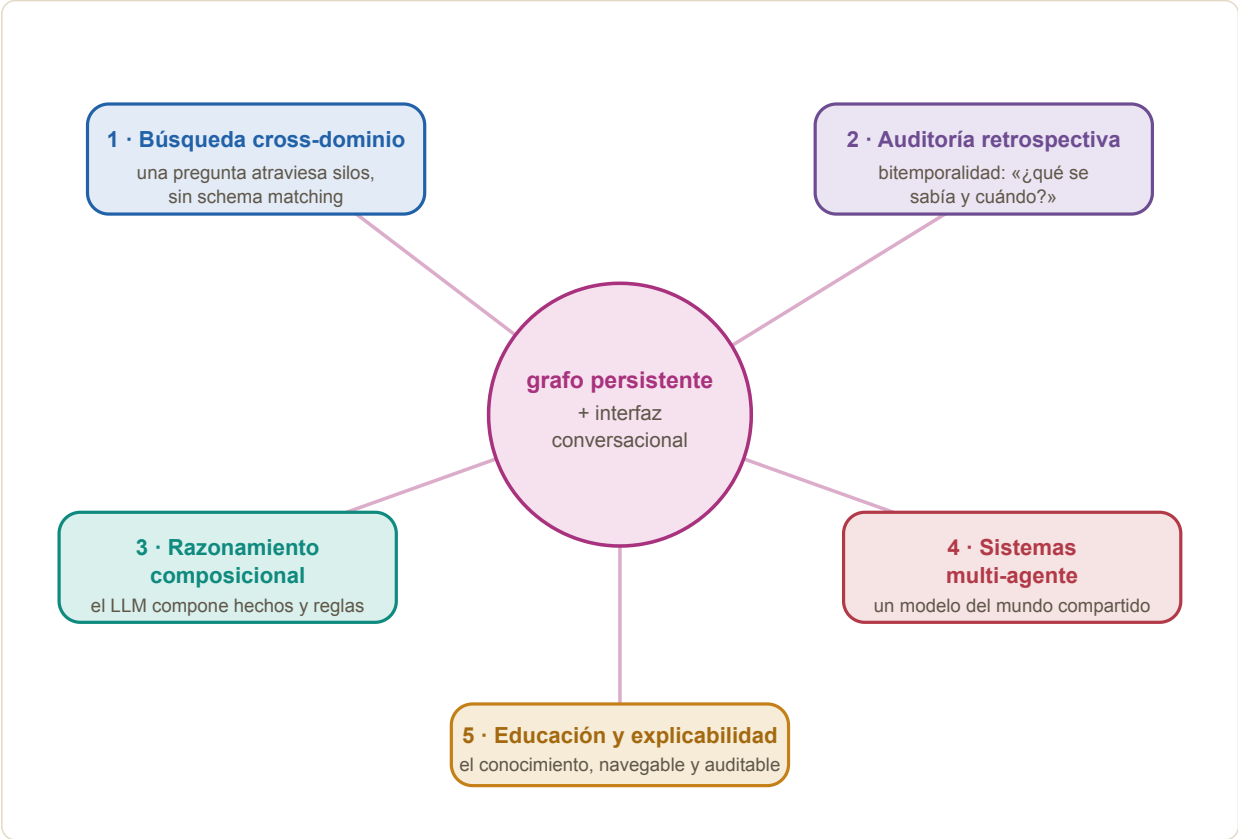


Figura 27.1. Cinco familias de aplicaciones que dejan de ser proyectos a medida cuando un grafo de preguntas vive bajo un modelo de lenguaje. Todas radian del mismo núcleo (un sustrato persistente con interfaz conversacional) y por eso maduran juntas: no hay que construir cinco productos, sino habilitar una sola capa.

Familia 1 · Búsqueda cross-dominio sin *schema matching*

La búsqueda corporativa vive hoy en silos. Las ventas habitan un sistema, el personal otro, el marketing un tercero, las finanzas un cuarto, y la operación algún *legacy* que nadie se atreve a tocar. Cada uno tiene su esquema. Para responder algo tan razonable como *«¿cuál fue el costo total de adquisición de los clientes que cancelaron en sus primeros noventa días?»* hay que cruzar cuatro sistemas, y eso (en la práctica) significa abrir un proyecto de integración con presupuesto propio y meses de calendario.

WQuestions sustituye ese cruce por un sustrato común. Las áreas siguen operando sus sistemas tal cual; lo que cambia es que los hechos relevantes (ventas, contactos, transacciones, bajas) se publican al grafo central, cada cual con su propio dialecto de dominio. La pregunta del director financiero se convierte entonces en una consulta sobre el grafo, que el modelo de lenguaje formula y ejecuta. El *schema matching* (ese cuello de botella histórico de la integración empresarial, donde un analista pasa semanas decidiendo que el campo `cust_id` de un sistema equivale al `id_cliente` de otro) sencillamente desaparece, porque todos los dialectos ya están mapeados contra el mismo catálogo canónico invisible.

DECISIÓN REFERENCIADA

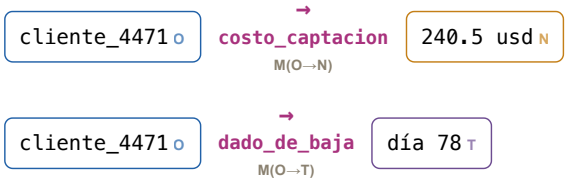
La idea de un catálogo canónico que nadie ve, con el lexicon como única interfaz, es la decisión D8, enunciada en el capítulo 14. Aquí solo cosechamos su consecuencia: dos dialectos que jamás se conocieron resultan, sin embargo, interoperables.

Conviene mostrar el mecanismo, no solo nombrarlo. Tres áreas describen al mismo cliente con claves que no se parecen en nada; el catálogo las reconcilia porque todas responden, sin saberlo, a las mismas preguntas:

JSON

```
// Ventas (CRM)        → dialecto comercial
{ "cust_id": 4471, "ltv_usd": 1820, "estado": "baja" }

// Finanzas (ERP)      → dialecto contable
{ "id_tercero": "C-4471", "costo_captacion": 240.5 }

// Soporte (legacy)    → dialecto operativo
{ "subject": "acct/4471", "churn_day": 78 }
```

Por debajo, los tres registros se resuelven contra el mismo individuo, y el modelo puede responder la pregunta del director sin que un humano haya tendido jamás un puente entre `cust_id`, `id_tercero` y `subject`. La consulta cruza tres dominios y se apoya en un solo eje de identidad:



PRECEDENTE: UNA PROMESA DE HACE UN CUARTO DE SIGLO (31)

Esto no es una idea nueva. Tim Berners-Lee la formuló en el artículo fundacional de la *web semántica* hace ya veinticinco años: una capa de significado sobre la que las máquinas pudieran cruzar datos sin un acuerdo previo, sistema por sistema. Lo que faltaba eran dos piezas. Primero, un modelo lo bastante *simple* para que las áreas lo adoptaran sin un comité de ontólogos (las siete preguntas lo son). Segundo, un traductor lo bastante *fluido* para no exigir vocabulario técnico al usuario final (los modelos de lenguaje lo son). La promesa era correcta; le faltaban estas dos manos.

Familia 2 · Auditoría retrospectiva con bitemporalidad

Esta es la familia que rinde el beneficio más inmediato en los sectores regulados (finanzas, salud, derecho) y la única donde la vigencia temporal no es una comodidad, sino un requisito legal. El principio cabe en una frase: **el sistema nunca olvida, y puede mostrar lo que sabía en cualquier instante del pasado**.

DECISIÓN REFERENCIADA

La reificación de las propiedades cambiantes con rango `inicio` / `fin` (la bitemporalidad) es la decisión D6, del capítulo 9. Su pago se cobra aquí, años después de escrito el primer hecho.

Una corte que pregunta *«¿qué políticas internas estaban vigentes el 30 de junio de 2023?»* obtiene una respuesta exacta, no una reconstrucción aproximada a partir de correos y memorias. Un supervisor financiero que pide *«¿cuál era el límite de exposición de este cliente cuando se firmó la operación?»* recibe el valor histórico, no el de hoy. Un comité que estudia un evento adverso reconstruye *qué se sabía del paciente al momento de prescribir* (y, por tanto, si la decisión fue defendible bajo el conocimiento entonces disponible, que es la única vara justa para juzgarla).

La diferencia con los sistemas tradicionales no está en la *capacidad*: Richard Snodgrass[17] formalizó la bitemporalidad en SQL ya en los años noventa, y bases de datos como Datomic la llevan a producción tratando cada hecho como inmutable. La diferencia está en la **uniformidad**. En un sistema convencional, alguien decide de antemano qué columnas merecen historial y cuáles se sobrescriben; la pregunta retrospectiva tarde o temprano choca contra un *«ese dato no se preservó»*. En WQuestions, todo hecho lleva su vigencia porque la inmutabilidad es la regla, no la excepción cuidadosamente elegida.

DOS TIEMPOS, NO UNO

«Bitemporal» quiere decir que cada hecho carga *dos* líneas de tiempo: **cuándo fue cierto en el mundo** (tiempo de validez) y **cuándo el sistema lo supo** (tiempo de transacción). Por eso la pregunta del fiscal («¿qué sabía el directorio el 15 de noviembre?») es contestable: no pregunta por la realidad de esa fecha, sino por el estado del *conocimiento registrado* en ella. Un sistema con un solo reloj no distingue ambas cosas y, al hacerlas colapsar, pierde justamente lo que un auditor necesita.

Los terrenos donde esto cambia las reglas del juego son concretos: cumplimiento normativo bancario, *due diligence* corporativa, ensayos clínicos longitudinales, registros catastrales, expedientes académicos, archivos periodísticos. En todos, lo que importa no es solo el estado actual, sino la historia auditable que llevó hasta él.

Familia 3 · Razonamiento composicional sobre el conocimiento

Esta es la familia más ambiciosa y la que más promete a largo plazo. La idea: un modelo de lenguaje con acceso al grafo no se limita a *recuperar* información, sino que **razona** combinando piezas que nadie precombinó. Tres preguntas, de tres dominios del repertorio, ilustran el salto:

Patrones de comportamiento. *«¿Qué clientes muestran un patrón de uso parecido al de quien canceló el mes pasado y todavía no se han dado de baja?»* El modelo no busca una fila idéntica; *compone* un perfil a partir de varios hechos y lo proyecta sobre el resto del grafo.

Simulación sobre hechos y reglas. *«Si bajo el plan mensual un 10 %, ¿cuántos clientes con intención registrada de contratarlo se convertirían según el modelo de elasticidad?»* Aquí se combinan hechos del grafo con una regla de negocio y un cálculo.

Reglas más estado más cronología. *«Lista los pacientes cuyo medicamento actual entra en contraindicación con la nueva guía clínica de agosto.»* El modelo cruza el estado vigente de cada paciente con una regla fechada.

Cada una de estas preguntas mezcla hechos, reglas y el razonamiento del propio modelo. WQuestions aporta la materia prima inequívoca; el modelo compone. La vieja frontera entre **base de conocimiento** y **motor de inferencia** se vuelve porosa: el modelo da los saltos, y el grafo le ofrece la huella firme donde apoyar cada pie.

LA TRAMPA: LA ALUCINACIÓN COMPOSITIVA

El riesgo conocido de esta familia tiene nombre: el modelo puede inventar conexiones *plausibles* que no están en el grafo: un «paciente con diabetes» que nunca recibió ese diagnóstico, una «cláusula que justifica a otra» que nadie redactó. La mitigación es exigir **trazabilidad**: toda conclusión debe poder citar los hechos del grafo que la sostienen, con su identificador y su momento. Cuando una respuesta no se puede trazar, el sistema lo declara abiertamente en lugar de fabricar. La diferencia entre un asistente útil y uno peligroso, en sectores serios, es exactamente esta línea.

Familia 4 · Multi-agente con un modelo del mundo compartido

Hasta hace poco, «agente de IA» significaba un único modelo con unas herramientas. Hoy ya se ven sistemas donde **varios agentes especializados colaboran**: uno investiga, otro redacta, otro verifica, otro audita. El cuello de botella es que cada agente vive en su propio contexto, y la coordinación entre ellos resulta frágil: se pasan resúmenes, pierden matices, se contradicen sin notarlo.

WQuestions ofrece a esos sistemas un **modelo del mundo compartido**: un grafo persistente que todos los agentes consultan y al que todos contribuyen. Las decisiones de uno se vuelven hechos consultables por los demás. La cadena de razonamiento de un

agente (*«el agente A diagnosticó X a partir de la evidencia Y, justificado por la regla Z»*) queda registrada con la misma estructura que cualquier otra situación. La coordinación deja de ser implícita y se vuelve **observable**: un agente puede citar literalmente a otro.

TRIPLETAS

```
(diagnostico_001, agente,       lim_clinico_v3)     ∈ M(O, Q)   # un agente de IA actúa
(diagnostico_001, motivado_por, evidencia_023)     ∈ M(O, O)
(opinion_legal_002, agente,      lim_legal_v1)      ∈ M(O, Q)
(opinion_legal_002, motivado_por, diagnostico_001)  ∈ M(O, O)   # ← un agente cita a otro
```

El patrón abre escenarios muy concretos: equipos de auditoría donde un agente sospecha, otro investiga y un tercero verifica; equipos clínicos donde un agente sugiere el diagnóstico, otro revisa contraindicaciones y otro lo registra; equipos legales donde un agente redacta, otro audita el riesgo y otro confronta con la jurisprudencia. Todos sobre el **mismo** grafo, todos con la **misma** trazabilidad, todos dejando un rastro que un humano puede recorrer después.

DECISIÓN REFERENCIADA

Que el rol `agente` lo pueda ocupar un programa (no solo un humano) es la agencia contextual, la decisión D5 del capítulo 9. Aquí rinde su dividendo final.

Aquí la agencia contextual paga su último dividendo. Los modelos de lenguaje entran al sistema como agentes Q de pleno derecho, tratados con la misma uniformidad que una persona o una organización. El sistema no necesita un módulo especial para «agentes de IA»: el modelo ya los admite, porque desde el principio admitió agentes que no eran humanos: sensores, software, instituciones. Lo que se diseñó para un inclinómetro en una mina sirve, sin un solo cambio, para un equipo de modelos colaborando.

Familia 5 · Educación y explicabilidad

Una aplicación menos comercial, pero quizás la más significativa a la larga: WQuestions como **herramienta pedagógica**. Cuando a un estudiante se le explica un dominio complejo (anatomía, derecho constitucional, química orgánica) la dificultad de fondo es que el dominio se presenta como un texto lineal, una sucesión de párrafos, cuando su estructura real es un grafo de conceptos enlazados.

Si el dominio está modelado en WQuestions, el estudiante puede **navegarlo** en lugar de recitarlo: parte de un concepto, ve sus instancias, sigue sus relaciones causales y normativas, lanza consultas exploratorias. *«¿Qué procesos dependen de la disponibilidad de ATP?»*, *«¿qué causa qué en la cascada inflamatoria?»*, *«¿qué cláusulas se justifican mutuamente en este ordenamiento?»*. El libro de texto se vuelve interactivo y consultable; el conocimiento, explorable en vez de memorizable.

Y hay un correlato directo en la explicabilidad de la IA. Cuando un sistema basado en modelos toma una decisión que afecta a alguien (denegar un crédito, sugerir un tratamiento, sancionar un contenido) la pregunta *«¿por qué?»* exige una respuesta auditable. Si esa decisión se construyó sobre un grafo con relaciones canónicas del porqué (`causado_por`, `motivado_por`, `con_finalidad`, `justificado_por`), la cadena de explicación es **literalmente un recorrido del grafo**. No es una racionalización a posteriori, fabricada para sonar convincente: es la reconstrucción del razonamiento que de verdad ocurrió.

RECORDATORIO

El porqué no es un eje, sino cuatro cables distintos. Esa descomposición (del capítulo 10) es lo que convierte una explicación en un camino recorrible, en lugar de un párrafo de texto libre.

Aquí las dos caras se tocan. La misma estructura que deja a un estudiante *navegar* un dominio deja a un regulador *auditar* una decisión: en ambos casos, entender es recorrer aristas con etiqueta. El conocimiento explicable y el conocimiento enseñable son, resulta, el mismo grafo leído con dos propósitos.

Una pregunta, varios dominios

Para ver de golpe por qué esto importa, conviene un ejemplo donde una sola pregunta cruce fronteras que hoy son murallas. Una periodista económica escribe: *«¿Cómo afectó la reforma tributaria de marzo a las ventas del trimestre?»* Esa frase, inocente en apariencia, exige tocar tres mundos que en cualquier organización viven separados: la esfera normativa, los indicadores macroeconómicos y los datos comerciales. El modelo descompone la pregunta en consultas a cada dominio y las vuelve a coser en una respuesta narrativa, con la huella de cada dato a la vista.

Figura 27.2. Una misma pregunta atraviesa tres dominios (normativo K, macroeconómico N y comercial O) que en cualquier organización viven en sistemas distintos. El modelo descompone, consulta y recombina sobre el *mismo* grafo, y devuelve una respuesta narrativa cuyas afirmaciones se pueden rastrear hasta el hecho que las sostiene. Lo que une los tres dominios no es un puente *ad hoc*, sino que los tres responden a las mismas preguntas.

Repara en que ninguno de los tres dominios «sabe» de los otros. La esfera normativa no tiene idea de las ventas; el sistema comercial ignora la inflación. Lo que los hace dialogar no es un conector que alguien programó entre ellos, sino que los tres descomponen su realidad sobre las mismas siete preguntas. La interoperabilidad deja de ser un proyecto y pasa a ser una propiedad del sustrato.

La constante: una identidad estable en el tiempo

Si hubiera que destilar las cinco familias en una sola observación, sería esta. Lo que las hace posibles a todas es la misma propiedad humilde: **el grafo preserva una identidad estable de cada entidad a lo largo del tiempo**. La cardióloga del segundo escenario es el mismo individuo en 2026 y en 2034; el spa que abrió el capítulo es la misma persona jurídica antes y después de expandirse; la cláusula de garantía es la misma cláusula con o sin enmiendas posteriores.

Suena trivial. No lo es. En los sistemas tradicionales, la identidad se reconstruye con *claves foráneas* que apuntan a registros editables, borrables, renombrables. Es una identidad **frágil**: una migración de esquema, un cambio de criterio, una limpieza de datos bienintencionada, y la trazabilidad se rompe sin que nadie lo note hasta que un auditor la pide. WQuestions hace lo contrario: cada individuo recibe un identificador inmutable, los hechos sobre él se acumulan en lugar de sobrescribirse, y los cambios son hechos nuevos con su propia vigencia. La identidad deja de ser una convención y se vuelve **infraestructura**.

> *La identidad estable es la propiedad que hace que decir «este» signifique*

siempre lo mismo.

EL CIMIENTO DE TODO LO ANTERIOR

Sobre esa base se levanta todo lo demás. La auditoría retrospectiva, el razonamiento composicional, el multi-agente, la explicabilidad: las cuatro exigen poder decir *«este individuo, en este momento, era así»* y recibir una respuesta unívoca. Quítale al grafo la identidad estable y las cinco familias se desploman a la vez, porque ninguna sabría ya de qué entidad está hablando.

El sesgo de optimismo

Conviene cerrar matizando. Los escenarios de este capítulo son **plausibles**, no inevitables. Cada uno depende de que varias condiciones se cumplan a la vez, y conviene nombrarlas con precisión:

1 APERTURA DEL DATO

Que las organizaciones acepten exponer su información a un grafo común. No hace falta que sea *público*, pero sí *consultable* dentro de su perímetro.

2 MODELOS MÁS HONESTOS

Que los modelos de lenguaje sigan mejorando no solo en exactitud, sino en su honestidad sobre lo que *no* saben. La trazabilidad sin honestidad es un teatro.

3 CÓMPUTO MÁS BARATO

Que el costo de un asistente conversacional con grafo persistente siga bajando hasta caber en el presupuesto de una organización mediana.

4 UN ROL NUEVO

Que aparezca el **ingeniero de lexicon**: un oficio equivalente al del ingeniero de datos, pero centrado en mapear el vocabulario de un dominio a los roles canónicos.

5 REGULACIÓN QUE ACOMPAÑE

Que la norma habilite en lugar de bloquear, sobre todo en los sectores donde más rinde (salud, finanzas, derecho), que son también los más celosos.

6 ADOPCIÓN GRADUAL

Que el modelo se pueda adoptar por capas, sin un «día cero» en que todo migra. Un dominio publica al grafo; luego otro; el valor crece con cada uno que entra.

Algunas de estas condiciones se están cumpliendo solas; otras exigen empuje deliberado. La especulación es honesta justamente cuando reconoce la diferencia entre lo que ya rueda cuesta abajo y lo que aún hay que empujar cuesta arriba.

El mapa de lo que falta

Conviene terminar señalando el escalón que separa este capítulo del presente. Todas estas aplicaciones suponen una versión *madura* del proyecto. La versión actual (la que el prototipo ejecuta) está, deliberadamente, incompleta:

LO CONSTRUIDO FRENTE A LO PROMETIDO

- Falta **persistencia industrial**: un almacén que aguante volumen y concurrencia reales, no un grafo en memoria de demostración.
- Falta un **motor de inferencia** que derive hechos a partir de reglas, en lugar de exigir que todo esté afirmado de antemano.
- Falta la **bitemporalidad completa** sobre la que descansa la familia 2, hoy solo esbozada.
- Falta un **lexicon poblado** en varios idiomas y dominios, no un puñado de mapeos de ejemplo.
- Faltan **herramientas** para que una organización defina su propio dialecto sin un ingeniero al lado.

Mientras las aplicaciones de este capítulo viven en el futuro, ese trabajo pendiente vive en el presente, muy operativo. Enumerarlo con precisión (como un **mapa de implementación** para quien quiera contribuir o adoptar la propuesta) es lo que queda por hacer, y es a lo que dedicamos el cierre del libro. Pero antes hay una prueba que esta arquitectura debe pasar, más exigente que cualquier dominio industrial: la de describirse a sí misma. A eso vamos.

28

La prueba reflexiva

La carga más exigente no es modelar otro dominio: es pedirle al modelo que se describa a sí mismo. Si las preguntas son universales, también la maquinaria que las procesa debería poder expresarse en preguntas.

Imagina una pantalla partida en dos. A la izquierda, una aplicación de gestión cualquiera: un menú lateral, un formulario de venta con sus campos, una grilla con los últimos registros. Nada que un programador no haya construido cien veces. A la derecha, en un panel angosto, una lista que crece a cada clic: ventaO tiene_campo campo_venta_montoO, y debajo otra, y otra. Ese panel no describe la aplicación: *es* la aplicación. Cada opción del menú, cada campo del formulario, cada columna de la grilla que ves a la izquierda existe porque a la derecha hay una tripleta que lo afirma. Lo que estás mirando es un programa hecho enteramente de hechos WQuestions, y a su lado, los hechos que lo sostienen.

> **HILO DEL LIBRO**
> A lo largo de la Parte V sometimos el modelo a ocho dominios industriales y cuatro escenarios de estrés (capítulo 25). Esta es una carga de otra naturaleza: no apunta hacia afuera, sino hacia dentro.

A lo largo de este libro hemos apretado WQuestions desde muchos ángulos. Lo apretó el lenguaje, con sus nominalizaciones y sus modismos, en el capítulo 15. Lo apretaron ocho dominios que no se parecen en nada, del spa a la operación minera. Lo apretó un sistema de verdad, en producción, cuando hicimos arqueología del yaku. Y lo apretaron, a propósito, cuatro dominios incómodos elegidos para que crujiera. Todas esas presiones tienen algo en común: vienen de fuera. El modelo describe un mundo (ventas, pacientes, partidos, contratos) y la pregunta es si el mundo cabe en sus siete ejes.

Queda una presión más, y es la más exigente de todas: pedirle al modelo que **se describa a sí mismo**. No modelar un dominio externo, sino construir una herramienta (un programa con menús, formularios, tipos de dato y reglas de comportamiento) cuya estructura, cuyos tipos y cuya conducta sean, ellos mismos, hechos WQuestions. Si la tesis del libro es cierta, si las preguntas son la base universal de la información, entonces la propia maquinaria que manipula información tendría que poder expresarse en esas preguntas. Este capítulo cuenta qué pasó cuando lo intentamos. No venimos a presumir de una varita mágica; venimos, como en el capítulo 15, con honradez: a mostrar qué aguantó, qué se dobló para aguantarlo y qué quedó al descubierto.

> **LA PRUEBA REFLEXIVA**
> Un modelo de la información merece confianza cuando es capaz de **describirse con sus propios medios**. Si los ejes, los predicados, las signaturas y el esquema de la herramienta viven como tripletas dentro del mismo grafo que todo lo demás, el modelo deja de ser una teoría aplicada a los datos y pasa a ser datos él mismo.

Una aplicación hecha de preguntas

Construimos, sobre el prototipo, una pequeña aplicación de gestión: un menú navegable, formularios para dar de alta y editar, grillas para consultar, y entidades de negocio (personas, productos, ventas, compras) con sus campos. Lo decisivo no es la

aplicación; es **de qué está hecha**. Cada opción del menú, cada campo de cada formulario, cada tipo de dato, cada relación entre entidades y cada acción que el programa ejecuta es una tripleta `(sujeto, rol, valor)` en un único grafo.

El menú es un objeto reificado que `tiene_opcion` a otros objetos; cada opción `tiene_accion` una acción; cada acción es `instancia_de` un verbo (`mostrar_texto` , `abrir_formulario` , `abrir_grilla` , `guardar`) que un evaluador genérico interpreta. El esquema de una entidad también es dato: la entidad `tiene_campo` a descriptores que declaran su etiqueta, su `tipo_dato` y su `orden` . Un registro es un individuo `instancia_de` su tipo, con un hecho por campo. No hay, en ninguna parte, una tabla `venta` con columnas fijas ni una clase `Formulario` con atributos cableados. Hay hechos, y un motor que los lee.

TRIPLETAS · EL MENÚ ES DATO

```
(menu_principal, instancia_de, menu)             ∈ M(0, K)
(menu_principal, tiene_opcion, opcion_ventas)    ∈ M(0, 0)
(opcion_ventas,  etiqueta,     "Ventas")         ∈ M(0, K)
(opcion_ventas,  tiene_accion, abrir_grilla_venta) ∈ M(0, 0)
(abrir_grilla_venta, instancia_de, abrir_grilla) ∈ M(0, K)
(abrir_grilla_venta, sobre,        venta)         ∈ M(0, K)
```

La consecuencia se ve en la pantalla del propio programa, en ese panel de la derecha que abrió el capítulo: al lado de cada vista, un **inspector** muestra las tripletas que la sostienen. Lo que ves es, literalmente, lo que hay en el grafo. La interfaz no es una capa que traduce una base de datos a botones; es una proyección directa de los hechos.

> *Lo que ves en la pantalla es, sin intermediarios, lo que hay en el grafo.*
>
> EL PRINCIPIO DEL INSPECTOR

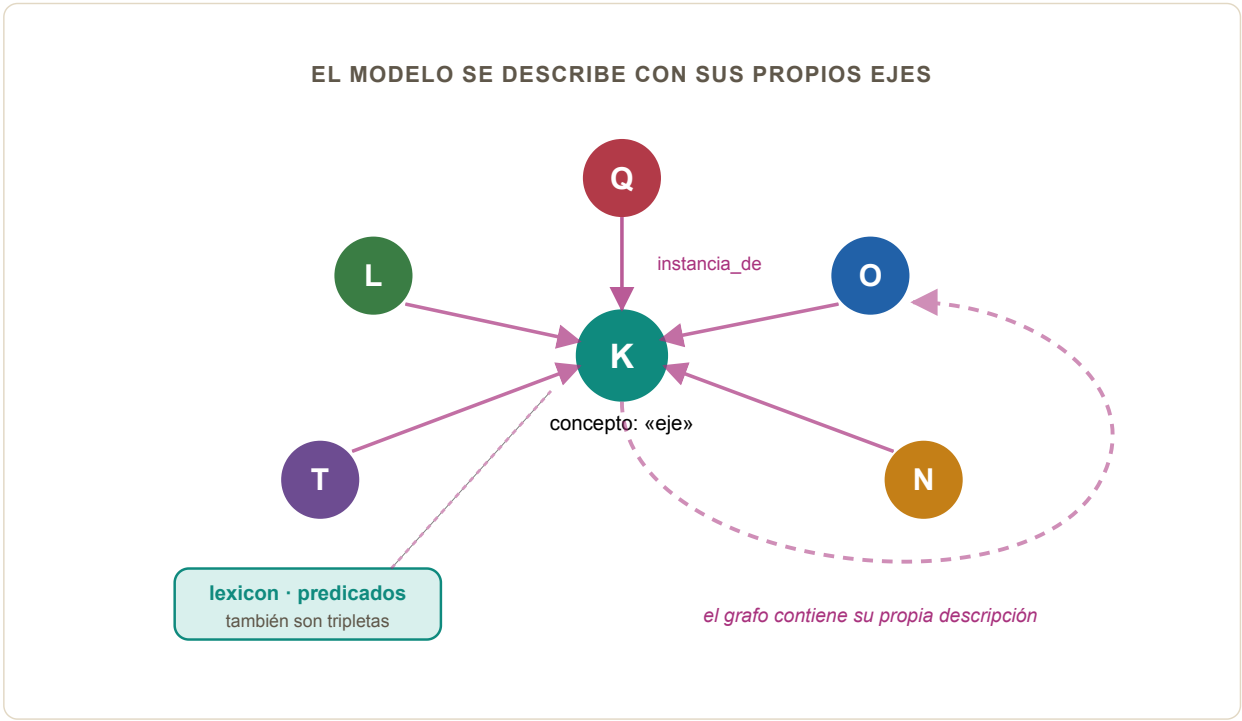


Figura 28.1. El modelo reflejándose a sí mismo. Cada uno de los siete ejes es, a su vez, una `instancia_de` el concepto «eje» (una clase del eje K), y el lexicon y los predicados viven como tripletas en el mismo grafo. La flecha de trazos que regresa sobre el anillo es la reflexión: el grafo contiene su propia descripción.

La prueba reina

Toda la apuesta se puede resumir en una sola operación. Si la estructura y el comportamiento son de verdad datos, entonces **agregar estructura debería costar insertar hechos, no escribir código**. Esa es la prueba reina: ¿cuánto cuesta agregar un campo nuevo a una entidad que ya existe?

Lo medimos en vivo, con el sistema corriendo, agregando el campo «Documento» a la entidad `venta`. Estas son las tripletas que insertamos, y nada más:

TRIPLETAS · CAMPO NUEVO = 5 HECHOS

```
(campo_venta_documento, instancia_de, campo)   ∈ M(O, K)
(venta, tiene_campo, campo_venta_documento)    ∈ M(K, O)
(campo_venta_documento, tipo_dato, texto)      ∈ M(O, K)
(campo_venta_documento, orden, 5)              ∈ M(O, N)
(campo_venta_documento, rol, documento)        ∈ M(O, K)
```

Cinco tripletas. Ninguna línea de código en el motor, en el servidor ni en la interfaz. Al recargar, el formulario dibujó el campo, la grilla le sumó la columna y el guardado lo persistió, porque las tres operaciones leen el esquema del grafo en lugar de conocerlo de antemano. Ese número (cinco hechos, cero código) es el resultado central del experimento, y la confirmación más limpia de la primera mitad de la tesis: **la estructura es dato**.

POR QUÉ IMPORTA

En un sistema convencional, «agregar un campo» toca la migración de la base, el modelo del servidor, el formulario, la grilla y a menudo la API. Cinco lugares, cinco oportunidades de error. Aquí, un solo lugar: el grafo.

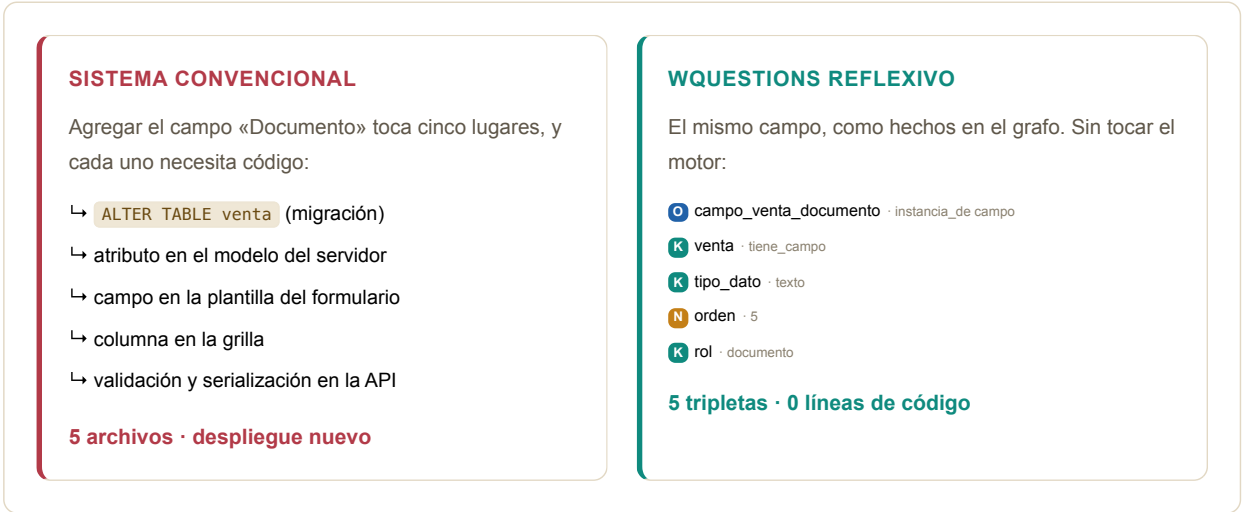


Figura 28.2. La prueba reina, antes y después. A la izquierda, el campo nuevo se reparte en cinco archivos y exige un despliegue. A la derecha, es un puñado de hechos que el motor ya sabe leer. El color de cada hecho marca el eje de su valor.

Lo que la carga confirmó

Más allá del número, el experimento puso a prueba tres ideas que el libro había defendido en abstracto. Las tres aguantaron en concreto.

El comportamiento también es dato

Una y otra vez, a lo largo del libro, apareció la figura del *evaluador externo*: el motor que recorre el grafo y ejecuta lo que los hechos prescriben, separado del grafo, que solo almacena. Aquí ese evaluador dejó de ser una promesa y pasó a ser código que corre: un despachador genérico que, dado un verbo (un tipo del eje K), ejecuta la conducta asociada.

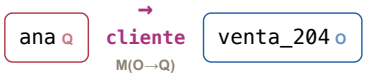
PYTHON · EL DESPACHADOR NO CONOCE DOMINIOS

```
def ejecutar(self, accion):
    """El motor no sabe qué es 'venta' ni 'compra'.
    Lee el verbo de la acción y delega en el manejador del verbo."""
    verbo = self.grafo.objeto(accion, "instancia_de")     # p.ej. abrir_grilla
    manejador = self.verbos[verbo]                          # tabla genérica
    objetivo = self.grafo.objeto(accion, "sobre")          # p.ej. venta
    return manejador(objetivo)                              # nada cableado

# Agregar la transacción "compras" no toca este código:
# basta declarar el verbo, la entidad y sus campos como hechos.
```

Agregar una transacción entera nueva («compras», hermana de «ventas») costó esencialmente datos, no motor. El comportamiento vive en el grafo y se interpreta desde afuera, tal como el modelo prometía.

El grafo compartido cae solo

El capítulo 1 abrió con la torre de Babel: cada sistema con su esquema, incapaz de hablar con el de al lado. El antídoto del libro es un único piso de hechos donde las entidades se reúsan. Lo comprobamos modelando «cliente» y «proveedor» no como tipos distintos, sino como **roles contextuales**: el nombre del campo en la venta o en la compra. Es la agencia contextual de la decisión D5 en acción.



La misma persona, `ana`, es cliente de una venta y proveedora de una compra sin duplicarse: un solo individuo, dos papeles según el contexto. El anti-Babel no fue una aspiración que hubo que forzar; cayó solo en cuanto dejamos de modelar el rol como un tipo.

La vigencia es la forma natural del «editar»

> **REFERENCIA**
> La bitemporalidad (reificar las propiedades que cambian con un rango de inicio y fin) se enuncia formalmente en el capítulo 9. Aquí solo la usamos.

La edición de un dato se implementó sin borrar nada: editar un valor es asentar un hecho nuevo y leer el más reciente. Es bitemporalidad en miniatura (la vigencia de la decisión D6): historial gratis, ninguna sobreescritura destructiva. Y resultó ser la forma *natural* de implementar el «update» de un CRUD, no un agregado teórico que hubo que justificar.

> **TRES PROMESAS, COBRADAS**
> El evaluador externo, el grafo compartido y la bitemporalidad eran, hasta aquí, decisiones de diseño defendidas sobre la pizarra. El experimento reflexivo las cobró las tres a la vez, en un solo programa pequeño: comportamiento que se interpreta, entidades que se reúsan por contexto y un historial que aparece sin esfuerzo.

La auto-corrección: `instancia_de` pasa a `V→K`

El experimento no solo validó: **forzó una corrección del prototipo**, y vale la pena contarla, porque es el método del libro operando sobre sí mismo. Para clasificar a una persona (decir «Ana es un cliente») hace falta el hecho `(ana, instancia_de, cliente)`.

Pero `ana` vive en el eje Q, es un agente, y el prototipo había restringido `instancia_de` a sujetos del eje O. La carga reveló la contradicción de inmediato.

Lo irónico es que este mismo libro, en el capítulo 3, ya escribía sin parpadear hechos como estos:

TRIPLETAS · CLASIFICAR ES UNIVERSAL

```
(vendedor_17,    instancia_de, profesion_vendedor) ∈ M(Q, K)
(municipalidad_centro, instancia_de, organizacion) ∈ M(Q, K)
(tienda_central, instancia_de, punto_de_venta)     ∈ M(L, K)
(camiseta_88,    instancia_de, camiseta)           ∈ M(O, K)
```

Sujetos en Q, en L, en O: la teoría siempre supuso que clasificar es una operación universal. El prototipo estaba, sencillamente, *detrás* de su propia teoría. La corrección fue generalizar `instancia_de` de `O→K` a `V→K`, donde **V** es un comodín que significa «cualquier eje de valor».

EL COMODÍN V

V denota cualquiera de los seis ejes de valor (Q O L T N K), por oposición al eje estructural M de los predicados. Una signatura `V→K` dice: «el sujeto puede ser un individuo de cualquier eje de valor, y el objeto es una clase». Es la forma de declarar que una operación (como clasificar) no pertenece a un eje particular.

Ese comodín no resuelve solo este caso. Es exactamente el mecanismo que el capítulo 30 reclama para relajar otras signaturas demasiado estrechas (pensemos en `paciente` o `partes`, hoy fijadas a `O→Q` cuando deberían admitir `O→V`). La fricción, sometida a carga, no abrió un agujero en el modelo: lo empujó a alinearse con lo que ya afirmaba.

> *La presión no abrió un agujero en el modelo: lo empujó a alinearse con lo que ya afirmaba.*
>
> SOBRE LA AUTO-CORRECCIÓN

Las fricciones nuevas: dos cerradas, una abierta

La presión reflexiva destapó tres fronteras que los ocho dominios industriales no habían tocado. Dos las cerramos dentro del mismo experimento; la tercera quedó señalada como trabajo pendiente.

El texto libre. Los siete ejes son semánticos, no tipos primitivos: no existe un eje «string». Un nombre como «Ana», el contenido de un mensaje, el número de un documento (texto arbitrario) no tienen casa propia. La decisión, fiel a la teoría, fue alojar el texto libre como un **literal en K, minteado y único**, con su valor en la etiqueta y una marca que lo distingue de una *categoría controlada*. Así, dos personas llamadas «Ana» no terminan compartiendo un mismo individuo.

El catálogo como dato. El catálogo canónico tipa con fuerza el núcleo del modelo, pero los campos que el usuario define como datos pasaban sin validar, bajo política liberal. La carga lo expuso: el tipado del esquema vivía a medias en el código. Lo cerramos haciendo que **cada campo derive su signatura de sus propios datos** (dominio del eje de la entidad, rango de su `tipo_dato`) y la registre en el catálogo. Escribir en un campo dinámico ahora se valida como un rol canónico.

LITERAL EN K · CATEGORÍA CONTROLADA EN K

Dos cosas conviven en el eje **K** y conviene no confundirlas:

Literal

Texto arbitrario. Minteado y *único*: cada «Ana» es su propio nodo. No se comparte ni se reúsa entre sistemas.

Categoría controlada

Vocabulario cerrado (unidades monetarias, estados de un trámite). Es una clase nombrada y *compartida*; el mismo nodo vale para todos.

El catálogo canónico es invisible para el usuario final (es el lexicon su interfaz, como fija la decisión **D8**), pero la carga reflexiva lo obligó a abarcar también lo que el usuario crea sobre la marcha. El tipado dejó de vivir en Python y pasó a vivir en el grafo:

TRIPLETAS · EL CAMPO DECLARA SU PROPIA SIGNATURA

```
(rol_documento, dominio, O)        ∈ M(K, K)
(rol_documento, rango,   K)        ∈ M(K, K)
(rol_documento, etiqueta_en, "Documento") ∈ M(K, K)
# el motor valida el guardado leyendo estas tres tripletas:
# venta (O) escribe documento (K) → coincide con dominio O, rango K → válido
```

El humano. Queda en pie la tercera frontera, y no es del modelo, sino del lector. Cuando *todo* es `(sujeto, rol, valor)`, se pierde el andamiaje de tablas y formularios con nombre que normalmente carga el significado. El poder de la abstracción reflexiva tiene un peaje cognitivo. El inspector («lo que ves es lo que hay») re-concreta lo suficiente para no ahogarse, pero la pieza que falta es hacer de las **vistas y proyecciones con nombre** ciudadanas del grafo: datos, no accidente de la interfaz. Es trabajo pendiente, y reaparece como tal en el capítulo que sigue.

LA FRONTERA QUE NO CERRAMOS

Un sistema donde todo es una tripleta gana en uniformidad lo que pierde en asideros. El programador acostumbrado a una tabla `venta` con columnas a la vista tiene, de pronto, solo hechos. La re-concreción (darle nombre y forma a las vistas, dentro del propio grafo) no es un lujo cosmético: es lo que vuelve usable un modelo que, por construcción, disuelve las categorías familiares.

Honradez intelectual

Conviene nombrar el vértigo, porque es un hallazgo y no un defecto del lector. Un modelo capaz de describirse a sí mismo es, por construcción, mareante: en el límite no hay «tablas» ni «pantallas», solo preguntas sobre preguntas. ¿Qué es un campo? Una clase. ¿Qué es esa clase? Una instancia del concepto «campo». ¿Y ese concepto? Otro nodo del grafo, descrito por más tripletas. La torre no tiene fondo, y al principio eso desconcierta.

Pero ese mareo no invalida la propuesta; la confirma. Y, al mismo tiempo, explica por qué un modelo así necesita prosa, ejemplos y una capa de re-concreción para que las personas puedan usarlo. Es, en el fondo, la razón de ser de este libro: **el grafo es operable; el texto es lo que lo vuelve comprensible**.

Que el modelo pueda describir su propia herramienta es la evidencia más fuerte que el prototipo produjo. Más que ocho dominios distintos, es el modelo hablando de sí mismo sin salirse de sus siete ejes: corrigiéndose cuando una signatura quedó corta, absorbiendo como datos lo que un sistema convencional cablearía en código, y midiendo su propia ambición en un número limpio (cinco hechos, cero líneas).

EL SALDO DE LA PRUEBA REFLEXIVA

La primera mitad de la tesis (**estructura y comportamiento son datos**) queda saldada, y medida. Lo que sigue es la otra mitad: el inventario de lo que todavía falta para que esto deje de ser un prototipo y se vuelva infraestructura de uso diario. Antes de ese inventario, sin embargo, hay una pregunta que un grafo compartido vuelve urgente: si todos los hechos viven en un mismo piso, ¿quién puede ver —y preguntar— qué?

A esa pregunta (la seguridad y la privacidad del grafo compartido) dedicamos el próximo capítulo. Y al inventario de lo pendiente, el siguiente.

29

Seguridad y privacidad

Cuando todos los sistemas hablan el mismo idioma, preguntar deja de tener fricción. Y ahí aparece la pregunta que importa: ¿quién puede preguntar qué?

Son las diez y cuarto de la mañana. Un analista de una aseguradora abre su consola de evaluación de riesgo. Tiene delante la solicitud de un seguro de vida. La solicitante es Vega, la misma paciente que en el primer capítulo llegó de madrugada a urgencias con una historia clínica que nadie logró reunir a tiempo. El analista quiere saber una sola cosa: si Vega tiene antecedentes que encarezcan la prima. Su sistema es moderno y, como todo en este libro, habla en preguntas. Así que formula una consulta perfectamente legible: *¿qué situaciones de clase* `diagnostico_medico` *tienen a esta persona como paciente?*

El grafo responde sin titubear: una arritmia, vigilada desde hace dos años, con un episodio de descompensación documentado. La prima se dispara. En el peor caso, la aseguradora niega la cobertura. Y aquí está lo inquietante: **nadie hackeó nada.** No hubo una brecha, ni una contraseña filtrada, ni un empleado vendiendo registros. El sistema funcionó *exactamente como se diseñó*: alguien preguntó y obtuvo una respuesta precisa y confiable. El mismo grafo que ante un médico de guardia reúne en segundos una historia clínica dispersa y salva una vida, ante un analista a media mañana le niega el seguro.

> **MARCO**
> La madrugada de Vega abrió el libro: una historia clínica que existía pero que nadie pudo reunir a tiempo. Aquí esa misma historia vuelve por su lado oscuro. Por sí sola, la interoperabilidad no distingue entre el médico que salva una vida y el analista que sube una prima.

Esta es la objeción más seria que se le puede hacer al modelo, y no la voy a esquivar. Un grafo compartido en el que cada persona tiene una identidad estable es, por defecto, **una infraestructura de vigilancia**. No es un efecto colateral: es la cara opuesta de su mayor virtud. Las leyes de protección de datos del mundo (el RGPD europeo, la HIPAA estadounidense, la Ley 29733 peruana) existen justamente para impedir lo que WQuestions vuelve fácil: cruzar, sin esfuerzo, datos que vivían separados. Si la fuerza del modelo es que el paciente, el cliente y el lugar son *los mismos* en todos los dominios, esa misma fuerza permite que el banco vea el diagnóstico y la clínica vea la deuda.

> **LA OBJECIÓN, SIN MAQUILLAJE**
>
> Un modelo que vuelve fácil preguntar cualquier cosa sobre cualquiera sirve tanto para la mejor medicina como para el peor abuso. La diferencia no está en la potencia del grafo, sino en una sola política: **quién puede preguntar qué**. El resto del capítulo trata de meter esa política dentro del modelo, no de pegarla por fuera con cinta.

Lo que sigue defiende dos cosas. Primero, que el problema no solo tiene solución: bien entendido, el modelo ofrece *mejores* herramientas para resolverlo que la arquitectura relacional a la que reemplaza. Segundo, que queda un frente que ningún modelo enlazable resuelve todavía. Pero antes de la ingeniería hay que deshacer un malentendido, porque la primera respuesta no es técnica.

Un idioma, no una base de datos central

La imagen que produce pánico, y con razón, es la de un gran depósito central: un único servidor donde el historial completo de cada persona cuelga de un identificador, y donde cualquiera con credenciales puede consultarlo. Si WQuestions fuera *eso*, sería indefendible y este capítulo no tendría nada que defender.

No lo es. El grafo compartido es **semántico, no físico**: lo que se comparte es un significado común, no un disco común. El banco, la clínica, la endocrinóloga y el cardiólogo no comparten una base de datos. Comparten un *idioma de preguntas*: los siete ejes y el lexicon que traduce hacia ellos. Cada uno sigue guardando sus propios hechos en su propio almacén, bajo su propio control y su propia llave. La clínica no lee el grafo del banco. Lo que ocurre es que, cuando hace falta interoperar, y solo con autorización, los dos grafos hablan la misma lengua. Y la fusión que antes era un proyecto de integración de seis meses se vuelve sencilla.

> *Lo que se comparte es el idioma de las preguntas, no la memoria de los hechos. La interoperabilidad vive en el borde, bajo consentimiento; no en un cerebro central que todo lo sabe.*

LA DISTINCIÓN QUE LO CAMBIA TODO

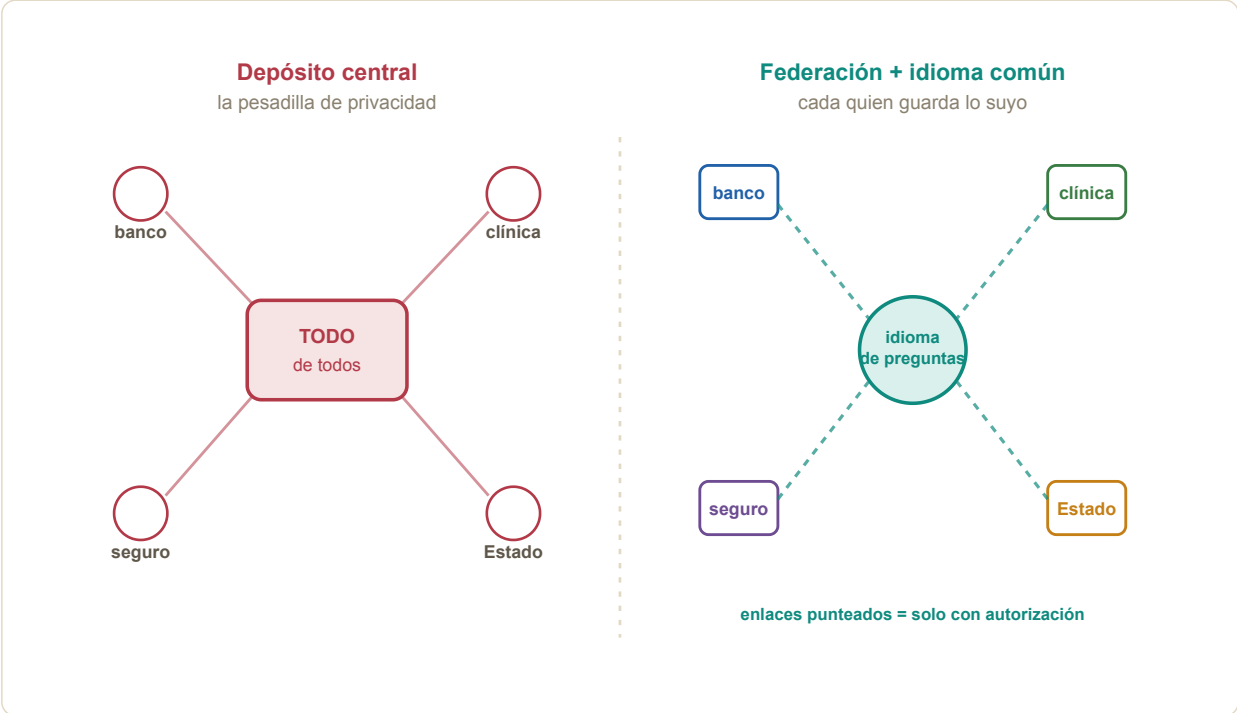


Figura 29.1. A la izquierda, el depósito único donde todos leen de todos: la imagen que produce pánico, y con razón. A la derecha, lo que de verdad es WQuestions: cada parte conserva sus propios hechos bajo su propia llave, y solo los une, bajo consentimiento, un K idioma común de preguntas. La privacidad empieza por la forma de la red, no por las reglas que le pongas encima.

Releída así, la escena del médico de guardia cambia de sentido. No fue que el médico «entrara al expediente global de Vega». Fue que cada parte (los dos hospitales, la endocrinóloga, el cardiólogo) **publicó hacia un espacio que la paciente había autorizado de antemano** para emergencias. Y como todos esos hechos hablaban el mismo idioma, se ensamblaron solos. La interoperabilidad no exige un cerebro central; le bastan un idioma compartido y un permiso. Eso disuelve el susto inicial, pero abre la pregunta de ingeniería de verdad: si cada parte controla su propio acceso, ¿con qué lo controla? Aquí el modelo deja de defenderse y empieza a atacar.

El permiso es un hecho

En una arquitectura relacional, el control de acceso es una capa aparte, escrita en un lenguaje distinto del de los datos: tablas de permisos, jerarquías de roles, sentencias `GRANT` y `REVOKE`, reglas sueltas por todo el código. Es un sistema que corre en paralelo a los datos. Y los sistemas paralelos terminan igual: *se desincronizan*. La política dice una cosa, el código hace otra y nadie sabe cuál de las dos manda de verdad.

En WQuestions no hay capa aparte, por una razón que a estas alturas del libro debería sonar inevitable: **el permiso es, él mismo, un hecho del grafo**. Es una situación reificada, es decir, un hecho descrito con las mismas siete preguntas que cualquier otro. «Que el cardiólogo Rosales pueda ver el diagnóstico de arritmia de Vega» no es una fila en una tabla de permisos: es una afirmación del grafo, con su quién, su qué, su agente y su cuándo. El control de acceso habla el mismo idioma que aquello que controla.

> **RECUERDA**
> Que un permiso pueda modelarse como situación no es un truco inventado para este capítulo. Es la regla general de reificación que ya viste en acción: un evento o una relación se vuelve un nodo en **O** cuando necesita atributos propios. Y el permiso los necesita todos: titular, alcance, vigencia y procedencia.

El permiso, descrito con el propio modelo: una situación cuyos ejes son su contenido

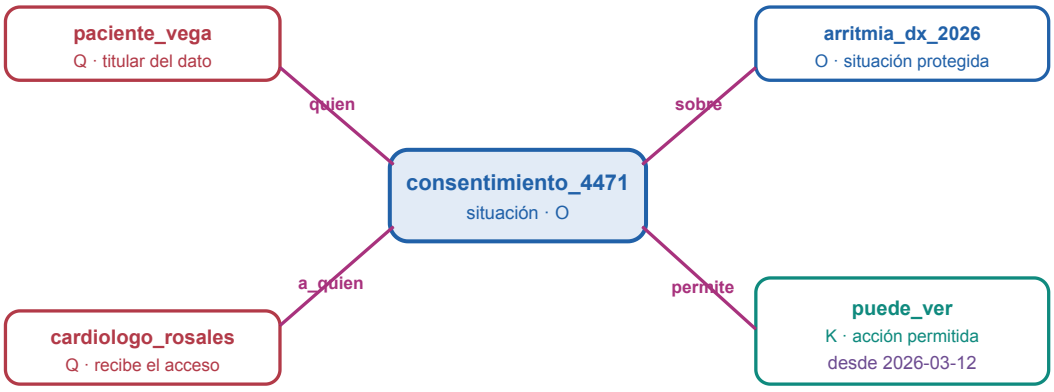
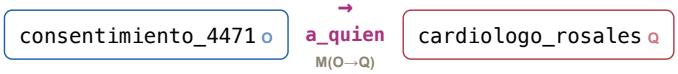


Figura 29.2. El permiso no es una entrada en un sistema ajeno: es una situación reificada en **O**, cuyos cables **M** apuntan a *quién* recibe el acceso, *sobre qué* situación, *qué* acción y *desde cuándo*. El control de acceso habla el mismo idioma que el dato que protege.

Visto como tripletas, el consentimiento se descompone en cables hacia cada eje, igual que cualquier otro hecho del libro:



TRIPLETAS

```
(consentimiento_4471, instancia_de, consentimiento)      ∈ M(O, K)
(consentimiento_4471, quien,        paciente_vega)       ∈ M(O, Q)   # titular
(consentimiento_4471, a_quien,      cardiologo_rosales)  ∈ M(O, Q)   # beneficiario
(consentimiento_4471, permite,      puede_ver)           ∈ M(O, K)   # acción
(consentimiento_4471, sobre,        arritmia_dx_2026)    ∈ M(O, O)   # alcance
(consentimiento_4471, desde,        2026-03-12)          ∈ M(O, T)   # vigencia
(consentimiento_4471, caracter,     revocable)           ∈ M(O, K)
```

Y ese mismo consentimiento, escrito para viajar entre sistemas, no usa un formato especial de seguridad: es un hecho más del grafo, con la misma forma que un diagnóstico o una transferencia.

JSON

```
{
  "id": "consentimiento_4471",
  "instancia_de": "consentimiento",
  "quien": "paciente_vega",
  "permite": "puede_ver",
  "sobre": "arritmia_dx_2026",
  "a_quien": "cardiologo_rosales",
  "desde": "2026-03-12",
  "caracter": "revocable"
}
```

Que el permiso sea un hecho, y no un sistema aparte, trae tres regalos. Conviene verlos uno por uno, porque cada uno cura un dolor clásico del control de acceso relacional.

1 CONSENTIMIENTO CON FECHA

No es una casilla marcada en un formulario perdido: es una situación que dice *quién* consintió *qué*, para *quién* y *desde cuándo*. Y como es un hecho, tiene su propia procedencia: quién recogió el consentimiento, cómo y bajo qué versión de los términos.

2 REVOCAR ES PUBLICAR

Cuando Vega retira el permiso, no se edita ni se borra el hecho anterior: se añade uno nuevo que lo deja sin efecto. El acceso caduca en el acto y queda constancia, auditable, de que existió y de cuándo terminó. El sistema no olvida que hubo permiso; recuerda, además, que se revocó.

3 AUDITORÍA GRATIS

Cada acceso es también una situación: «el analista del seguro consultó `arritmia_dx_2026` el 2028-02-01 a las 10:14». El registro de auditoría no es un sistema de bitácoras que haya que construir y proteger aparte: son más hechos en el mismo grafo.

El segundo punto merece una palabra más, porque es donde el modelo y la ley se cruzan con elegancia. «Revocar es publicar» funciona gracias a la **vigencia temporal** que ya conoces: ante dos hechos que se contradicen, gana el último que sigue vigente. Retirar un permiso no borra nada; añade un hecho nuevo, con su propia marca de tiempo, que cancela al anterior. Así el grafo conserva la historia completa del consentimiento, imprescindible para auditar, y a la vez sabe sin dudas qué vale *ahora*.

CRUCE

La regla que permite «revocar» sin borrar (gana el último hecho vigente) es la misma bitemporalidad T que el libro fija al tratar las situaciones y la vigencia. El control de acceso no inventa mecanismos propios: reutiliza los del tiempo.

«QUIÉN VIO QUÉ Y CUÁNDO», EN UNA LÍNEA

Como cada acceso es un hecho, la pregunta forense que en una arquitectura clásica obliga a cruzar bitácoras dispersas, rotadas y en formatos distintos, aquí es una consulta más sobre los mismos siete ejes:

TRIPLETAS

```
# ¿Quién accedió a la situación arritmia_dx_2026, y cuándo?
?acceso  instancia_de  acceso_a_dato
?acceso  sobre         arritmia_dx_2026
?acceso  quien         ?solicitante      # Q  → quién preguntó
?acceso  cuando        ?momento          # T  → cuándo
```

Redacción por eje

Hay una diferencia de grano fino que juega a favor del modelo, y es quizá su ventaja más subestimada en privacidad. Una base relacional da acceso por tabla o, con bastante esfuerzo, por fila: o ves el registro entero de la transacción, o no ves nada. WQuestions da acceso por **hecho**, e incluso por **eje**.

Como cada situación parte la realidad en coordenadas separadas y de tipo conocido, se pueden mostrar unas y tapar otras sin romper nada. Un auditor financiero puede recibir la prueba de que una transacción *ocurrió* (su O qué y su T cuándo) sin ver nunca el N cuánto ni la Q contraparte. Un investigador epidemiológico puede saber que hubo un diagnóstico de cierta clase, en cierta fecha y cierto lugar (K, T y L), con el Q quién guardado bajo llave.

Figura 29.3. Redacción por eje: la misma situación servida a dos solicitantes. El auditor ve que la transferencia O ocurrió y T cuándo, pero el N monto y la Q contraparte quedan bajo candado. Mostrar a cada quien solo lo que le toca viene de la propia forma del hecho, no se añade después.

Esto no es un parche encima del modelo: es la propia forma del hecho atómico haciendo trabajo de privacidad. La **minimización de datos** (el principio legal de no entregar más de lo necesario para cada fin) deja de ser una promesa de buena conducta y pasa a ser algo que la estructura del dato expresa con precisión de bisturí. En una tabla, «mostrar el hecho pero ocultar el monto» exige una vista a medida, todo un proyecto. Aquí basta con decir qué ejes se revelan.

Esa distinción se vuelve crítica en un caso concreto: cuando la pregunta es por el agregado. A veces quieres que alguien sepa *cuántos* pacientes tienen cierta condición sin permitirle llegar a ninguno en particular; el control de acceso, entonces, tiene que separar contar de ver.

Las tensiones que quedan crudas

No todo está resuelto. Quedan tres tensiones reales, y la última no la resuelve nadie todavía, para ningún modelo de datos enlazable. Las pongo sobre la mesa sin maquillaje, porque el modelo se defiende mejor cuando no finge.

El sistema que nunca olvida frente al derecho al olvido

La vigencia temporal cancela un hecho, pero no lo borra; el RGPD, en cambio, da derecho a la *eliminación* de verdad. Son dos exigencias que parecen compatibles y no lo son del todo. La salida pasa por separar dos cosas que el lenguaje confunde: que un hecho **deje de ser verdad** y que su contenido **se destruya**. La técnica concreta se llama *crypto-shredding*, «triturado por cifrado»:

el contenido de los ejes de valor se guarda cifrado, y «borrar» consiste en destruir la llave. Sin la llave, lo que queda es ruido imposible de recuperar, aunque el nodo de la situación siga ahí para no romper las referencias y poder auditar.

> **POR QUÉ AQUÍ EL BORRADO ES MÁS FACTIBLE, NO MENOS**
>
> Como cada eje dice *con claridad* dónde vive cada dato personal (el nombre en Q, el monto en N, la dolencia en K), borrar a propósito un dato es **más** fácil aquí que en el amasijo relacional, donde el mismo dato sensible aparece desperdigado por columnas, índices, bitácoras y copias de respaldo de medianoche. El modelo sabe dónde apuntar la tijera.

La identidad estable es a la vez el arma y la herramienta

El poder del modelo viene de que la persona es la misma en todos los dominios; y eso es, justamente, lo que abre la puerta a los ataques de re-identificación, los que vuelven a poner nombre a un dato anónimo. La defensa se llama **seudónimos pareados**: la persona sigue siendo una sola por dentro, pero cada relación la ve con un identificador propio, válido solo en ese dominio. Es como usar un nombre distinto en cada ventanilla, donde solo un árbitro autorizado sabe que detrás de todos está la misma persona. Así, cruzar dos identificadores deja de ser gratis: pasa a exigir un permiso explícito.

> **CRUCE**
>
> El «enchufe» de identificadores que se conecta al tratar el zócalo categórico (capítulo 3) es justo el punto donde esta política puede intervenir: averiguar que dos identificadores son la misma persona se vuelve una operación con permiso, no un dato de libre acceso.

El frente genuinamente abierto: inferencia y agregación

Aun con permisos por hecho y con la redacción por eje, un modelo que unifica las preguntas vuelve *más* fácil el ataque por combinación: varios hechos inofensivos por separado, al cruzarse, reconstruyen lo que ninguno mostraba solo. Dónde vives, a qué hora sales de casa, qué farmacia frecuentas: ninguno es secreto, pero juntos te identifican y hasta dicen qué enfermedad tienes.

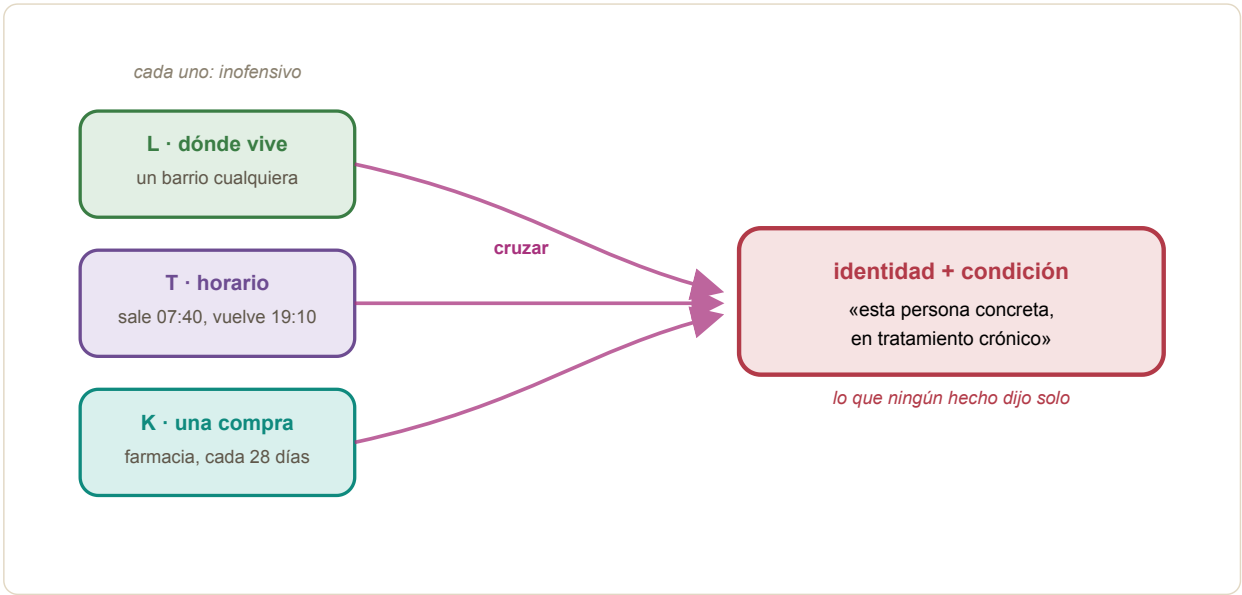


Figura 29.4. El ataque por agregación: un L lugar, un T horario y una K compra (inofensivos por separado) convergen y reconstruyen una identidad y una condición que ninguno revelaba solo. Es el frente que ningún modelo enlazable resuelve por completo.

Esto **no lo resuelve bien nadie**, y WQuestions, justo por facilitar el enlace desde el diseño, está obligado a tomárselo *más* en serio, no menos. Existen defensas serias en el momento de responder una consulta: el *k*-anonimato (no contestar si la respuesta señala a menos de *k* personas) y la privacidad diferencial (añadir un poco de ruido a las cifras agregadas para que nadie quede al descubierto). Pero son eso, defensas, no soluciones. Por eso este problema pertenece, con todas las letras, al capítulo siguiente: es parte de lo que falta.

El veredicto

En limpio, esto es lo que el grafo compartido sí ofrece, y lo que todavía no:

El grafo compartido es un **idioma, no una base central**. La privacidad empieza por la forma de la red: cada parte controla lo suyo y solo lo une bajo consentimiento. El permiso, el consentimiento y la auditoría son **hechos del propio grafo**; se escriben en el mismo modelo que los datos, no en una capa aparte que se desincroniza con el tiempo.

El control fino por eje y el borrado dirigido por *crypto-shredding* son **más viables** aquí que en lo relacional, porque el modelo sabe con exactitud dónde vive cada dato. Pero esa misma facilidad para enlazar, que es su fuerza, convierte la **inferencia por agregación** en el riesgo central, todavía sin solución cerrada para nadie.

La conclusión práctica es incómoda y conviene decirla en voz alta: **una arquitectura de información que no piensa la privacidad desde el primer día no merece adoptarse.** Un modelo que vuelve fácil preguntar cualquier cosa sobre cualquiera sirve tanto para la mejor medicina como para el peor abuso, y la diferencia no está en la potencia del grafo, sino en quién puede preguntar qué.

Que esa pregunta —quién puede preguntar qué— se responda *con el mismo modelo*, y no con un sistema aparte pegado con cinta, es la mejor noticia de este capítulo. Que la inferencia por agregación siga abierta es la peor. Las dos cosas son verdad a la vez, y la página siguiente empieza, justamente, por admitir todo lo que falta.

> *La diferencia entre la mejor medicina y el peor abuso no está en la potencia del grafo, sino en quién puede preguntar qué.*
>
> EL VEREDICTO

30
Qué falta

La propuesta está completa donde más cuesta estarlo: en lo conceptual. Lo que queda no es pensar el modelo, sino hacerlo real. Esta es la hoja de ruta.

Imagina que cierras este libro y decides que la idea merece existir de verdad. Clonas el repositorio, levantas el prototipo, cargas un dominio propio (tu spa, tu clínica, el sistema de tu empresa) y empiezas a registrar hechos. Funciona: los hechos entran como tripletas tipadas, el lexicon traduce tu vocabulario, las consultas responden. Y entonces, casi de inmediato, chocas con una pared. Quieres que el sistema dispare solo la regla *«a la séptima sesión, la octava es gratis»*; el prototipo guarda la regla como dato, pero no la *ejecuta*. Quieres preguntarle no qué era cierto, sino qué sabía el sistema en abril; la respuesta a medias te dice que esa versión del pasado no se conservó. Ninguna de esas paredes es teórica. Son, exactamente, lo que falta.

Este capítulo cambia de registro a propósito. No es una recapitulación ni una mirada inspiradora al horizonte; es un **mapa de implementación**. Hasta aquí el libro presentó la propuesta entera y la sometió a prueba: un prototipo en Python con sus pruebas en verde, ocho dominios industriales modelados que confirman que el catálogo se sostiene en territorios disímiles y (en el capítulo 28) el experimento reflexivo, donde el modelo bastó para describir su propia herramienta y se corrigió a sí mismo bajo carga. Lo que sigue es el inventario de la distancia que separa eso de una pieza de infraestructura que cualquiera pueda adoptar un martes por la tarde.

> **PARA QUIÉN**
> Este capítulo está escrito para tres lectores: quien quiera contribuir al núcleo, quien evalúe adoptarlo temprano en su organización, y quien solo necesite medir cuánto camino queda. Si buscas la visión, salta al cierre; si buscas la lista de tareas, estás en la página correcta.

Lo organizo en seis frentes. De cada uno digo tres cosas: **qué falta**, **qué tan urgente es** y **qué requiere para resolverlo**. Donde el experimento reflexivo ya adelantó terreno lo señalo, porque eso mueve el punto de partida. Y cierro con la pieza que pesa más que todas las demás juntas, y que el autor no controla solo.

Figura 30.1. Los seis frentes que separan el prototipo de la infraestructura, cada uno con la fracción ya resuelta y la pendiente. Ningún frente arranca de cero: en todos hay un piso operable. Lo que falta es altura, no cimiento.

Frente 1 — El motor de inferencia

Apareció una y otra vez a lo largo del libro: el evaluador externo que recorre el grafo y dispara reglas. La regla *«siete sesiones cumplidas, la octava gratis»* del spa. La verificación *«prescripción contra contraindicación»* de la clínica. El cálculo del marcador a partir de cada gol del partido. La cláusula de rescisión que se activa en un contrato. Todas comparten la misma forma estructural: una condición declarativa sobre los hechos del grafo y un consecuente que se efectúa cuando la condición se cumple.

El experimento reflexivo dio el **primer medio paso**: demostró que la conducta puede vivir como dato y ejecutarse desde afuera (un evaluador genérico que despacha verbos leyéndolos del grafo). Lo que ese motor **todavía no es** es un evaluador de **reglas declarativas**. Sabe ejecutar verbos; no sabe resolver «si se cumplen estas condiciones sobre los hechos, deriva este hecho nuevo». Esa es la pieza ausente.

Hay al menos cuatro tecnologías candidatas para esa capa, cada una con su fortaleza y su flanco débil:

SHACL

Shapes Constraint Language (W3C, 2017), pensado para validar grafos RDF[8]. Excelente para chequear consistencia («ningún paciente con dos diagnósticos contradictorios vigentes»); pobre para el razonamiento positivo.

DATALOG

Lenguaje lógico deductivo. Sobresale en razonamiento composicional («si X causa Y e Y causa Z, entonces X causa Z»). Motores como Soufflé alcanzan rendimiento industrial.

CÓDIGO DE APLICACIÓN

Funciones en Python, Rust o Go que reciben el grafo y producen el efecto. Lo más flexible y lo menos auditable. Es, en esencia, la forma que tomó el evaluador de verbos del prototipo.

LLM CON *FUNCTION CALLING*

El propio modelo de lenguaje como evaluador, útil para reglas borrosas o ambiguas. Apropiado para lo pragmático; inapto para el cumplimiento estricto, por las razones que veremos enseguida.

QUÉ FALTA · URGENCIA · QUÉ REQUIERE

Qué falta: la capa de evaluación de reglas declarativas sobre los hechos del grafo.

Urgencia: alta. El prototipo ejecuta comportamiento, pero todavía no *infiere*: almacena la regla del spa, no la dispara. Para la mayoría de las aplicaciones útiles, eso es un bloqueador.

Qué requiere: una *API de motor de inferencia* que el universo exponga — `u.evaluate(regla)`, `u.evaluate_all()` — cuyos resultados sean, otra vez, hechos atómicos firmados por el evaluador. El primer paso es elegir un motor de referencia (SHACL para validación, Datalog para inferencia) y montarlo sobre el evaluador genérico que ya existe. Estimación: tres a seis meses para una primera versión usable.

El lugar de la IA en este frente

La cuarta opción de la lista (usar un modelo de lenguaje) merece una aclaración, porque la tentación es grande. ¿No podría una IA, incluso de pesos abiertos y autoalojada, hacer de motor de inferencia, con un buen *prompt* y un modelo bien definido? Hay que separar dos clases de regla. Para las **borrosas** (¿este reclamo es urgente?, ¿este texto implica consentimiento?) la respuesta es sí, y ahí el LLM es la herramienta correcta. Para las **estrictas** (las del spa, las del banco, las del contrato) la respuesta es no, por cuatro razones que ningún *prompt* arregla.

POR QUÉ UN LLM NO ES EL MOTOR DE LAS REGLAS ESTRICTAS

1. Es **probabilístico**, no determinista: la misma entrada puede dar respuestas distintas.
2. No entrega una **derivación auditable**, y la tesis del modelo es justamente la auditabilidad: hechos firmados, bitemporales, rastreables.
3. Es poco confiable en **conteo y agregación exactos** (contar goles, sumar costos, cerrar una transitividad).
4. Es caro de **escalar** y de **reproducir** una decisión pasada idéntica.

La forma productiva de sumar IA, entonces, no es ponerla de *runtime* sino de **autor**. Que un LLM **compile** una regla escrita en lenguaje natural (*«a las siete sesiones, la octava es gratis»*) a una regla declarativa en Datalog o SHACL sobre los hechos del grafo, y que el **motor determinista la ejecute**. El modelo bien definido vuelve esa compilación casi directa, porque los roles ya están tipados y la regla mapea a patrones sobre hechos canónicos. Así se obtiene lo mejor de los dos mundos: la flexibilidad del lenguaje natural para *escribir* reglas y el rigor del motor para *ejecutarlas*. La IA entra dos veces —para convertir lenguaje en hechos (lo vimos en el capítulo 26) y para compilar reglas—, pero el corazón de la inferencia estricta queda determinista y demostrable.

A FAVOR DE LO ABIERTO

Un modelo autoalojado, con sus pesos fijados, es *más* reproducible que una API cerrada que cambia bajo los pies. Y la reproducibilidad es exactamente lo que una auditoría exige: la misma pregunta debe dar la misma respuesta dentro de cinco años.

En concreto, esa compilación se ve así. El modelo recibe la regla en español y el catálogo tipado, y emite una regla declarativa que el motor ejecutará por su cuenta:

PROMPT

```
SISTEMA. Compila la regla de negocio a Datalog sobre hechos WQuestions
(sujeto, rol, valor). No la ejecutes; solo compílala.

Catálogo: servicio_spa(S) con roles cliente, estatus_factual.
Regla: "A las siete sesiones finalizadas de un cliente, la octava es gratis."
```

DATALOG

```
sesion_ok(C, S)  :- instancia_de(S, servicio_spa),
                    cliente(S, C), estatus_factual(S, finalizada).
```

```
octava_gratis(C) :- count{ S : sesion_ok(C, S) } >= 7.
```

El modelo de lenguaje *escribió* la regla; el motor determinista la *ejecuta* y firma cada hecho derivado. La IA nunca toca el dato en caliente. Y el hecho derivado entra al grafo con la misma forma que cualquier otro: una tripleta tipada, con su autor y su vigencia.

Frente 2 — Bitemporalidad completa

Hoy el modelo soporta **tiempo de validez** (*valid time*): cada hecho lleva su rango `[inicio, fin)`, y una consulta «en el momento T» recupera lo que era cierto del mundo entonces. Esta es la mecánica de vigencia que el libro reifica con rangos (la decisión D6, enunciada en el capítulo 9). Lo que **falta** es el **tiempo de transacción** (*transaction time*): cuándo el sistema afirmó ese hecho.

La diferencia importa cuando alguien pregunta no «¿qué era cierto?» sino «¿qué sabíamos?». Toma un caso del banco. En mayo registramos que el plan mensual de un cliente venció el 31 de marzo; en abril el cliente aparece, demuestra que había renovado y retrocedemos el plan. La pregunta del auditor («¿qué estaba en el sistema entre el 1 de abril y la fecha de la corrección?») solo se responde si el sistema preservó esa versión transitoria, la que era falsa pero estuvo vigente como creencia del sistema durante semanas.

DOS RELOJES

Valid time responde «qué era cierto en el mundo». *Transaction time* responde «qué creía el sistema». Un dominio regulado necesita los dos a la vez: poder reconstruir no solo la historia, sino la historia de lo que se sabía de la historia.

QUÉ FALTA · URGENCIA · QUÉ REQUIERE

Qué falta: cerrar el tiempo de transacción de punta a punta. La estructura del hecho ya reserva un campo para él, pero la persistencia aún no lo conserva en el viaje de ida y vuelta a disco. Queda como deuda concreta.

Urgencia: alta para dominios regulados (finanzas, salud, derecho); baja para el resto. Cuando un dominio lo necesita, no es opcional. El destino es un par de intervalos por hecho y consultas *as-of* duales: `query(valido_en=T1, registrado_en=T2)`.

Qué requiere: una refactorización compatible del módulo del hecho para llevar dos intervalos en lugar de uno, y de la capa de persistencia para preservarlos. Snodgrass[17] formalizó esto en los años noventa; las bases bitemporales modernas (Datomic, XTDB) lo implementan en producción. Estimación: un mes de implementación, más pruebas sobre un dominio regulado real.

Frente 3 — Persistencia industrial

El experimento reflexivo cambió el punto de partida de este frente: el prototipo **ya no vive solo en memoria**. Su capa de persistencia guarda individuos y hechos en **SQLite** y los recarga al arrancar. Bastan dos tablas: una de individuos `(id, eje, payload_json)` y una de hechos `(sujeto, rol, valor, inicio, fin, …)`. Es la prueba de que el modelo persiste sin esfuerzo: la geometría de las tripletas se aplana a tabla casi sin pérdida.

Lo que **falta**, entonces, no es «persistencia» (ya existe) sino **persistencia industrial y plural**: una interfaz `Storage` abstracta de la que SQLite sea la primera implementación, y otros respaldos con perfiles distintos según el caso de uso.

RESPALDO	PERFIL	ESTADO
SQLite	Sistemas chicos, monousuario. El piso para empezar a construir.	✅ ya existe en el prototipo
Postgres + JSONB	Sistemas multiusuario medianos. Índices GIN sobre el *payload*, búsqueda de texto sobre etiquetas, particionado por tiempo de transacción para auditoría.	pendiente · prioritario
Kùzu / Neo4j	Sistemas que privilegian consultas de grafo (rutas, caminos transitivos). Modelo más natural; curva de adopción mayor.	pendiente
RDF / SPARQL	Sistemas que quieran interoperar con la web semántica. Mapeo trivial: cada hecho es una tripleta RDF; la vigencia (D6) se mapea con grafos nombrados.	pendiente

QUÉ FALTA · URGENCIA · QUÉ REQUIERE

Urgencia: media. El piso (SQLite) ya permite construir; lo que falta habilita escala y casos exigentes, no la operación básica.

Qué requiere: extraer la interfaz `Storage` del SQLite actual y escribir al menos un segundo respaldo (Postgres). El módulo del universo delega la lectura y la escritura a esa interfaz, sin saber qué hay debajo. Estimación: dos a tres meses para los dos respaldos primarios.

Frente 4 — Tooling

Una propuesta como WQuestions vive o muere por sus herramientas. Los conceptos pueden ser impecables; si la fricción operativa es alta, nadie la adopta. La regla es vieja y cruel: la mejor idea con mal tooling pierde frente a la idea mediocre con buen tooling. Aquí está la lista, ordenada por prioridad.

4.1 — Ingestor de lexicon. Hoy las entradas del lexicon se escriben a mano. Hace falta una herramienta que ingiera entradas desde recursos masivos ya construidos —FrameNet[14], VerbNet[15], PropBank— y produzca entradas válidas del lexicon WQuestions. Esto da, de un golpe, un piso de cobertura de miles de verbos sin trabajo manual.

4.2 — Parser de lenguaje natural a hechos. Hoy el análisis se hace vía LLM con *function calling*. Hace falta también un parser **local y determinista** para textos donde la latencia del modelo importa (sistemas de tiempo real) o donde la privacidad lo exige (datos médicos que no deben salir de la sala). Combinar análisis sintáctico con el lexicon es trabajo concreto, no investigación abierta.

4.3 — IDE / inspector. El experimento reflexivo entregó una **primera versión**: un inspector que, junto a cada vista, muestra las tripletas que la sostienen («lo que ves es el dato»). Falta el resto: explorar individuos y vecinos a voluntad, consultar con patrones, seguir la cadena `causado_por` o `justificado_por` de cualquier nodo, y (la pieza que el propio experimento señaló como necesaria) **vistas con nombre** definidas como dato, para re-concretar el grafo sin ahogar al modelador en la abstracción. La base existe; el salto es de semanas, no de meses.

4.4 — Validador de migración. Cuando un sistema heredado quiere migrar a WQuestions, hay que verificar que su dialecto de dominio mapea bien al canónico. Una herramienta de validación detecta los cuellos: roles sin signatura, ejes ambiguos, vigencias inconsistentes. El paso «el dato registra su propia signatura», que la prueba reflexiva ya introdujo, facilita esta herramienta.

4.5 — Generador de servidor MCP. Dado un lexicon, generar automáticamente el servidor MCP correspondiente. Esto convierte cada dominio en un asistente conversacional sin escribir *glue code*. Trabajo de pocas semanas; impacto desproporcionado en la adopción, porque baja a casi cero el costo de probar la propuesta con datos propios.

URGENCIA DEL TOOLING

Media en agregado, alta en el caso del **ingestor de lexicon** (4.1) y el **generador MCP** (4.5): son los dos que más reducen el costo de que alguien pruebe la propuesta. Y la barrera de entrada es, casi siempre, lo que decide si una idea se difunde o se queda en el libro que la enunció.

Frente 5 — Lexicon poblado en varios idiomas

El catálogo canónico del libro (la decisión D8, que hace invisible el catálogo y deja al lexicon como única interfaz) define 38 roles. El lexicon del prototipo registra una decena de verbos. Para un sistema productivo, el lexicon de *un solo idioma* necesita del orden de **dos a cinco mil entradas**: los verbos frecuentes del español, sus formas nominales, las locuciones idiomáticas que la gente usa sin pensar.

Lo que **falta** es trabajo lingüístico paciente. Por dominio o por familia semántica (transferencia, comunicación, movimiento, percepción, estados) hay que poblar el lexicon. FrameNet español, el corpus español de Universal Dependencies y AnCora son fuentes legítimas para mecanizar parcialmente la tarea, pero no para eliminar el juicio humano que cada entrada delicada exige.

CATÁLOGO NEUTRAL

El catálogo D8 es idiomáticamente neutral: nombres internos como `agente`, `paciente` o `tema` son etiquetas que podrían ser griegas o números. Lo que cambia entre idiomas es la **capa de alias** y los **patrones de polisemia** propios de cada lengua.

Para la internacionalización, cada idioma necesita su propio lexicon, pero no su propio catálogo: la espina es común y solo cambia la piel que la traduce. Esto es, a la vez, una oportunidad y una carga. Oportunidad, porque un dominio modelado en español queda inmediatamente disponible en cualquier idioma que tenga su lexicon. Carga, porque cada idioma nuevo es un proyecto léxico sostenido, no un fin de semana.

QUÉ FALTA · URGENCIA · QUÉ REQUIERE

Urgencia: la cobertura del lexicon *es* la usabilidad. Sin verbos suficientes, el modelo se siente incompleto aunque sea correcto. Es esfuerzo sostenido, no proyecto puntual: el lexicon nunca está «terminado», solo más o menos completo.

Qué requiere: equipo lingüístico-computacional; idealmente, colaboración con universidades que ya producen recursos léxicos; y un modelo de gobernanza para aceptar contribuciones de terceros sin perder coherencia. Lo cual nos lleva, naturalmente, al sexto frente.

Frente 6 — Comunidad y gobernanza

Aquí entramos en la pieza que el autor no controla. WQuestions, para volverse útil más allá de un libro, necesita **comunidad**: gente modelando dominios, contribuyendo lexicon, reportando fricciones, escribiendo herramientas, adoptándolo en proyectos reales. Una arquitectura sin comunidad es una idea bonita en un PDF; una arquitectura con comunidad es infraestructura. La diferencia no la firma el autor, la firman los demás.

Qué falta, en concreto:

Repositorio canónico abierto, con licencia permisiva. Una primera versión ya vive en un repositorio público mientras lees esto.

Proceso de contribución: cómo proponer nuevas entradas al catálogo, nuevos dominios al lexicon, parches al motor. Necesita criterios escritos, no costumbres tácitas.

Foro o canal de discusión para resolver las fricciones que surjan al modelar dominios nuevos. Cada conversación cualifica el catálogo (y el experimento reflexivo es evidencia de que las fricciones más valiosas aparecen cuando se somete el modelo a una carga real).

Estandarización gradual: una vez que varios proyectos adopten el modelo, vale la pena llevar las partes más universales del catálogo a un proceso formal (IETF, W3C, ISO). Eso da estabilidad legal para el uso empresarial.

Dialectos de dominio mantenidos por comunidades sectoriales (clínico, financiero, legal, manufactura), cada uno con su propia gobernanza dentro de la espina común.

> *Las arquitecturas duraderas no se imponen: se adoptan. Y se adoptan cuando alguien que no es su autor decide que vale la pena empujarlas.*

LA PIEZA QUE EL AUTOR NO CONTROLA

QUÉ FALTA · URGENCIA · QUÉ REQUIERE

Urgencia: el reloj corre. Si la comunidad no se forma en el momento en que los LLMs con MCP se popularizan, alguna otra propuesta (menos cuidadosa) ocupará el espacio. La ventana es de dos a cinco años.

Qué requiere: lo mismo que cualquier proyecto de código abierto serio: un autor o equipo fundador dispuesto a moderar, a criticar contribuciones, a sostener la coherencia y a decir que no cuando hace falta. Buena documentación. Casos de uso ejemplares. Adoptantes tempranos visibles.

La pila completa

Conviene ver los seis frentes juntos, como capas de una misma pila. Cada capa puede evolucionar por su cuenta (cambiar el respaldo de persistencia no obliga a tocar el lexicon, y mejorar el motor de reglas no altera la aplicación que ve el usuario). Esa independencia es, en sí misma, una propiedad de diseño: el modelo del medio es el contrato estable que sostiene todo lo demás.

Figura 30.2. La pila completa. Abajo, la persistencia (hoy SQLite, mañana plural tras una interfaz `Storage`); encima, el núcleo de los siete ejes (el contrato estable); encima, el motor de reglas y el lexicon; encima, el LLM que traduce y compila vía MCP; arriba, la aplicación que ve la persona. La bitemporalidad atraviesa el núcleo y la persistencia, porque el tiempo no es una capa: es una dimensión de cada hecho.

Y si los seis frentes se miden por su grado de madurez, el panorama es claro y no desalentador. Tres de ellos tienen un piso operable (persistencia, una primera versión de inspector, el medio paso del motor); tres están casi por construir (el lexicon a escala, la comunidad, el tooling completo). Nada está en cero; nada está cerca de cien.

Figura 30.3. Madurez estimada por componente, en una escala de cero a cien. La persistencia y la bitemporalidad parten con ventaja porque el prototipo ya las tocó; el lexicon a escala y la comunidad son los frentes más jóvenes. Pasa el cursor sobre las barras para ver las cifras. Es una estimación de autor, no una medición: sirve para ver la *forma* del trabajo pendiente, no para presumir precisión.

Las fricciones documentadas que siguen abiertas

Además de los seis frentes mayores, el prototipo expuso un puñado de fricciones puntuales del catálogo al chocar con los dominios de estrés. El experimento reflexivo (capítulo 28) **cerró varias** (el texto libre, el tipado de los campos definidos por datos, el display derivado de hechos) y entregó el comodín `V` (*cualquier eje de valor*), que resuelve de raíz toda la familia «esta signatura es demasiado estrecha». Lo que queda abierto es esto:

FRICCIÓN	ORIGEN	PARCHE PROPUESTO
`paciente` / `partes: 0→Q` demasiado estrechos	química, fútbol	Relajar a `0→V`: el comodín `V` ya existe; es una línea
`tema: 0→0` rechaza K (obra, medicamento)	música, clínica	`tema: 0→V`, o un `tema_categorico: 0→K`
Patrones temporales finos; tiempo musical (compás, pulso)	clínica, música	Reificar como O con estructura
Reglas de derivación versionadas	contrato	Frente 1 (motor) + vigencia (D6) sobre las reglas
Vistas y proyecciones con nombre como dato	prueba reflexiva	El siguiente escalón de la re-concreción (Frente 4.3)

Ninguna de estas fricciones bloquea el funcionamiento del modelo, y se repite un patrón: las relajaciones del tipo «signatura demasiado estrecha» se resuelven con el comodín `V` que el experimento ya introdujo, y varias otras quedan cubiertas por roles de dominio bajo la política liberal del catálogo. Es **la lista de mejoras concretas**: se acumula a medida que el modelo se prueba en territorios nuevos y se acorta a medida que se lo somete a cargas exigentes. Una lista que crece y mengua a la vez es exactamente lo que cabe esperar de una arquitectura viva.

El libro como semilla

Cerremos con la idea menos técnica del capítulo. Un libro no es una propuesta terminada: es **una invitación a que alguien la termine**. Las arquitecturas que duraron (Unix, TCP/IP, HTTP, SQL, RDF) empezaron como artículos, manifiestos, RFCs, libros: textos que articulaban una idea con la claridad suficiente para que otros pudieran apropiársela y empujarla adelante. Ninguna nació completa; todas nacieron *legibles*.

LO QUE TIENES EN LA MANO AL CERRAR EL LIBRO

No es un producto. Es una **base operable**: suficiente para entender la propuesta, ejecutarla, criticarla y extenderla. El catálogo está bien diseñado pero incompleto; el lexicon está bosquejado; las herramientas son embrionarias; la comunidad está por construirse. Y aun así, la propuesta no solo funciona en ocho dominios distintos: **funciona aplicada a sí misma**.

Eso es lo que hace la diferencia entre una especulación y una semilla. El modelo bastó para describir su propio menú, sus formularios, sus esquemas y su conducta, y para corregirse cuando la carga reveló una signatura demasiado estrecha. La propuesta aguanta el peso que prometió aguantar. Desde aquí, la tarea no es demostrarla (ya está demostrada en lo esencial), sino perfeccionarla: poblar el lexicon, montar el motor, escribir las herramientas, formar la comunidad. Trabajo, no misterio.

> *El catálogo está incompleto, el lexicon bosquejado, las herramientas embrionarias. Pero la propuesta funciona aplicada a sí misma. Lo que falta es construirla, no descubrirla.*

EL ESTADO DEL PROYECTO

Queda una sola página antes del cierre, y es la que responde la pregunta que ha estado latiendo bajo todo el libro: si esto es tanto trabajo, ¿por qué vale la pena? Por qué importan, a fin de cuentas, las preguntas.

31

Por qué importan las preguntas

La misma sala, otra noche, dos años después. La paciente es la misma; el dolor es otro. Lo que cambió no fue la tecnología: fue que los sistemas, por fin, aprendieron a hablar en preguntas.

Son otra vez las dos de la madrugada, y otra vez entra **Vega** en urgencias. Han pasado dos años desde la noche con la que abrimos este libro; ahora tiene cuarenta y cuatro. Aquel dolor en el pecho resultó ser una crisis de ansiedad; esta vez es una cosa distinta: una arritmia que su endocrinóloga le venía vigilando desde hace meses y sobre la que el cardiólogo de la otra ciudad ya le había advertido. Lo cuenta entre frases entrecortadas, mientras la enfermera le toma la presión y el oxígeno. El médico de guardia es nuevo (no es el de aquella noche, no la conoce de nada) y tiene los mismos minutos, no horas, para decidir bien.

> **CIERRE DEL HILO**
> Esta escena es la otra cara de la que abrió el capítulo 1. Allí, la consulta que salvaba una vida era imposible. Aquí, es una sola línea.

Teclea el nombre en la tablet. Y esta vez sí aparece —entera— la historia. No la trae un convenio firmado entre los dos hospitales, ni una conexión bilateral negociada durante meses, ni el esfuerzo heroico de algún integrador de turno a las dos de la madrugada. Aparece porque ambos hospitales, junto con la endocrinóloga privada y el cardiólogo de la otra ciudad, **hablan en preguntas**. Donde hace dos años uno escribía `dx_p` y el otro `diagnostico_principal` (y por esa sola diferencia de vocabulario la información existía pero no podía consultarse), hoy los dos reconocen que están afirmando lo mismo: una situación de tipo `diagnostico_medico`, con un *paciente*, un *agente*, un *momento* y un *diagnosticado_como*. El vocabulario interno de cada sistema sigue siendo el suyo. Lo único que comparten es la pregunta que cada hecho responde.

El médico ve la cronología completa, en orden: el primer eco que detectó la endocrinóloga, los controles trimestrales, el ajuste de medicación de noviembre, la consulta de marzo con el cardiólogo, la recomendación específica para los episodios de descompensación. Toma una decisión informada en treinta segundos, sin tener que despertar a un especialista. Vega queda estable. Y el equipo registra la situación nueva (`consulta_emergencia_2028_4471`) que mañana otro médico, en otro hospital, podrá consultar exactamente igual.

> *Entre las dos noches no hubo un milagro tecnológico: hubo el reconocimiento de algo viejo y la disciplina de construir, encima, las piezas mínimas para volverlo operativo.*

EL ARCO DEL LIBRO, EN UNA FRASE

Esta escena (banal, casi aburrida cuando funciona) es lo que el libro entero quiso explicar cómo conseguir. Las dos noches son la misma sala, la misma paciente, el mismo tipo de pregunta clínica. Lo que cambió cabe en una figura.

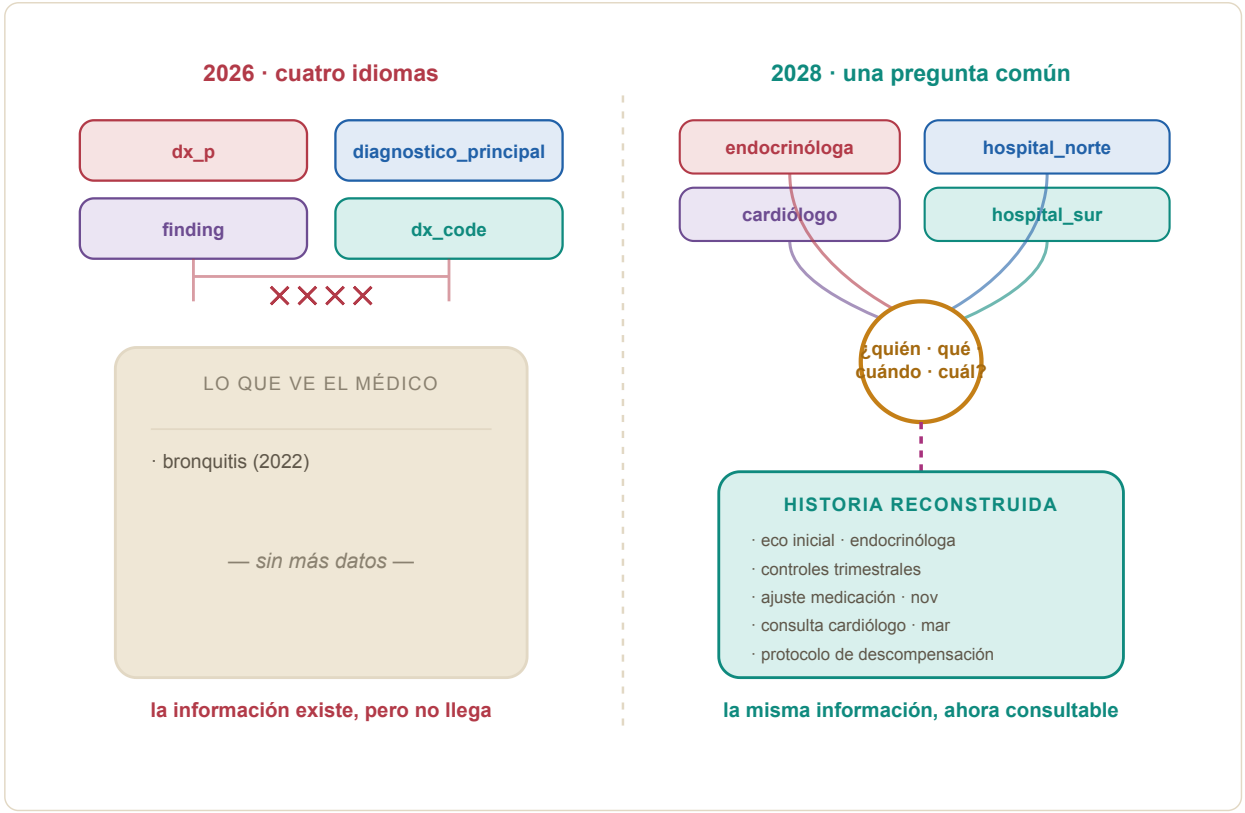


Figura 31.1. Las dos noches de Vega. En **2026** cuatro sistemas guardan el mismo hecho clínico con cuatro claves incompatibles (`dx_p`, `diagnostico_principal`, `finding`, `dx_code`): la información existe, pero choca contra tabiques y el médico solo ve una línea. En **2028** las mismas fuentes publican al grafo común declarando *qué pregunta responde cada hecho*; la historia se reconstruye entera. No cambió el dato: cambió que ahora se puede consultar.

Lo que las preguntas resolvieron

Visto en retrospectiva, lo que las seis partes de este libro destilaron es una respuesta a una pregunta vieja, la misma que dejamos planteada en el capítulo 1: *¿existe algo más simple que una ontología y más estructurado que un texto libre, que pueda servir de común denominador entre sistemas que no se conocen?*

LA RESPUESTA DEL LIBRO

Sí: hay **siete preguntas**. Quién, qué, dónde, cuándo, cuánto, cuál y cómo (donde *cuál* es, justamente, la que organiza las categorías). Son lo bastante universales para que cualquier descripción del mundo se mapee a ellas, y lo bastante simples para que un humano (o un modelo de lenguaje) las entienda sin manual.

Esas siete preguntas no son un invento de este libro ni de su autor. Son anteriores a cualquier ontología: están en las categorías de Aristóteles[1], codificadas por Cicerón[2] en su hexámetro retórico, redescubiertas por los manuales de periodismo de 1900, formalizadas por la semántica de eventos que la lingüística viene refinando hace sesenta años, y presentes en la cognición infantil mucho antes que la escritura. El capítulo 6 rastreó esa convergencia: cuando tantas tradiciones independientes, separadas por siglos y sin coordinarse, llegan a la misma lista, esa lista deja de ser una hipótesis y empieza a parecer un descubrimiento.

Pero las preguntas, por sí solas, son apenas una intuición. Para volverlas **operativas** (ejecutables por una máquina, consultables con rigor) hicieron falta unas pocas decisiones de diseño, las que recorre la Parte III. Conviene recapitularlas, porque son lo que separa una buena metáfora de una arquitectura:

REIFICAR CUANDO IMPORTA D4

Un evento o una relación se convierte en una *situación* de pleno derecho (con identidad propia, consultable) solo cuando lo amerita. La `consulta_emergencia_2028_4471` de esta noche es una de esas situaciones: existe como objeto porque mañana habrá que preguntarle cosas.

AGENCIA CONTEXTUAL D5

El rol de *agente* no es exclusivo de los humanos. Lo puede ocupar una persona, una organización, un programa o un sensor, según el verbo. La endocrinóloga, el hospital y el monitor cardíaco son, cada uno en su hecho, *quién*.

VIGENCIA TEMPORAL D6

Las propiedades que cambian se reifican con un rango `inicio` / `fin`: el sistema nunca olvida. Por eso el médico no ve solo la medicación de hoy, sino el ajuste de noviembre y lo que había antes. La cronología es consultable porque el tiempo es un dato, no un descarte.

CUATRO PORQUÉS D7

No hay un eje «por qué»: el porqué se reparte en cuatro cables (`causado_por`, `motivado_por`, `con_finalidad`, `justificado_por`), porque el lenguaje natural ya los distingue. La arritmia fue *causada* por una condición; el ajuste de dosis fue *motivado* por un control.

Y por encima de las preguntas y de las decisiones, una sola capa más: el **lexicon**. Es lo que traduce el vocabulario del usuario al catálogo canónico sin obligar a nadie a aprender etiquetas internas (el médico escribe en su jerga; el sistema la mapea). Su forma final encierra la coincidencia más afortunada de todo el proyecto.

EL LEXICON ES, TÉCNICAMENTE, UN *FUNCTION SCHEMA*

La interfaz que un modelo de lenguaje necesita para invocar una herramienta (*function calling*: un nombre, unos parámetros tipados, una descripción) es, hasta en su estructura, lo mismo que una entrada del lexicon. No lo buscamos al empezar el proyecto; lo descubrimos al final. Esa coincidencia es la que vuelve la propuesta accionable hoy, en 2026, y no en una pizarra teórica. El capítulo 26 la desarrolla.

Por qué las preguntas son anteriores a las ontologías

Aquí el libro toma su única licencia filosófica, y conviene dejarla explícita. Las ontologías de dominio (CIDOC CRM[4], Schema.org[30], FHIR[6], Biolink[5], las decenas de iniciativas que repasamos en el capítulo 3) son catálogos de **qué cosas hay**. Cada una hizo el trabajo paciente de definir entidades, relaciones y vocabularios consistentes para su parcela. Cada una es valiosa dentro de su perímetro. No competimos con ninguna.

Lo que el libro propone es que, **antes que las entidades, están las preguntas**. Antes de *paciente, médico, diagnóstico, prescripción*, está la pregunta *¿quién hizo qué a quién, cuándo y de qué tipo?*, y sus respuestas tipadas. Un dominio nuevo, al modelarse, no se reduce a otro dominio existente: se mapea a las preguntas, que son siempre las mismas. La ontología específica del dominio sobrevive como **dialecto** del lexicon; el catálogo común sobrevive como infraestructura. Cada ontología sigue siendo valiosa donde ya está; lo que cambia es que, por fin, pueden hablar entre ellas sin un proyecto de integración de por medio.

Visto así, la torre de Babel del primer capítulo nunca fue un problema de ontologías *de más*. Fue un problema de ontologías *sin un suelo común*. La solución no era una ontología más grande que las tragara a todas (el sueño de la ontología universal fracasó tantas veces que ya es un género literario), sino una capa *por debajo*: más simple, más vieja, más estable.

Esa capa son las preguntas. Mientras las ontologías son lo variable (cambian con cada dominio, con cada empresa, con cada década), las preguntas son lo invariante. Lo único que este libro hizo fue proponer una **articulación operativa** de esa intuición antigua: un catálogo concreto, un prototipo ejecutable y ocho dominios modelados como prueba de que el catálogo se sostiene fuera de un solo nicho.

Esa es, al final, la diferencia entre las dos noches de Vega expresada en una sola tripleta. En 2026, el hecho clínico vivía encerrado en el dialecto de un sistema. En 2028, el mismo hecho se publica declarando a qué pregunta responde cada parte:

TRIPLETAS · LA HISTORIA DE VEGA, HOY

```
(consulta_emergencia_2028_4471, paciente,         vega)              ∈ M(O, Q)
(consulta_emergencia_2028_4471, agente,           guardia_nocturno)  ∈ M(O, Q)
(consulta_emergencia_2028_4471, instante,         2028-03-04T02:11)  ∈ M(O, T)
(consulta_emergencia_2028_4471, diagnosticado_como, arritmia)        ∈ M(O, K)
(consulta_emergencia_2028_4471, causado_por,      condicion_cardiaca) ∈ M(O, O)
```

Cualquiera de los cuatro sistemas (y cualquier modelo de lenguaje) lee ese registro de izquierda a derecha, como una frase. Ninguno necesitó conocer a los otros tres. Solo necesitaron coincidir en las preguntas.

Lo que el libro no terminó

Conviene cerrar nombrando lo que falta. El capítulo 30 lo enumeró en detalle:

LO QUE AÚN NO ESTÁ

Un motor de inferencia maduro. Bitemporalidad completa. Persistencia a escala industrial. Herramientas de autoría y depuración. Un lexicon poblado en varios idiomas. Y, sobre todo, una comunidad. La propuesta es completa **conceptualmente**; no está lista para producción. El prototipo que acompaña al libro ejecuta lo que el libro afirma (los ocho dominios pasan sus validaciones, los tests pasan, los ejemplos corren), pero está lejos, todavía, de ser infraestructura.

Hay una verdad más, incómoda, que conviene decir en voz alta: **ninguna arquitectura sobrevive por su elegancia**. Las que sobrevivieron (Unix, TCP/IP, HTTP, SQL) no ganaron por ser las más bellas, sino porque hubo gente que las cuidó durante décadas hasta volverlas invisibles. RDF, que cumple veinticinco años en 2026, sigue siendo un activo subutilizado precisamente porque nunca alcanzó esa masa crítica. WQuestions, hoy, está en el día uno de ese proceso. Tiene valor (los ocho dominios que pasamos por el prototipo lo demuestran), pero la tarea de aquí en adelante es que *otros* lo perfeccionen.

LA IDEA DE FONDO

El libro como semilla, no como punto final. Lo que importa no es que la propuesta esté terminada, sino que esté escrita, sea operable y se pueda criticar.

Por eso vale la pena que exista escrita. La siguiente generación de modelos de información tiene aquí una base operable de la que partir: el repositorio en línea, el prototipo ejecutable, los ocho dominios modelados, las decisiones de diseño documentadas. No es mucho (apenas lo suficiente para entender la propuesta y atacarla con argumentos). Pero esa es, exactamente, la idea.

Una última nota sobre el momento

Queda una coincidencia histórica que el libro mencionó solo de pasada y que merece decirse en limpio. Esta propuesta (un modelo de información organizado sobre preguntas-coordenadas, traducido por un lexicon que es a la vez catálogo de funciones) habría sido casi imposible de adoptar hace cinco años. Faltaba la pieza que la vuelve usable: una capa capaz de traducir con fluidez entre el modelo estructurado y el lenguaje natural de las personas. Hoy esa pieza existe, y son los grandes modelos de lenguaje.

DOS PIEZAS COMPLEMENTARIAS

El **grafo** aporta lo que al LLM le falta: identidad estable, memoria persistente, hechos que no se alucinan. El **LLM** aporta lo que al grafo le falta: fluidez conversacional, traducción a cualquier idioma, tolerancia a la ambigüedad humana. Lo que una no hace, la otra sí. Y la pieza que las une (el lexicon como *function schema*) resulta ser, exactamente, lo que se necesita a ambos lados de la frontera.

Eso es lo que justifica este libro *ahora*, en 2026, y no antes ni después. Antes no había con qué construirlo. Después puede haber demasiadas propuestas competidoras, cada una con un costo de adopción que nadie querrá pagar dos veces. La ventana entre que los LLMs maduraron y que el espacio se consolide en torno a un estándar dominante es estrecha. Si en esa ventana las preguntas-coordenadas aparecen como una opción razonable (discutible, mejorable, pero razonable), el esfuerzo del libro habrá valido la pena.

Cierre

Empezamos con una sala de emergencias donde la información existía pero no podía consultarse. La cerramos con otra noche en la misma sala, dos años después, donde la información sigue existiendo y ahora *sí* se consulta. Entre las dos noches no hubo magia: hubo el reconocimiento de algo muy viejo (que las preguntas que hace dos mil años se le hacían a un acto moral son las mismas que hoy se le hacen a una historia clínica) y la disciplina de construir, encima de ese reconocimiento, las piezas mínimas para volverlo operativo.

Vega salió estable de urgencias. No porque la medicina avanzara en dos años (avanzó, pero no es de eso de lo que trata este libro), sino porque la información que la describía aprendió a responder, en voz alta, a qué pregunta pertenecía. Quién, qué, dónde, cuándo, cuánto, cuál, cómo. Las mismas siete de siempre. Las que un niño domina antes de saber qué es un sustantivo. Las que no cambian.

Q quién O qué L dónde T cuándo N cuánto K cuál M cómo

> *Las preguntas seguirán siendo las mismas. Lo que falta es construir entre todos las respuestas.*
>
> WQUESTIONS

Quizá esa sea la promesa más discreta del libro: modelar a uno con cuidado fue, sin proponérnoslo, la forma de poder preguntar por todos.

El resto, lector, lo escribes tú.

ANEXOS

32

Anexo: el código

Una hoja de ruta técnica para quien va a implementar. Aquí están, en un solo lugar, las estructuras de datos, el catálogo de ejes y predicados, las cuatro APIs del prototipo y los formatos en que viaja un hecho. Menos relato, más referencia.

A lo largo del libro el código apareció a cuentagotas: una llamada aquí para ilustrar la reificación de una situación, una consulta allá para mostrar la bitemporalidad. Ese goteo es deliberado en el cuerpo del texto (cada fragmento llega cuando la idea que lo justifica está fresca), pero estorba a quien se sienta a programar. Este anexo invierte la prioridad: consolida en un único sitio las piezas técnicas, ordenadas no por el hilo narrativo sino por su función. Es un manual de referencia, no un capítulo para leer de corrido.

La buena noticia, que este anexo deja a la vista mejor que ninguna otra página, es que hay poco que memorizar. El núcleo del modelo cabe en un puñado de estructuras de datos y cuatro funciones. Todo lo demás (los ocho dominios industriales, los cuatro escenarios de estrés, la aplicación que se describe a sí misma) se construye combinando esas pocas piezas. Si solo vas a leer una sección, que sea el catálogo de predicados: ahí está el vocabulario con el que se escribe cualquier hecho.

> **SOBRE LOS IDENTIFICADORES DE ESTE ANEXO**
>
> Los ejemplos usan el repertorio del libro (la venta de Marco, el gol de Messi, la película de Serra, la ordenanza de micromovilidad, la sesión de IA, el episodio del paciente Vega) para que reconozcas los casos. Las *estructuras* (los campos de un hecho, la forma de un patrón, las firmas de las APIs) son las del prototipo real, que vive en `prototipo/wq/`. Cambia los nombres a tu dominio; la maquinaria no cambia.

Las cuatro APIs del núcleo

Todo el modelado de dominios pasa por cuatro funciones. Dos escriben en el grafo, dos lo interrogan. No hay una quinta función «importante» escondida: con estas se cubren los ocho dominios de la Parte V de punta a punta.

PYTHON

```
# ── ESCRITURA ──────────────────────────────────────────────────────

ingest_situation(u, lex, verbo, *, roles={}, complements=[], extra={}, sit_id=None)
#   Reifica una situación en el universo `u`. Resuelve `verbo` contra el
#   lexicon `lex`, valida los roles obligatorios de su signatura y asienta
#   los hechos atómicos. `complements` desambigua la polisemia; `extra`
#   admite metadatos como `estatus_factual`. Devuelve el Individual creado.

u.assert_fact(sujeto, rol, valor, *, valid_from=None, valid_to=None)
#   Asienta un hecho atómico directo (una tripleta). `valid_from`/`valid_to`
#   activan la bitemporalidad (D6). Nunca sobreescribe: cada llamada agrega.
```

```
# ── LECTURA ──────────────────────────────────────────────────────────

query(u, Pattern(fixed={}, ask={}, type_constraint=None), *, at=None)
#   Resuelve un patrón contra el grafo. Devuelve una lista de bindings (un
#   dict por coincidencia). `at` proyecta la consulta a un instante de
#   tiempo de validez. Las variables a resolver se declaran con `Var()`.

count(u, Pattern(...), *, at=None)
#   Igual que `query`, pero devuelve solo la cardinalidad (un entero).
```

> **NOTA**
> El primer argumento de las cuatro APIs es siempre el universo `u`: el contenedor del grafo. En un sistema con varios espacios (por inquilino, por entorno) cada `u` es uno de ellos. El capítulo 29 usa esa frontera para el control de acceso.

Junto a `count` conviven agregadores del mismo molde (`suma`, `promedio`) que aplican una función sobre el rol numérico de las coincidencias. Comparten la firma exacta de `count`, cambiando solo qué devuelven; por eso no merecen entrada propia.

Estructuras de datos

El grafo se apoya en cuatro tipos. Conviene leerlos antes que cualquier código de dominio: todo lo demás los manipula.

El eje (`Axis`) y el individuo (`Individual`)

Cada nodo del grafo es un `Individual` anclado a uno de los seis ejes de valor. El eje no es decorativo: determina qué pregunta responde el nodo y, por tanto, qué predicados puede recibir. El séptimo eje, **M** (cómo), no etiqueta nodos: es el de los cables que los unen.

PYTHON

```
class Axis(Enum):
    Q = "quien"      # agente
    O = "que"        # objeto, evento, situación
    L = "donde"      # lugar
    T = "cuando"     # tiempo
    N = "cuanto"     # número, magnitud
    K = "cual"       # clase, categoría
    # M (cómo) no es eje de nodos: es el eje de los predicados (cables).

@dataclass
class Individual:
    id: str                # identificador estable y único
    axis: Axis             # a qué eje pertenece
    label: str = ""        # rótulo legible (opcional)
```

> **RECUERDA**
> En **K** viven los conceptos atemporales (las plantillas, como `venta` o `largometraje`); en **O** y los pilares, las entidades situadas, como `venta_001` (D1). El predicado `instancia_de` es el puente entre ambos mundos.

El hecho (`Fact`)

La unidad atómica del modelo es la tripleta tipada `(sujeto, rol, valor)` (D3). En el prototipo es un `Fact`. Los campos `valid_from` y `valid_to` implementan la vigencia (D6): un hecho no se borra ni se edita: se cierra con un `valid_to` y se asienta otro que lo reemplaza.

```python
@dataclass
class Fact:
    subject: Individual          # el nodo del que se predica
    role:    str                 # el cable (predicado del eje M)
    value:   Union[Individual, str, int, float]   # nodo o literal
    valid_from: datetime = None   # inicio de vigencia (D6)
    valid_to:   datetime = None   # fin de vigencia; None = abierto al futuro
```

Un literal (un número, una cadena) puede ocupar la casilla del valor cuando el destino es una magnitud o un texto; cuando el valor es otra entidad del grafo, es un `Individual`. Esa es la única bifurcación de tipo en toda la estructura.

El patrón de consulta (`Pattern` y `Var`)

Una consulta es un patrón con tres ranuras. `fixed` ata roles a valores concretos (lo que ya sabes); `ask` declara con `Var()` las variables a resolver (lo que preguntas); `type_constraint` restringe el tipo del sujeto vía `instancia_de`.

```python
class Var:
    """Marca una ranura a resolver dentro de un patrón."""

@dataclass
class Pattern:
    fixed:  dict = field(default_factory=dict)   # roles atados a valores
    ask:    dict = field(default_factory=dict)   # roles con Var() a resolver
    type_constraint: Individual = None           # filtra por instancia_de
```

La entrada de lexicon (`LexiconEntry`)

El lexicon es el compilador que traduce un verbo del lenguaje a la signatura de una situación (D8). Cada `LexiconEntry` declara qué tipo de situación produce el verbo, qué roles son obligatorios y (si el verbo es polisémico) qué complemento la dispara.

```python
@dataclass
class LexiconEntry:
    verb:           str            # el verbo de superficie ("servir", "anotar"…)
    situation_type: str            # la clase de situación que produce (→ K)
    obligatory:     list           # roles que la signatura exige
    pattern:        tuple = ()     # complemento que dispara esta acepción
    notes:          str = ""       # acepción genérica / aclaración
```

Catálogo de los ejes

Las seis preguntas de valor más el eje de los predicados. Cada hecho fija una posición en este espacio: una lectura por eje, cosida por los cables de **M**.

Figura 32.1. Los siete ejes. Seis fijan el valor de un hecho (una lectura por eje, a la derecha el ejemplo de `venta_001`); el séptimo, **M**, no aloja nodos: es el de los predicados que cosen las seis lecturas en un hecho consultable.

Q quién · agente
O qué · objeto / evento
L dónde · lugar
T cuándo · tiempo
N cuánto · número
K cuál · clase
M cómo · predicado

Catálogo de predicados canónicos

Los predicados son los cables del eje **M**. Propiedades y relaciones se unifican: ambas son cables; solo difieren en cardinalidad —funcional (un solo valor) o múltiple (D2)—. La firma de cada predicado declara de qué eje sale y a cuál llega; la escribimos `M(O→Q)`: «desde un nodo O hacia un nodo Q». El motor solo valida este repertorio canónico; cualquier rol que un dominio invente (por la política liberal del catálogo) se acepta sin declararlo.

Conviene mostrar primero la forma del cable con el componente insignia. El hecho «Messi es el autor de `gol_001`» es una tripleta cuyo cable es `agente`:

Predicados de rol (signatura de las situaciones)

Estos cables conectan una situación (O) con sus participantes. Son los roles que un `LexiconEntry` declara obligatorios. El rol `agente` es contextual: lo puede ocupar un humano, una organización, un software o un sensor, según el verbo (D5).

Predicados de rol más usados. La firma indica eje de origen → eje de destino.

PREDICADO	FIRMA	CONECTA	EJEMPLO
`agente`	M(O→Q)	situación → quien actúa	`gol_001 · agente · messi`
`paciente`	M(O→Q)	situación → quien la recibe	`diagnostico_771 · paciente · vega`
`beneficiario`	M(O→Q)	situación → a favor de quién	`asig_001 · beneficiario · luis`
`tema`	M(O→O)	situación → su objeto	`canc_001 · tema · viaje_088`
`instrumento`	M(O→O)	situación → con qué	`asig_001 · instrumento · vehiculo_12`
`lugar_de`	M(O→L)	situación → dónde	`venta_001 · lugar_de · tienda_centro`
`momento`	M(O→T)	situación → cuándo (puntual)	`venta_001 · momento · t_16_32`
`inicio` / `fin`	M(O→T)	situación → ventana temporal	`sesion_ia_5521 · inicio · t0`

Predicados estructurales

Cables que arman la jerarquía y la categorización del grafo. `instancia_de` es el más importante de todo el modelo: ancla una entidad situada (O) a su plantilla atemporal (K), y es el que lee `type_constraint` en una consulta.

Predicados estructurales: categorización y composición.

PREDICADO	FIRMA	SIGNIFICA	EJEMPLO
`instancia_de`	M(O→K)	esta entidad es un caso de esta clase	`venta_001 · instancia_de · venta`
`subclase_de`	M(K→K)	esta clase especializa a otra	`venta · subclase_de · transaccion`
`parte_de`	M(O→O)	esta entidad compone a otra mayor	`escena_42 · parte_de · pelicula_marea`
`estatus_factual`	M(O→K)	estado de la situación	`viaje_088 · estatus_factual · cancelado`

POR QUÉ IMPORTA

`estatus_factual` es la clave de la inmutabilidad: en vez de borrar un viaje cancelado, se asienta un hecho que lo marca `cancelado`. El dato viejo sobrevive; la historia queda completa.

Predicados del «por qué» (D7)

No hay un eje «por qué». El porqué se reparte en cuatro cables distintos, y la distinción no es cosmética: confundir la causa con la finalidad es un error de modelado. Las cuatro pueden coexistir sobre la misma situación sin pisarse.

Los cuatro predicados en que se descompone el «por qué» (D7).

PREDICADO	RESPONDE	APUNTA A	EJEMPLO
`causado_por`	¿qué lo produjo?	causa eficiente	tarifa elevada → estado de alta demanda
`motivado_por`	¿por qué razón?	motivo del agente	prescripción → diagnóstico
`con_finalidad`	¿para qué?	fin perseguido	prescripción → objetivo terapéutico
`justificado_por`	¿con qué fundamento?	norma o autoridad	ordenanza → competencia municipal

Predicados de trazabilidad

Como el modelo nunca borra, necesita cables que registren la enmienda: cuál hecho reemplaza a cuál. Son la firma de la inmutabilidad (D6) en el grafo.

Predicados que registran la enmienda sin destruir el dato previo.

PREDICADO	SIGNIFICA	EJEMPLO
`rectifica`	corrige un hecho anterior conservándolo	rediagnóstico → diagnóstico previo
`cancela`	anula una situación previa	cancelación → viaje original
`verificado_contra`	se validó frente a una regla	prescripción → regla de contraindicación

LA POLÍTICA LIBERAL DEL CATÁLOGO

Este catálogo es *canónico*, no *cerrado*. El motor valida estrictamente los predicados de arriba, pero deja pasar cualquier rol de dominio que un negocio necesite: `medicamento_prescrito`, `medida_de`, `diagnosticado_como`, `equipo`, `frecuencia`, `duracion`, `monto`… Ninguno está declarado en el núcleo y todos funcionan. Es la decisión que mantiene el motor pequeño mientras el lexicon crece.

Recetario por dominio

Lo que sigue es el código publicado en la Parte V, consolidado y agrupado por la maniobra que ilustra. Sirve como banco de patrones: cuando te topes con un caso parecido en tu dominio, copia el molde y cambia los identificadores.

Reificar una situación con polisemia

Vender una camiseta es una situación con vendedor, cliente, lugar y momento. El parámetro `complements` resuelve la ambigüedad del verbo: «vender» puede ser vender un producto o «venderse» (delatarse); el complemento lo decide.

PYTHON

```
vta = ingest_situation(u, lex, "vender",
    roles={
        "agente":   vendedor_17,
        "lugar_de": tienda_centro,
        "momento":  t_16_32,
    },
    complements=["camiseta"],          # dispara la acepción correcta
    extra={"estatus_factual": cerrada},
```

```
    sit_id="venta_001",
)
```

La desambiguación se sostiene en el lexicon: tres acepciones del mismo verbo, cada una con su `pattern` disparador y su `situation_type`. El parser elige de la más específica a la más general.

PYTHON

```
lex.register(LexiconEntry(
    verb="vender",
    situation_type="transaccion_venta",
    obligatory=["agente", "objeto"],
    pattern=("camiseta",),                # dispara con el complemento "camiseta"
))

lex.register(LexiconEntry(
    verb="vender",
    situation_type="accion_delatar",
    obligatory=["agente", "paciente"],
    pattern=("se",),                      # locución idiomática «venderse»
))

lex.register(LexiconEntry(
    verb="vender",
    situation_type="accion_vender",
    obligatory=["agente", "tema"],
    notes="genérico — ceder a cambio de un precio",
))
```

El dialecto de dominio: hablar el idioma del negocio

Un dialecto mapea el vocabulario del comercio a los roles canónicos. El usuario dice «vendedor», «venta», «artículo»; el sistema traduce a `agente`, `transaccion_venta`, `objeto` sin que se entere (D9).

PYTHON

```
lex.register_domain_dialect("tienda_aurora", {
    "vendedor":  "agente",
    "venta":     "transaccion_venta",
    "articulo":  "objeto",
    "fidelidad": "beneficio_fidelidad",
})
```

Consultar: patrón fijo + restricción de tipo

«¿Cuántas camisetas vendió Marco hoy?» es un conteo con sujeto fijo y restricción de tipo. El patrón ata `agente` y `estatus_factual`, y restringe a la clase `transaccion_venta`.

PYTHON

```
n = count(u, Pattern(
    fixed={"agente": vendedor_17, "estatus_factual": cerrada},
    type_constraint=u.ind("transaccion_venta"),
))
# n == 8
```

Y una decisión de negocio («a la séptima, una gratis») no es código del motor: es Python que consume el resultado de la consulta. La regla vive en la aplicación, no en el grafo.

PYTHON

```
califica = n >= 7
# True
```

Consulta bitemporal: la misma pregunta, distinto instante (D6)

La misma consulta sobre el mismo individuo devuelve respuestas distintas según el `at=` que recibe. La directora Serra vivió en `ciudad_a` hasta 2025 y luego se mudó a `ciudad_b`; el grafo recuerda ambas residencias.

PYTHON

```
res_2022 = query(u, Pattern(
    fixed={"agente": serra},
    ask={"lugar_de": Var()},
    type_constraint=u.ind("residencia"),
), at=datetime(2022, 6, 1, tzinfo=timezone.utc))
# res_2022[0]["lugar_de"].id == "ciudad_a"

res_2027 = query(u, ..., at=datetime(2027, 6, 1, tzinfo=timezone.utc))
# res_2027[0]["lugar_de"].id == "ciudad_b"
```

Software como agente (D5)

El rol `agente` no exige un humano. Al asignar un viaje, el agente es la *app*: cuatro participantes con roles distintos —app (agente, Q), solicitud previa (tema, O), conductor (beneficiario, Q), vehículo (instrumento, O)—.

PYTHON

```
asig = ingest_situation(u, lex, "asignar", roles={
    "agente":       app,            # ¡la APP ocupa el rol Q!
    "tema":         sol,            # asigna la solicitud previa
    "beneficiario": luis,           # al conductor disponible
    "instrumento":  vehiculo_12,    # con su vehículo
    "momento":      at(1),
}, sit_id="asig_001")
```

Causalidad emergente: explicar un valor

Una tarifa elevada no se afirma porque sí: queda explicada por `causado_por` apuntando a un estado de alta demanda, que a su vez se reifica con su lugar y su momento. Recuperar la explicación es un salto del grafo.

PYTHON

```
estado_demanda = u.add_individual(Individual(
    id="alta_demanda_2026_05_16_14_30", axis=Axis.O,
    label="alta demanda 16/5 14:30"))
u.assert_fact(estado_demanda, "instancia_de", alta_demanda)
u.assert_fact(estado_demanda, "lugar_de", plaza_mayor)
u.assert_fact(estado_demanda, "momento", at(0))

tarifa = u.add_individual(Individual(id="tarifa_viaje_088", axis=Axis.O))
u.assert_fact(tarifa, "instancia_de", category("tarifa"))
```

```
u.assert_fact(tarifa, "monto", n_25_usd)
u.assert_fact(tarifa, "causado_por", estado_demanda)
```

Y la consulta inversa («¿por qué costó tanto?») recorre los hechos de la tarifa buscando el cable `causado_por`:

PYTHON

```
explicaciones = u.facts_about(tarifa)
causa = [f for f in explicaciones if f.role == "causado_por"]
# causa[0].value.id == "alta_demanda_2026_05_16_14_30"
```

Nunca borramos: la cancelación como hecho nuevo

Cancelar no destruye nada. Es una situación nueva que `cancela` la previa y le pone `estatus_factual: cancelado`. El historial sobrevive intacto.

PYTHON

```
canc = ingest_situation(u, lex, "cancelar", roles={
    "agente":   valeria,
    "tema":     viaje_088,
    "momento":  at(62),
}, sit_id="canc_001")
u.assert_fact(canc, "cancela", viaje_088)
u.assert_fact(viaje_088, "estatus_factual", cancelado)
```

Rediagnóstico con vigencia y trazabilidad

Un diagnóstico se reemplaza por otro vía bitemporalidad. El primero es válido entre la consulta y el rediagnóstico; el segundo, desde ahí en adelante (sin `valid_to`). La relación `rectifica` deja el rastro del cambio.

PYTHON

```
# Diagnóstico original (vigente hasta el rediagnóstico)
u.assert_fact(diag, "diagnosticado_como", hta_g1,
              valid_from=t_consulta, valid_to=t_redx)

# Rediagnóstico (vigente desde t_redx, abierto al futuro)
u.assert_fact(diag2, "diagnosticado_como", hta_g2,
              valid_from=t_redx)
u.assert_fact(diag2, "rectifica", diag)
```

Roles de dominio que apuntan a K

«El medicamento prescrito» no es un objeto físico individual (O), sino una categoría farmacológica (K). El catálogo declara `tema: 0→0`, así que el dominio agrega su propio rol `medicamento_prescrito: 0→K` gracias a la política liberal. Lo mismo con `medida_de`.

PYTHON

```
# El medicamento prescrito es una categoría (K), no un objeto.
u.assert_fact(pres, "medicamento_prescrito", enalapril)

# La variable medida también es una categoría.
u.assert_fact(medicion, "medida_de", presion_arterial)
```

El «por qué» en triple: motivo, finalidad y verificación

Tres cables del porqué (D7) sobre una misma prescripción, sin pisarse: `motivado_por` apunta al diagnóstico, `con_finalidad` al objetivo terapéutico, `verificado_contra` a la regla que confirma que no contradice una contraindicación.

PYTHON

```
pres = ingest_situation(u, lex, "prescribir", roles={
    "agente":               dra_torres,
    "paciente":             vega,
    "medicamento_prescrito": enalapril,
    "frecuencia":           cada_manana,
    "duracion":             indefinida,
    "momento":              at(0),
})

u.assert_fact(pres, "motivado_por",      diag)       # por qué se prescribió
u.assert_fact(pres, "con_finalidad",     objetivo)   # para qué se prescribió
u.assert_fact(pres, "verificado_contra", regla_contraindicacion)
```

Recuperar una cadena por composición

Un partido completo es una entidad superior que contiene sus jugadas vía `parte_de`. Filtrar por ese cable devuelve toda la cadena de una sola pasada.

PYTHON

```
partes = [f.subject for f in u.facts_with_role("parte_de")
          if f.value.id == "partido_arg_per_2026"]
# devuelve todas las jugadas registradas del partido
```

Estado derivado: no se asienta, se calcula

El marcador de un partido no es un hecho que alguien escriba: emerge de contar los goles de cada equipo. La consulta es un doble conteo con una composición final.

PYTHON

```
goles_argentina = count(u, Pattern(
    fixed={"equipo": argentina},
    type_constraint=u.ind("gol_jugada_abierta"),
))
goles_peru = count(u, Pattern(
    fixed={"equipo": peru},
    type_constraint=u.ind("gol_jugada_abierta"),
))
marcador = f"{goles_argentina} – {goles_peru}"
```

Un LLM ingiere una nota en lenguaje natural

Como el lexicon expone las signaturas de los verbos como *function schemas*, un modelo de lenguaje puede leer una nota clínica y producir una invocación `ingest_situation` por verbo identificado. La extracción es directa: el modelo no inventa el esquema, lo lee del lexicon.

PYTHON

```
ingest_situation("consultar", roles={
    "agente": dra_torres, "paciente": vega,
    "momento": fecha_nota, "motivo": cefalea,
})
ingest_situation("medir", roles={
    "agente": dra_torres, "paciente": vega,
    "medida_de": presion_arterial,
    "monto": "145/92", "unidad": mmHg,
})
ingest_situation("diagnosticar", roles={
    "agente": dra_torres, "paciente": vega,
    "diagnosticado_como": hta_g1,
})
ingest_situation("prescribir", roles={
    "agente": dra_torres, "paciente": vega,
    "medicamento_prescrito": enalapril,
    "frecuencia": cada_manana,
})
ingest_situation("controlar", roles={
    "paciente": vega, "agente": dra_torres,
    "momento": fecha_nota_mas_30,
    "estatus_factual": previsto,
})
```

Una consulta que cruza dominios

El mismo grafo permite cruzar territorios que en sistemas tradicionales jamás se hablarían. Aquí, dos consultas paralelas —una a un dominio público (una ordenanza), otra a uno de telemetría (sesiones de IA)— que la aplicación combina en una respuesta única.

PYTHON

```
normas = query(u, Pattern(
    fixed={"tema_categorico": micromovilidad},
    type_constraint=u.ind("ordenanza_municipal"),
    ask={"agente": Var(), "momento": Var(), "entra_en_vigor": Var()},
))

uso = query(u, Pattern(
    fixed={"modelo": modelo_lumen_2026},
    type_constraint=u.ind("sesion_ia"),
    ask={"momento": Var(), "tokens_salida": Var()},
))
```

Formatos: cómo viaja un hecho

El mismo hecho se puede mirar en tres representaciones. Todas dicen lo mismo; cambian la audiencia. La *tripleta* es para leer; el *JSON* es para transmitir; el *function schema* es para que un LLM lo invoque.

Forma tripleta (para leer)

La notación canónica del libro. Cada línea es un hecho atómico; el comentario tras `∈` recuerda la firma del cable. Esta forma es la que aparece en las figuras y la que conviene tener en la cabeza al modelar.

TRIPLETAS

```
(venta_001) ∈ O                                  # la situación
(venta_001, instancia_de, venta)          ∈ M(O, K)
(venta_001, agente,       vendedor_17)    ∈ M(O, Q)
(venta_001, objeto,       camiseta_88)    ∈ M(O, O)
(venta_001, lugar_de,     tienda_centro)  ∈ M(O, L)
(venta_001, momento,      16:32)          ∈ M(O, T)
(venta_001, monto_usd,    49.90)          ∈ M(O, N)   # USD
(venta_001, estatus_factual, cerrada)     ∈ M(O, K)
```

Forma JSON (para transmitir)

La serialización por la que viaja un hecho entre sistemas. Cada hecho lleva su sujeto, su rol y su valor; el valor distingue entre referencia a otro nodo (`ref`) y literal (`lit`) con su eje. La bitemporalidad va en `valido` .

JSON

```
{
  "sujeto": "venta_001",
  "eje_sujeto": "O",
  "hechos": [
    {"rol": "instancia_de", "valor": {"ref": "venta", "eje": "K"}},
    {"rol": "agente",       "valor": {"ref": "vendedor_17", "eje": "Q"}},
    {"rol": "objeto",       "valor": {"ref": "camiseta_88", "eje": "O"}},
    {"rol": "lugar_de",     "valor": {"ref": "tienda_centro", "eje": "L"}},
    {"rol": "momento",      "valor": {"lit": "2026-05-14T16:32:00-05:00", "eje": "T"}},
    {"rol": "monto_usd",    "valor": {"lit": 49.90, "eje": "N", "unidad": "Currency:USD"}}
  ],
  "valido": {"desde": "2026-05-14T16:32:00-05:00", "hasta": null}
}
```

Forma function schema (para que un LLM lo invoque)

El lexicon proyecta cada verbo como una herramienta que un modelo de lenguaje puede llamar. Los roles obligatorios de la signatura se vuelven parámetros requeridos; los opcionales, opcionales. Así el LLM no improvisa el esquema: lo recibe.

JSON

```
{
  "name": "ingest_vender_camiseta",
  "description": "Registra que un vendedor vendió una camiseta a un cliente.",
  "parameters": {
    "type": "object",
    "properties": {
      "agente":   {"type": "string", "description": "quién la vendió (Q)"},
      "objeto":   {"type": "string", "description": "qué se vendió (O)"},
      "lugar_de": {"type": "string", "description": "dónde se vendió (L)"},
      "momento":  {"type": "string", "format": "date-time"},
      "monto_usd": {"type": "number", "description": "importe en dólares (N)"}
    },
    "required": ["agente", "objeto"]
  }
}
```

Lo que se ve al mirarlo todo junto

Reunir el código disperso revela patrones que el flujo capítulo a capítulo diluye. Cinco regularidades valen la pena nombrar, porque explican por qué el modelo escala sin engordar.

1 LA API ES DIMINUTA

Cuatro funciones (`ingest_situation`, `assert_fact`, `query`, `count`) modelan ocho dominios completos. No hay doscientas funciones que aprender; hay cuatro.

2 LAS LLAMADAS SON UNIFORMES

El patrón visual es siempre el mismo: verbo, diccionario de roles, opcionales al final. Una vez que reconoces la silueta, el código se lee como prosa.

3 EL VERBO VA PRIMERO

Se elige el verbo y después los roles, no al revés. Refleja la signatura tipada del verbo: primero el contrato, después los argumentos.

4 LA POLÍTICA LIBERAL TRABAJA SOLA

Roles que ningún capítulo declara (`medicamento_prescrito`, `equipo`, `frecuencia`) el motor los acepta. Valida lo canónico; deja pasar lo de dominio.

5 INMUTABILIDAD Y BITEMPORALIDAD POR DEFECTO

Casi todo fragmento importante asienta con `valid_from` / `valid_to`, marca `estatus_factual` o `rectifica` sin borrar. El modelo no tiene una operación «update»: tiene «asentar un hecho nuevo que reemplaza al previo conservando el historial».

> *El motor no crece con cada dominio nuevo; el lexicon y los datos sí. Esa es la propiedad que mantiene corto el núcleo a pesar de cubrir un spa, un banco, una historia clínica y un partido con la misma maquinaria.*

LA ECONOMÍA DEL MODELO

Esas cinco regularidades, juntas, explican por qué el núcleo de modelado cabe en poco más de dos mil líneas a pesar de la diversidad de territorios que cubre. Y anticipan la prueba más exigente, la del capítulo 28: una aplicación con menús, formularios y entidades enteras que se levantó sobre este mismo núcleo casi sin agregarle código, porque también su estructura y su conducta son, al final, datos. El siguiente anexo abre esa caja y muestra el prototipo por dentro.

ANEXOS

33

Anexo: el prototipo

Todo lo que el libro afirma (las siete coordenadas, los hechos con signatura, las situaciones reificadas, la bitemporalidad, el motor de consulta) cabe en nueve archivos cortos de Python que corren sin una sola dependencia externa.

Una arquitectura que solo vive en prosa es una promesa; una arquitectura que corre es una prueba. Este anexo reúne el **código fuente íntegro de la librería núcleo** del prototipo de WQuestions: alrededor de ochocientas cincuenta líneas repartidas en nueve módulos. No es pseudocódigo ni un boceto ilustrativo. Es Python real, ejecutable, que da cuerpo a cada decisión de diseño discutida en los capítulos anteriores y que sirvió de banco de pruebas mientras el modelo tomaba forma, dominio tras dominio.

> **ALCANCE**
> El anexo anterior recoge el código que acompaña a los capítulos a lo largo del libro. Este se concentra en una sola cosa: la *librería núcleo*, leída de principio a fin como un programa coherente.

Conviene fijar el alcance desde la primera línea, sin letra pequeña. Lo que sigue es **solo la librería núcleo** (el paquete `wq/`). Los ejemplos por dominio que ejercita la Parte V (el spa, el taxi, la clínica, el banco), la batería de tests del modelo y los scripts de utilidad *no aparecen aquí*, pero existen, corren y están publicados. El proyecto completo vive en el repositorio del libro, y al final de este anexo se indica exactamente qué encontrarás allí y cómo ponerlo en marcha en cinco minutos.

> **QUÉ DEMUESTRA ESTE CÓDIGO**
> Que el modelo de las preguntas no es una metáfora afortunada, sino una arquitectura *coherente* y *operable*. Ochocientas cincuenta líneas bastan para sostener las siete coordenadas, validar hechos contra sus signaturas, reificar situaciones, resolver polisemia y responder preguntas-WH con vigencia temporal. La validación industrial es otro trabajo (el del capítulo 30); lo que estas líneas prueban es que la idea cierra.

La arquitectura en una sola mirada

El prototipo se organiza en nueve módulos que se apilan como capas. En la base están los siete ejes; sobre ellos, la noción de individuo; sobre los individuos, el hecho atómico; a un lado, las dos estructuras transversales (el catálogo de roles y el lexicon); por encima de todo, el universo que lo contiene todo y, en la cima, la superficie pública que reexporta lo que el usuario va a tocar. Cada decisión de diseño del libro aterriza en un módulo concreto: el paralelismo es deliberado y casi uno a uno.

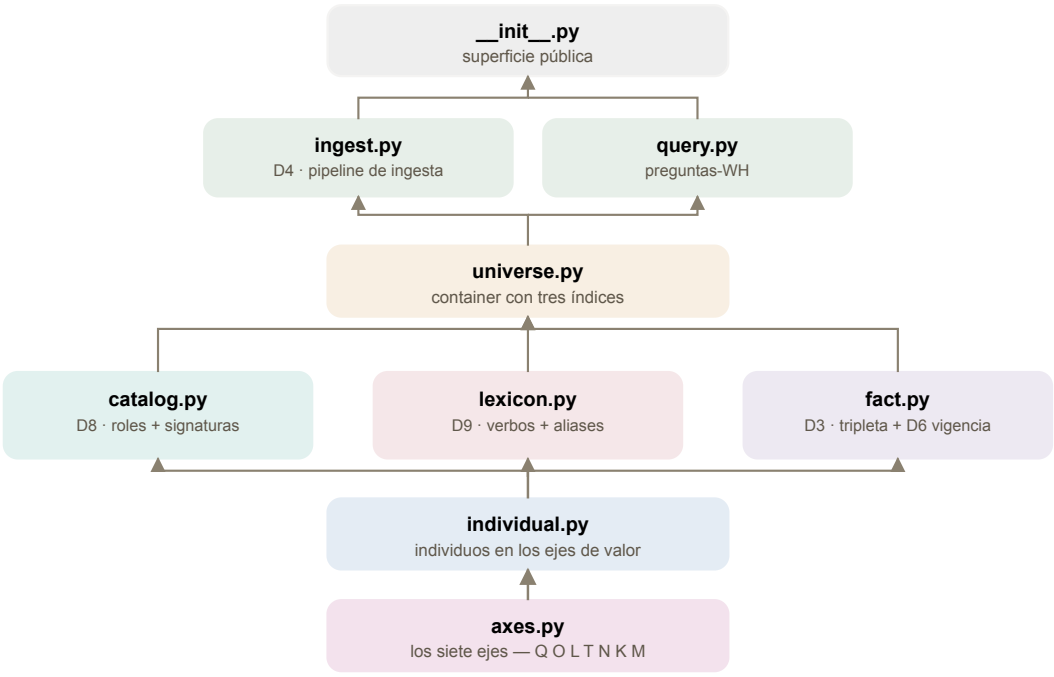


Figura 33.1. Los nueve módulos del paquete `wq/`, apilados de la base a la cima. El orden de lectura natural va de abajo hacia arriba: primero `axes.py` (los siete ejes), luego los individuos, luego las tres estructuras transversales —catálogo (K), lexicon (Q) y hecho (T)—, luego el universo que las une, y por fin la API de ingesta y consulta. Las nueve secciones de este anexo siguen ese mismo orden.

El recorrido de lectura es de abajo hacia arriba. Empezamos por la base (los ejes), subimos a los individuos que los pueblan, llegamos al hecho atómico, abrimos en abanico las dos piezas transversales, las reunimos en el universo y rematamos con la fachada que expone todo al usuario. Cada sección abre con una nota breve sobre qué resuelve el módulo y termina con su código íntegro.

CÓMO LEER EL CÓDIGO DE ESTE ANEXO

Los bloques están coloreados con el resaltador del libro: las palabras clave de Python, las cadenas, los números y los comentarios se distinguen al vuelo, y los identificadores de eje (`Q` , `O` , `L` , `T` , `N` , `K` , `M`) llevan su color habitual. Si una línea es larga, el bloque permite desplazarse en horizontal sin romper la página. El botón *copiar* de cada bloque entrega el código tal cual, listo para pegar en un archivo.

§A1 `axes.py` — los siete ejes

Es la pieza fundacional y la más corta. Declara, como un `Enum`, los seis ejes de valor (los que albergan individuos) y el único eje estructural de predicados. A partir de aquí, todo el modelo se referencia contra estos siete identificadores: no hay cadenas mágicas dispersas por el código, hay un vocabulario cerrado y verificable.

La distinción central que codifica este archivo es la que recorre todo el libro: seis ejes *contienen cosas* y uno *conecta cosas*. Q guarda agentes, O objetos y situaciones, L lugares, T momentos, N magnitudes y K categorías atemporales. El séptimo, M, no guarda valores: guarda los cables que enlazan los valores de los otros seis.

DETALLE

El `Enum` incluye un valor extra, `V`, que no es un eje del modelo sino un comodín de signatura: significa «cualquier eje de valor». Lo usa el catálogo para roles cuyo sujeto puede vivir en más de un eje, como `instancia_de`.

PYTHON

```python
"""Los 7 ejes de WQuestions.

Seis ejes de valor (contienen individuos):
    Q — quién — agentes
    O — qué  — objetos / situaciones reificadas
    L — dónde — lugares
    T — cuándo — momentos / intervalos
    N — cuánto — magnitudes con unidad
    K — cuál — categorías atemporales (tipos, unidades, estados, vocabularios)

Un eje estructural (contiene etiquetas con signatura):
    M — cómo — predicados/cables que conectan individuos. Cada predicado
        declara su cardinalidad en la signatura (`functional`: un solo valor
        por sujeto, o múltiple). La cardinalidad es un atributo del predicado,
        no un eje aparte.
"""

from enum import Enum


class Axis(Enum):
    Q = "Q"  # quién
    O = "O"  # qué (objectum)
    L = "L"  # dónde
    T = "T"  # cuándo (tempus)
    N = "N"  # cuánto
    K = "K"  # cuál (categórico)
    M = "M"  # cómo / predicados (modus)
    V = "V"  # comodín de signatura: cualquier eje de valor


VALUE_AXES = {Axis.Q, Axis.O, Axis.L, Axis.T, Axis.N, Axis.K}
PREDICATE_AXES = {Axis.M}


def is_value_axis(axis: Axis) -> bool:
    return axis in VALUE_AXES
```

SA2 `individual.py` — individuos en los ejes

Ochenta y un líneas. Define qué es un *individuo*: la entidad concreta que habita en alguno de los seis ejes de valor. Un agente, una situación, un lugar, un instante, una magnitud o una categoría son todos individuos; lo que cambia es el eje en el que viven. La clase es inmutable (un `dataclass` congelado) porque en este modelo un individuo, una vez acuñado, no cambia de identidad ni de eje.

El archivo aporta además tres fábricas de individuos «primitivos», esos que en realidad son datos: `time_point` crea un instante en T a partir de una marca ISO 8601, `quantity` crea una magnitud en N anclando su unidad en K, y `category` crea una categoría en K. El detalle de que `quantity` exija que la unidad viva en K no es decorativo: es la regla del eje cuantitativo hecha código: un número sin unidad anclada simplemente no se puede construir.

INDIVIDUO

Instancia concreta que ocupa **uno y solo un** eje de valor, con un identificador estable. El campo `payload` permite alojar el valor nativo cuando el individuo *es* un dato: el `datetime` de un instante, el par `{valor, unidad}` de una magnitud. Para agentes y situaciones, el identificador basta y el `payload` queda vacío.

PRODUCCIÓN

El generador `mint_id` produce identificadores cortos y monótonos (`n_000001`, `n_000002`…). Es una elección de observabilidad para el prototipo: en producción se reemplaza por UUID v7, que sacrifica legibilidad a cambio de unicidad global.

PYTHON

```python
"""Individuos del modelo: instancias de cualquier eje de valor.

Cada individuo tiene un identificador estable y vive en uno y solo un eje.
Para sujetos sintéticos usamos un mint determinista (id corto, monótono)
en vez de UUID — la observabilidad gana, la pérdida de globalidad no
importa para un prototipo.
"""

from __future__ import annotations
from dataclasses import dataclass, field
from typing import Optional, Any, Dict
from itertools import count

from .axes import Axis, is_value_axis


_counter = count(1)


def mint_id(prefix: str = "id") -> str:
    """Genera un id corto y monótono. En producción se reemplazaría por UUID v7."""
    n = next(_counter)
    return f"{prefix}_{n:06d}"


@dataclass(frozen=True)
class Individual:
    """Individuo en un eje de valor.

    `payload` permite alojar valores nativos cuando el "individuo" es en
    realidad un dato (un instante T, una magnitud N, una categoría K). Para
    Q y O suele ser None; el id basta.
    """

    id: str
    axis: Axis
    label: Optional[str] = None
    payload: Any = None
    meta: Dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):
        if not is_value_axis(self.axis):
            raise ValueError(
                f"Un individuo debe vivir en un eje de valor, no en {self.axis}."
            )

    def __repr__(self) -> str:
        tag = self.label or self.id
        return f"<{self.axis.value}:{tag}>"


# --- helpers para individuos "primitivos" ------------------------------------


def time_point(iso: str) -> Individual:
    """Crea un individuo T desde una marca ISO 8601."""
```

```
    from datetime import datetime
    try:
        dt = datetime.fromisoformat(iso.replace("Z", "+00:00"))
    except ValueError as e:
        raise ValueError(f"Marca temporal inválida: {iso}") from e
    return Individual(id=f"t_{iso}", axis=Axis.T, label=iso, payload=dt)


def quantity(value: float, unit_k: "Individual") -> Individual:
    """Crea un individuo N con valor numérico y unidad anclada en K."""
    if unit_k.axis != Axis.K:
        raise ValueError(
            f"La unidad debe vivir en K (recibido: {unit_k.axis})."
        )
    return Individual(
        id=mint_id("n"),
        axis=Axis.N,
        label=f"{value} {unit_k.label or unit_k.id}",
        payload={"value": value, "unit": unit_k.id},
    )


def category(label: str) -> Individual:
    """Crea un individuo K (categoría) con id derivado del label."""
    return Individual(id=label, axis=Axis.K, label=label)
```

§A3 `fact.py` — el hecho atómico con bitemporalidad

Sesenta y una líneas para la unidad mínima del modelo. Un hecho es una tupla inmutable `(sujeto, rol, valor)` donde sujeto y valor son individuos (viven en algún eje de valor) y el rol es una etiqueta del catálogo. Esto es **D3** en estado puro: todo, absolutamente todo lo que el modelo sabe, se descompone en estas tripletas tipadas. Lo complejo no se modela con estructuras más ricas: se apila como más tripletas.

> **D3 EL HECHO ATÓMICO**
>
> La clase `Fact` es la encarnación literal de D3: una tripleta `(subject, role, value)`, congelada e inmutable. No hay forma de representar un hecho fuera de esta forma. La riqueza del mundo se reconstruye apilando tripletas, no inflando la estructura de cada una.

Sobre esa base mínima, el archivo añade la dimensión temporal que pide **D6**. Cada hecho lleva un rango de vigencia opcional `[valid_from, valid_to)` (desde cuándo y hasta cuándo es cierto *en el mundo*) y un `tx_time` automático que registra cuándo entró *al sistema*. Son dos líneas de tiempo distintas: el tiempo de los hechos y el tiempo de los registros. Esa separación es lo que da bitemporalidad, y basta para modelar mudanzas, rediagnósticos o cláusulas contractuales que expiran.

El método `is_valid_at` condensa la lógica de D6 en seis líneas: si no hay rango de vigencia, el hecho es atemporal y vale siempre; si lo hay, el momento consultado debe caer en el intervalo, con el inicio inclusivo y el fin exclusivo. Un `valid_to` nulo significa «abierto al futuro».

> **CONVENCIÓN**
>
> Intervalo semiabierto `[inicio, fin)`: el instante de inicio cuenta, el de fin no. Es la misma convención que evita solapamientos en bases de datos temporales y la que usa el capítulo 9 al hablar de vigencia.

PYTHON

```
"""Hechos atómicos: la unidad mínima del modelo.

Cada hecho es una tupla `(sujeto, rol, valor)` donde:
- `sujeto` y `valor` son `Individual` (viven en algún eje de valor).
- `rol` es una etiqueta del catálogo canónico (D8) o de su capa lexicon.

D6 — vigencia temporal: cada hecho lleva opcionalmente un rango
`[valid_from, valid_to)` que indica desde cuándo y hasta cuándo es cierto en
el mundo. Si `valid_to is None`, está abierto al futuro.

Adicionalmente registramos `tx_time` (transaction time): el momento en el que
el hecho entró al sistema. Esto da bitemporalidad ligera, suficiente para
auditoría.
"""

from __future__ import annotations
from dataclasses import dataclass, field
from datetime import datetime, timezone
from typing import Optional

from .individual import Individual


def _utcnow() -> datetime:
    return datetime.now(timezone.utc)


@dataclass(frozen=True)
class Fact:
    subject: Individual
    role: str
    value: Individual
    valid_from: Optional[datetime] = None
    valid_to: Optional[datetime] = None
    tx_time: datetime = field(default_factory=_utcnow)

    def is_valid_at(self, moment: datetime) -> bool:
        """¿Este hecho es cierto en el mundo en `moment`?

        Reglas:
        - Si no hay vigencia, el hecho es atemporal: vale siempre.
        - `valid_from` inclusivo, `valid_to` exclusivo.
        - `valid_to is None` significa "abierto al futuro".
        """
        if self.valid_from is None and self.valid_to is None:
            return True
        if self.valid_from is not None and moment < self.valid_from:
            return False
        if self.valid_to is not None and moment >= self.valid_to:
            return False
        return True

    def __repr__(self) -> str:
        s = self.subject.label or self.subject.id
        v = self.value.label or self.value.id
        base = f"({s}, {self.role}, {v})"
        if self.valid_from or self.valid_to:
            f = self.valid_from.isoformat() if self.valid_from else "-∞"
            t = self.valid_to.isoformat() if self.valid_to else "+∞"
            base += f"  [{f} .. {t})"
        return base
```

> *Un hecho es una tripleta congelada con dos líneas de tiempo: la del mundo y la del registro. Todo lo demás se construye apilando hechos.*

EL CORAZÓN DEL PROTOTIPO

§A4 `catalog.py` — el catálogo canónico de roles

Ciento setenta y nueve líneas: es el módulo más largo de la librería núcleo y donde se materializa **D8**. El catálogo declara los treinta y ocho roles canónicos del modelo, cada uno con su signatura tipada (un *dominio* y un *rango*, ambos ejes de valor) y su cardinalidad. Esa signatura es lo que convierte una tripleta de texto libre en un hecho verificable: al insertar `(sujeto, rol, valor)`, el catálogo comprueba que el eje del sujeto coincida con el dominio del rol y el eje del valor con su rango.

D8 EL CATÁLOGO CANÓNICO DE ROLES

Cada rol es un predicado del eje M con una signatura `dominio → rango`. La clase `RoleSignature` la guarda; el método `validate` la hace cumplir. Así, `agente` solo admite un sujeto en O y un valor en Q; `monto`, un valor en N; `lugar_de`, uno en L. La basura no entra al grafo.

Conviene detenerse en una decisión que el libro defiende y que aquí ocupa apenas cuatro líneas: la **política liberal**. Si un rol no está declarado en el catálogo, el método `validate` *no lo rechaza*: simplemente no lo valida y lo deja pasar. Una política estricta abortaría ante cualquier rol desconocido; el prototipo prefiere la extensibilidad. Quien quiera un rol nuevo puede usarlo de inmediato sin pedir permiso al catálogo central; si más tarde quiere validación, lo registra. La elección está comentada en el propio código, a la vista.

La cardinalidad (el campo `functional`) merece una nota. Un rol funcional admite un solo valor por sujeto: una situación tiene un `agente`, un `lugar_de`, un `inicio`. Un rol no funcional admite varios: una situación puede estar `causado_por` varias otras, o `contiene` muchas partes. Esto es **D2** en acción: propiedades y relaciones no son cosas distintas; son cables del mismo eje M que solo difieren en cuántos valores admiten.

SUBCONJUNTO

Los treinta y ocho roles que carga `_load_canonical` son un subconjunto representativo del catálogo completo descrito en el documento maestro. Cubren los grupos que el libro discute: estructurales, participantes, lugar y tiempo, cuantitativos, clasificatorios, los cuatro cables del «por qué» (**D7**) e inter-situacionales.

PYTHON

```
"""Catálogo canónico de roles (D8).

Cada rol declara una signatura tipada `dominio → rango`, ambos ejes de valor,
más su **cardinalidad** (`functional`: un solo valor por sujeto, o múltiple).
Todos los roles son predicados del eje M (cómo); la cardinalidad es un atributo
de la signatura, no un eje aparte. La signatura habilita la validación mecánica
de hechos: al insertar `(s, role, v)` el catálogo verifica que `s.axis` coincida
con el dominio y `v.axis` con el rango.
"""

from __future__ import annotations
from dataclasses import dataclass
from typing import Dict, Optional

from .axes import Axis
```

```
from .individual import Individual


@dataclass(frozen=True)
class RoleSignature:
    name: str
    domain: Axis    # eje del sujeto
    range: Axis     # eje del valor
    functional: bool  # True → un valor por sujeto, False → múltiple
    description: str = ""


class SignatureError(ValueError):
    pass


class Catalog:
    """Catálogo de signaturas canónicas + validación de hechos."""

    def __init__(self):
        self._roles: Dict[str, RoleSignature] = {}
        self._load_canonical()

    def register(self, sig: RoleSignature) -> None:
        if sig.name in self._roles:
            existing = self._roles[sig.name]
            if existing != sig:
                raise SignatureError(
                    f"Rol '{sig.name}' ya registrado con signatura distinta: "
                    f"{existing} vs {sig}"
                )
            return
        self._roles[sig.name] = sig

    def get(self, name: str) -> Optional[RoleSignature]:
        return self._roles.get(name)

    def validate(self, role: str, subject: Individual, value: Individual) -> None:
        """Lanza `SignatureError` si el hecho viola la signatura."""
        sig = self._roles.get(role)
        if sig is None:
            # Rol no declarado: política liberal — se permite, no se valida.
            # Una política estricta lo rechazaría; preferimos extensibilidad.
            return
        if sig.domain != Axis.V and subject.axis != sig.domain:
            raise SignatureError(
                f"Sujeto en eje incorrecto para '{role}': se esperaba "
                f"{sig.domain.value}, recibido {subject.axis.value} "
                f"(sujeto={subject})"
            )
        if sig.range != Axis.V and value.axis != sig.range:
            raise SignatureError(
                f"Valor en eje incorrecto para '{role}': se esperaba "
                f"{sig.range.value}, recibido {value.axis.value} "
                f"(valor={value})"
            )

    def __contains__(self, name: str) -> bool:
        return name in self._roles

    def __len__(self) -> int:
        return len(self._roles)

    # ------------------------------------------------------------------
```

```
    # Carga del catálogo canónico (subset del documento WQuestions.md)
    # ------------------------------------------------------------------

    def _load_canonical(self) -> None:
        canonical = [
            # --- estructurales ---
            RoleSignature("instancia_de", Axis.V, Axis.K, False,
                          "sujeto (cualquier eje de valor) pertenece a la categoría"),
            RoleSignature("subtipo_de", Axis.K, Axis.K, False,
                          "subtipo conceptual"),
            RoleSignature("parte_de", Axis.O, Axis.O, False,
                          "subobjeto/subevento de"),
            RoleSignature("contiene", Axis.O, Axis.O, False,
                          "inversa de parte_de"),

            # --- participantes (Q es típico) ---
            RoleSignature("agente", Axis.O, Axis.Q, True,
                          "agente principal de la situación"),
            RoleSignature("paciente", Axis.O, Axis.Q, True,
                          "afectado por la situación"),
            RoleSignature("tema", Axis.O, Axis.O, True,
                          "objeto temático (cosa o sub-situación)"),
            RoleSignature("beneficiario", Axis.O, Axis.Q, True,
                          "destinatario o beneficiario"),
            RoleSignature("experimentador", Axis.O, Axis.Q, True,
                          "quien experimenta un estado mental"),
            RoleSignature("instrumento", Axis.O, Axis.O, True,
                          "objeto usado para ejecutar la acción"),
            RoleSignature("comprador", Axis.O, Axis.Q, True,
                          "comprador en una venta"),
            RoleSignature("cliente", Axis.O, Axis.Q, True,
                          "cliente de un servicio (alias frecuente de agente)"),

            # --- lugar / tiempo ---
            RoleSignature("lugar_de", Axis.O, Axis.L, True,
                          "lugar donde ocurre la situación"),
            RoleSignature("origen", Axis.O, Axis.L, True,
                          "lugar de origen"),
            RoleSignature("destino", Axis.O, Axis.L, True,
                          "lugar de destino"),
            RoleSignature("lugar_destino", Axis.O, Axis.L, True,
                          "alias de destino"),
            RoleSignature("momento", Axis.O, Axis.T, True,
                          "momento puntual"),
            RoleSignature("inicio", Axis.O, Axis.T, True,
                          "instante de inicio"),
            RoleSignature("fin", Axis.O, Axis.T, True,
                          "instante de fin"),

            # --- cuantitativos ---
            RoleSignature("monto", Axis.O, Axis.N, True,
                          "cantidad numérica con unidad"),
            RoleSignature("cantidad", Axis.O, Axis.N, True,
                          "alias de monto"),
            RoleSignature("por_cuanto", Axis.O, Axis.N, True,
                          "precio o medida asociada"),
            RoleSignature("unidad", Axis.O, Axis.K, True,
                          "unidad de medida (QUDT)"),

            # --- clasificatorios ---
            RoleSignature("estatus_factual", Axis.O, Axis.K, True,
                          "real / intencionado / no_realizable / ..."),
            RoleSignature("modalidad", Axis.O, Axis.K, True,
                          "volitiva / deóntica / alética / epistémica"),
```

```
            RoleSignature("polaridad", Axis.O, Axis.K, True,
                          "afirmativa / negativa"),
            RoleSignature("calificacion", Axis.O, Axis.K, True,
                          "atributo cualitativo"),

            # --- "por qué" (D7, capítulo 10) ---
            RoleSignature("causado_por", Axis.O, Axis.O, False,
                          "causalidad mecánica"),
            RoleSignature("motivado_por", Axis.O, Axis.O, False,
                          "motivación intencional"),
            RoleSignature("con_finalidad", Axis.O, Axis.O, False,
                          "propósito"),
            RoleSignature("justificado_por", Axis.O, Axis.O, False,
                          "regla que autoriza"),

            # --- inter-situacionales ---
            RoleSignature("precede", Axis.O, Axis.O, False,
                          "orden lógico/temporal"),
            RoleSignature("sigue_a", Axis.O, Axis.O, False,
                          "inversa de precede"),
            RoleSignature("cumple", Axis.O, Axis.O, False,
                          "cumple una obligación"),
            RoleSignature("cancela", Axis.O, Axis.O, False,
                          "deja sin efecto"),
            RoleSignature("rectifica", Axis.O, Axis.O, False,
                          "corrige otra situación"),
            RoleSignature("contrasta_con", Axis.O, Axis.O, False,
                          "relación adversativa (\"pero\")"),

            # --- atributos del sujeto Q ---
            RoleSignature("nombre", Axis.Q, Axis.K, True,
                          "nombre de un agente"),
            RoleSignature("identificador", Axis.Q, Axis.K, True,
                          "id documental"),
        ]
        for sig in canonical:
            self.register(sig)
```

SA5 `lexicon.py` — el lexicon con resolución de polisemia

Ciento dieciséis líneas. Aquí vive la capa que traduce entre el lenguaje del usuario y los roles canónicos (la materialización de **D9**). Cada entrada del lexicon es un `LexiconEntry`: declara un verbo, el tipo de situación que produce, sus roles obligatorios y opcionales, los aliases naturales por rol y por dominio, sus formas nominales y, sobre todo, un `pattern` de complementos que es el motor de la desambiguación.

> **D9 EL USUARIO NUNCA TOCA ETIQUETAS CANÓNICAS**
>
> El lexicon es la interfaz; el catálogo, invisible. Un negocio puede llamar «socio» a lo que el modelo registra como `cliente`, o «huésped» a lo que es un `paciente`: el método `translate_alias` y los dialectos de dominio absorben esa diferencia. El usuario habla su idioma; el sistema traduce por debajo.

La pieza ingeniosa es la resolución de polisemia. Un mismo verbo puede tener varias entradas (`dar la mano` no es `dar una conferencia`), y el lexicon elige la **más específica que coincide**. La especificidad se mide por la longitud del patrón de complementos: una entrada con patrón `('la_mano',)` es más específica que una entrada genérica sin patrón. Las entradas se

mantienen ordenadas de más específica a más general, de modo que `resolve` devuelve la primera que encaja con los complementos observados en la oración.

El módulo también resuelve nominalizaciones («la llegada del avión» remite al verbo `llegar`) mediante un índice de formas nominales, y guarda los dialectos de dominio en una tabla aparte. Tres mecanismos distintos, una sola idea: que la riqueza y la ambigüedad del lenguaje natural se resuelvan *antes* de tocar el grafo, no dentro de él.

PYTHON

```
"""Lexicon: diccionario verbo → tipo de situación + roles + aliases.

Implementa el paradigma del capítulo 14: cada entrada declara una
signatura (qué roles obligatorios y opcionales), un tipo de situación en K,
y aliases naturales por rol y por dominio. La resolución de polisemia
elige la entrada más específica que coincide con el patrón de complemento.
"""

from __future__ import annotations
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Tuple


@dataclass
class LexiconEntry:
    verb: str                          # "vender", "dar", ...
    situation_type: str                # K id, ej. "accion_vender"
    obligatory: List[str] = field(default_factory=list)
    optional: List[str] = field(default_factory=list)
    aliases: Dict[str, List[str]] = field(default_factory=dict)
    nominal_forms: List[str] = field(default_factory=list)
    pattern: Optional[Tuple[str, ...]] = None  # complementos clave para polisemia
    notes: str = ""
    example: str = ""

    def specificity(self) -> int:
        """Cuanto más largo el patrón, más específica la entrada."""
        return 0 if self.pattern is None else len(self.pattern)

    def matches(self, observed_complements: List[str]) -> bool:
        """¿Esta entrada coincide con los complementos vistos en la oración?

        Una entrada con `pattern=None` siempre coincide (entrada genérica).
        Una entrada con patrón coincide si TODOS sus elementos aparecen
        entre los complementos.
        """
        if self.pattern is None:
            return True
        obs = set(observed_complements)
        return all(p in obs for p in self.pattern)


class Lexicon:
    """Lexicon con resolución de polisemia.

    Las entradas se registran por verbo; varias entradas pueden compartir
    verbo (polisemia). `resolve(verb, complements)` devuelve la entrada
    más específica que coincide.
    """

    def __init__(self):
        self._by_verb: Dict[str, List[LexiconEntry]] = {}
        # Aliases globales de dominio: nombre_usuario → rol_canonico
```

```
        self._domain_aliases: Dict[str, Dict[str, str]] = {}
        # Aliases nominales globales: forma_nominal → verbo
        self._nominal_index: Dict[str, str] = {}

    # --- registro ------------------------------------------------------

    def register(self, entry: LexiconEntry) -> None:
        self._by_verb.setdefault(entry.verb, []).append(entry)
        # Orden por especificidad decreciente: más específico primero.
        self._by_verb[entry.verb].sort(key=lambda e: -e.specificity())
        for nf in entry.nominal_forms:
            self._nominal_index[nf] = entry.verb

    def register_domain_dialect(self, domain: str,
                                aliases: Dict[str, str]) -> None:
        """Agrega un dialecto de dominio: nombre_usuario → rol_canónico."""
        self._domain_aliases.setdefault(domain, {}).update(aliases)

    # --- resolución ----------------------------------------------------

    def resolve(self, verb: str,
                complements: Optional[List[str]] = None) -> Optional[LexiconEntry]:
        """Devuelve la entrada más específica que coincide con el verbo
        y los complementos observados. None si no hay match.
        """
        candidates = self._by_verb.get(verb, [])
        comps = complements or []
        for entry in candidates:  # ya ordenadas más específico → más general
            if entry.matches(comps):
                return entry
        return None

    def resolve_nominal(self, nominal_form: str,
                        complements: Optional[List[str]] = None
                        ) -> Optional[LexiconEntry]:
        """Resolución por forma nominal (nominalización).

        *"la llegada del avión"* → verbo `llegar` → entrada de llegar.
        """
        verb = self._nominal_index.get(nominal_form)
        if verb is None:
            return None
        return self.resolve(verb, complements)

    def translate_alias(self, domain: str, user_term: str) -> Optional[str]:
        """Traduce un término de usuario al rol canónico vía dialecto."""
        return self._domain_aliases.get(domain, {}).get(user_term)

    def alias_for_role(self, verb: str, role: str) -> List[str]:
        """Aliases declarados para un rol de una entrada específica."""
        entries = self._by_verb.get(verb, [])
        for e in entries:
            if role in e.aliases:
                return e.aliases[role]
        return []

    # --- introspección -------------------------------------------------

    def verbs(self) -> List[str]:
        return list(self._by_verb.keys())

    def entries_for(self, verb: str) -> List[LexiconEntry]:
        return list(self._by_verb.get(verb, []))
```

§A6 `universe.py` — el universo V

Ciento treinta líneas. El universo es el container que aloja todos los individuos y todos los hechos. Por dentro mantiene tres índices (por sujeto, por valor y por rol) para que las consultas no degeneren en un barrido lineal sobre toda la colección. Es aquí donde se conecta el catálogo: cada `assert_fact` delega la validación de signatura al catálogo si se le inyectó uno. Sin catálogo, el universo acepta cualquier hecho bien formado; con catálogo, rechaza los que violan una signatura declarada.

LOS HECHOS NO SE SOBREESCRIBEN JAMÁS

El universo solo *agrega*. Un dato que cambia no se edita: se modela con un rango de vigencia (D6) o con una nueva situación que `rectifica` o `cancela` a la anterior. La historia completa queda en el grafo, y el método `at=` de las consultas reconstruye cualquier momento del pasado.

El método `add_individual` protege una invariante sutil pero crucial: un mismo identificador no puede vivir en dos ejes distintos. Si alguien intenta registrar `sede_central` como lugar y luego como agente, el universo aborta. Es la garantía de que cada cosa ocupa un lugar único en el espacio de coordenadas. Los métodos `facts_about`, `facts_with_role` y `facts_with_value` son las tres puertas de recuperación, todas con un parámetro `at` opcional que filtra por vigencia temporal. Y `summary` ofrece un censo rápido: cuántos individuos por eje, cuántos hechos en total.

EN MEMORIA

El universo no persiste a disco: es una estructura en memoria, suficiente para tests y demos. La persistencia (y los motores que la harían escalar) se discute en el capítulo 30.

PYTHON

```
"""El universo V — unión disjunta de los ejes de valor.

`Universe` es el almacenamiento en memoria del prototipo: una lista de
individuos y una lista de hechos, con índices para consulta eficiente.

Diseño:
- Se accede vía `add_individual`, `assert_fact`, `query`.
- La validación de signatura ocurre al insertar el hecho (delegada al Catalog).
- Los hechos son inmutables; "cambios" se modelan como nuevas situaciones
  o como rangos de vigencia (D6).

No persiste a disco — el prototipo es en memoria. Para tests y demos basta.
"""

from __future__ import annotations
from dataclasses import dataclass, field
from datetime import datetime
from typing import Dict, List, Optional, Iterator, Tuple, Any

from .axes import Axis
from .individual import Individual
from .fact import Fact


@dataclass
class Universe:
    name: str = "default"
    individuals: Dict[str, Individual] = field(default_factory=dict)
    facts: List[Fact] = field(default_factory=list)
    catalog: Any = None  # Catalog inyectado opcionalmente para validación

    # Índices
```

```
    _by_subject: Dict[str, List[int]] = field(default_factory=dict)
    _by_value: Dict[str, List[int]] = field(default_factory=dict)
    _by_role: Dict[str, List[int]] = field(default_factory=dict)

    # --- registro de individuos ------------------------------------------

    def add_individual(self, ind: Individual) -> Individual:
        existing = self.individuals.get(ind.id)
        if existing is not None:
            if existing.axis != ind.axis:
                raise ValueError(
                    f"Conflicto de eje para {ind.id}: ya existe en {existing.axis} "
                    f"y se intenta registrar en {ind.axis}."
                )
            return existing
        self.individuals[ind.id] = ind
        return ind

    def ind(self, id_or_label: str) -> Individual:
        """Recupera un individuo por id."""
        if id_or_label in self.individuals:
            return self.individuals[id_or_label]
        raise KeyError(f"Individuo no registrado: {id_or_label}")

    # --- afirmación de hechos --------------------------------------------

    def assert_fact(
        self,
        subject: Individual,
        role: str,
        value: Individual,
        valid_from: Optional[datetime] = None,
        valid_to: Optional[datetime] = None,
    ) -> Fact:
        """Afirma un hecho atómico. Valida la signatura si hay catálogo."""
        self.add_individual(subject)
        self.add_individual(value)

        if self.catalog is not None:
            self.catalog.validate(role, subject, value)

        fact = Fact(
            subject=subject,
            role=role,
            value=value,
            valid_from=valid_from,
            valid_to=valid_to,
        )
        idx = len(self.facts)
        self.facts.append(fact)
        self._by_subject.setdefault(subject.id, []).append(idx)
        self._by_value.setdefault(value.id, []).append(idx)
        self._by_role.setdefault(role, []).append(idx)
        return fact

    # --- recuperación ----------------------------------------------------

    def facts_about(self, individual: Individual,
                    at: Optional[datetime] = None) -> List[Fact]:
        """Todos los hechos donde `individual` es el sujeto."""
        idxs = self._by_subject.get(individual.id, [])
        result = [self.facts[i] for i in idxs]
        if at is not None:
            result = [f for f in result if f.is_valid_at(at)]
```

```
        return result

    def facts_with_role(self, role: str,
                        at: Optional[datetime] = None) -> List[Fact]:
        idxs = self._by_role.get(role, [])
        result = [self.facts[i] for i in idxs]
        if at is not None:
            result = [f for f in result if f.is_valid_at(at)]
        return result

    def facts_with_value(self, individual: Individual,
                         at: Optional[datetime] = None) -> List[Fact]:
        idxs = self._by_value.get(individual.id, [])
        result = [self.facts[i] for i in idxs]
        if at is not None:
            result = [f for f in result if f.is_valid_at(at)]
        return result

    # --- utilidades ----------------------------------------------------

    def __len__(self) -> int:
        return len(self.facts)

    def summary(self) -> str:
        n_by_axis: Dict[Axis, int] = {}
        for ind in self.individuals.values():
            n_by_axis[ind.axis] = n_by_axis.get(ind.axis, 0) + 1
        parts = [f"Universe '{self.name}'",
                 f"  individuos: {len(self.individuals)}"]
        for ax in Axis:
            if ax in n_by_axis:
                parts.append(f"    {ax.value}: {n_by_axis[ax]}")
        parts.append(f"  hechos:     {len(self.facts)}")
        return "\n".join(parts)
```

§A7 `ingest.py` — el pipeline de ingesta

Noventa y seis líneas. Esta es la función que más se usa en los ejemplos de la Parte V: `ingest_situation`. Toma un verbo y un diccionario de roles ya identificados, y aplica un pipeline de seis pasos que produce una situación reificada (**D4**) con todos sus hechos atómicos colgando de ella. El parser lingüístico se asume externo: lo hace un LLM, o un parser dedicado. La ingesta no entiende español; *modela* lo que alguien ya entendió.

> **D4 REIFICAR LA SITUACIÓN**
>
> El paso clave del pipeline es acuñar un individuo nuevo en O que *representa la situación entera* (la venta, la consulta, el préstamo) y colgar de él cada rol como un hecho atómico. Un evento deja de ser una fila con columnas y pasa a ser un nodo con cables. Eso es reificación, y es lo que permite que una situación tenga, a su vez, modalidad, causa o vigencia propias.

El pipeline se lee casi como su docstring: resuelve el lexicon (con polisemia o forma nominal), reifica la situación en O, asienta su `instancia_de` apuntando al tipo, recorre los roles y asienta cada uno como un hecho, agrega los extras (modalidad, estatus factual, calificación) y, antes de devolver, verifica que no falte ningún rol obligatorio declarado por la entrada del lexicon. El valor de retorno es la situación reificada, para que quien llama pueda seguir enriqueciéndola.

Nótese la coherencia con la política liberal del catálogo: la ingesta *no exige* que cada rol pasado esté declarado en la entrada del lexicon. La entrada es informativa, no una camisa de fuerza; los roles extra se admiten. Lo único que se verifica con dureza es la presencia de los obligatorios (sin agente no hay venta) y, vía `assert_fact`, la signatura de cada hecho contra el catálogo.

SEPARACIÓN

Que el parser sea externo no es una carencia, es un principio de diseño: separa «entender la oración» (lingüística) de «modelar el hecho» (arquitectura). Así el mismo motor sirve para cualquier idioma y cualquier parser, sea un LLM o un analizador hecho a mano.

PYTHON

```
"""Ingesta: traduce una "oración" (estructurada) a hechos en el universo.

Para el prototipo no parsear español real — eso es trabajo del LLM.
En su lugar exponemos una API de ingesta que recibe verbo + roles ya
identificados y aplica el pipeline:
    1. Resolver lexicon (con polisemia / forma nominal)
    2. Reificar la situación en O
    3. Asentar instancia_de = tipo_situacion
    4. Asentar cada rol como hecho atómico
    5. Validar contra el catálogo D8
    6. Verificar obligatorios

Devuelve la situación reificada para que el caller pueda enriquecerla.
"""

from __future__ import annotations
from datetime import datetime
from typing import Dict, Optional, Any, List

from .axes import Axis
from .individual import Individual, mint_id, category
from .universe import Universe
from .lexicon import Lexicon, LexiconEntry


class IngestError(ValueError):
    pass


def ingest_situation(
    universe: Universe,
    lexicon: Lexicon,
    verb: str,
    roles: Dict[str, Individual],
    *,
    complements: Optional[List[str]] = None,
    nominal: bool = False,
    valid_from: Optional[datetime] = None,
    valid_to: Optional[datetime] = None,
    extra: Optional[Dict[str, Individual]] = None,
    sit_id: Optional[str] = None,
) -> Individual:
    """Ingesta una situación a partir de un verbo y sus roles.

    `roles` mapea NOMBRE_CANÓNICO → Individual (el caller ya resolvió aliases).
    `complements` lista patrones de complemento que disparan polisemia
    (p. ej. ['la_mano'] para `dar la mano`).
    `nominal=True` indica que el "verbo" es en realidad una forma nominal
    ("llegada") y debe buscarse vía `resolve_nominal`.
    `extra` agrega hechos adicionales que no están en la signatura (p. ej.
    `modalidad`, `estatus_factual`, `calificacion`, ...).

    Devuelve la situación reificada en O.
    """
    entry = (
        lexicon.resolve_nominal(verb, complements)
```

```
        if nominal else
        lexicon.resolve(verb, complements)
    )
    if entry is None:
        raise IngestError(
            f"Lexicon no tiene entrada para '{verb}' "
            f"con complementos {complements}."
        )

    # Verificar obligatorios
    missing = [r for r in entry.obligatory if r not in roles]
    if missing:
        raise IngestError(
            f"Faltan roles obligatorios para '{verb}': {missing}"
        )

    # Reificar la situación
    sid = sit_id or mint_id(entry.situation_type)
    situ = Individual(id=sid, axis=Axis.O, label=sid)
    universe.add_individual(situ)

    # instancia_de
    tipo = category(entry.situation_type)
    universe.assert_fact(situ, "instancia_de", tipo,
                         valid_from=valid_from, valid_to=valid_to)

    # Roles obligatorios y opcionales
    for role, value in roles.items():
        # No exigimos que esté declarado en la entrada — la entrada es
        # informativa; los extras se admiten. (Política liberal.)
        universe.assert_fact(situ, role, value,
                             valid_from=valid_from, valid_to=valid_to)

    # Extras (modalidad, calificación, ...)
    if extra:
        for role, value in extra.items():
            universe.assert_fact(situ, role, value,
                                 valid_from=valid_from, valid_to=valid_to)

    return situ
```

`query.py` — el motor de consulta

Ciento veinticuatro líneas. Aquí las preguntas-WH se vuelven operaciones concretas. Una consulta es un `Pattern`: un diccionario de roles `fixed` con valores conocidos (lo que ya sabemos) y uno o más roles en `ask` marcados con un `Var` (lo que queremos descubrir). «¿Quién vendió el circuito de hidroterapia?» se escribe fijando el rol `tema` en el circuito y pidiendo el rol `agente`. El motor busca todas las situaciones que satisfacen los roles fijos y proyecta el valor del rol preguntado.

> **LA PREGUNTA COMO PROYECCIÓN**
>
> Preguntar es fijar algunas coordenadas y dejar otra libre. ¿Quién deja libre el rol con valor en Q, el de ¿Cuándo, el de ¿Cuándo, el de N. El motor recorre las situaciones que casan con lo fijo y devuelve lo que ocupa la coordenada libre. La geometría del libro se vuelve aquí un bucle.

El algoritmo tiene dos fases. Primero elige un *ancla* para no recorrer todo el universo: si el patrón restringe por tipo (`type_constraint`, un filtro por `instancia_de`), parte de ahí; si no, toma el primer rol fijo como punto de entrada y usa el índice correspondiente. Después filtra las candidatas verificando que cada una cumpla *todos* los roles fijos, comprueba el tipo si se pidió, y

extrae los valores de los roles preguntados. Si un rol preguntado tiene varios valores, devuelve la lista; si tiene uno, el individuo suelto.

Toda consulta admite el parámetro `at`: el momento en que se quiere evaluar la vigencia. Es lo que permite preguntar «¿quién era el dueño en aquella fecha?» y obtener la respuesta correcta para *ese* instante, no para el presente (la consulta bitemporal de **D6**). Y `count` es simplemente `query` que devuelve la cardinalidad en vez de los bindings: el caso de «¿cuántos?».

> **ALCANCE**
> Este motor es deliberadamente ingenuo: un barrido con tres índices simples. Sirve para validar el modelo en dominios reales, no para producción a escala. El capítulo 30 nombra los reemplazos serios: Datalog, SHACL, motores RDF[8], bases columnares.

PYTHON

```
"""Motor de consulta: las preguntas-WH como proyecciones.

Una consulta es un `Pattern`: un diccionario de roles fijos con valores
conocidos, y al menos un rol marcado como `Var(...)` (la pregunta).
El motor busca todas las situaciones del universo que satisfacen los
roles fijos y proyecta el valor del rol pregunta.

Soporta:
- Consultas puntuales: ¿quién vendió X?
- Consultas temporales: ¿quién era el dueño de X en T0?  (D6)
- Filtros por tipo (instancia_de = K).
- Agregaciones: count, list.
"""

from __future__ import annotations
from dataclasses import dataclass, field
from datetime import datetime
from typing import Dict, List, Optional, Any

from .individual import Individual
from .universe import Universe


@dataclass
class Var:
    """Marcador de variable en un patrón de consulta."""
    name: str = "?"


@dataclass
class Pattern:
    """Patrón de consulta sobre una situación.

    `fixed`: roles cuyo valor está dado (Individual).
    `ask`:   uno o más roles cuyo valor queremos descubrir (Var).
    """
    fixed: Dict[str, Individual] = field(default_factory=dict)
    ask: Dict[str, Var] = field(default_factory=dict)
    type_constraint: Optional[Individual] = None  # filtra por instancia_de = K

    def __post_init__(self):
        # `ask` vacío es válido: cuenta/lista candidatos sin proyección.
        # En ese caso el binding contiene solo `_subject`.
        pass


def query(universe: Universe, pattern: Pattern,
          at: Optional[datetime] = None) -> List[Dict[str, Any]]:
```

```
    """Ejecuta el patrón contra el universo. Devuelve una lista de bindings.

    Cada binding es un dict con las claves de `pattern.ask` y los valores
    encontrados (Individual). Las situaciones-candidatas son los sujetos
    que tienen *todos* los roles fijos del patrón (con sus valores) y, si
    aplica, instancia_de = type_constraint.
    """
    # Punto 1: buscar candidatas — sujetos en O que tienen todos los roles
    # del patrón. Tomamos el primer rol fijo (o type_constraint) como ancla
    # para reducir el espacio.

    if pattern.type_constraint is not None:
        # Sujetos cuya instancia_de == type_constraint
        candidate_subjects = {
            f.subject.id
            for f in universe.facts_with_role("instancia_de", at=at)
            if f.value.id == pattern.type_constraint.id
        }
    elif pattern.fixed:
        # Tomamos el primer fixed como ancla.
        role0, val0 = next(iter(pattern.fixed.items()))
        candidate_subjects = {
            f.subject.id
            for f in universe.facts_with_role(role0, at=at)
            if f.value.id == val0.id
        }
    else:
        # Si solo hay ask, buscamos sobre todas las situaciones (raro).
        candidate_subjects = set(universe.individuals.keys())

    # Punto 2: filtrar candidatas por todos los roles fijos
    results: List[Dict[str, Any]] = []
    for sid in candidate_subjects:
        subject = universe.individuals[sid]
        sit_facts = universe.facts_about(subject, at=at)

        # Map role → list of values for this subject
        roles_map: Dict[str, List[Individual]] = {}
        for f in sit_facts:
            roles_map.setdefault(f.role, []).append(f.value)

        # Chequear roles fijos
        ok = True
        for role, expected_val in pattern.fixed.items():
            vals = roles_map.get(role, [])
            if not any(v.id == expected_val.id for v in vals):
                ok = False
                break
        if not ok:
            continue

        # Chequear type_constraint si está
        if pattern.type_constraint is not None:
            instancia_vals = roles_map.get("instancia_de", [])
            if not any(v.id == pattern.type_constraint.id for v in instancia_vals):
                continue

        # Extraer valores para los roles preguntados
        binding: Dict[str, Any] = {"_subject": subject}
        all_present = True
        for ask_role in pattern.ask:
            vals = roles_map.get(ask_role, [])
            if not vals:
                all_present = False
```

```
                break
            # Si hay más de uno, devolvemos lista; si uno, el individuo.
            binding[ask_role] = vals[0] if len(vals) == 1 else list(vals)
        if all_present:
            results.append(binding)
    return results


def count(universe: Universe, pattern: Pattern,
          at: Optional[datetime] = None) -> int:
    """Cuenta sujetos que satisfacen el patrón."""
    return len(query(universe, pattern, at=at))
```

§A9 `__init__.py` — la superficie pública

Treinta y dos líneas. Es lo último que ve quien importa la librería y lo primero que toca al escribir `from wq import *`. Reexporta cada clase y función que el usuario va a usar, con un docstring que recapitula el modelo entero en una pantalla. Esta fachada es el contrato del paquete: si un nombre está aquí, es público y estable; si no, es interno.

Vale la pena leer el `__all__` como un índice del propio modelo. Están los `Axis`, los constructores de individuos (`category`, `quantity`, `time_point`), el `Fact` y el `Universe`, el `Catalog` con sus signaturas, el `Lexicon`, el motor de consulta (`Pattern`, `Var`, `query`, `count`) y la ingesta. Nada más. Toda la potencia del prototipo cabe en estos diecisiete nombres.

PYTHON

```
"""WQuestions — prototipo de validación en Python.

Implementa el modelo descrito en `WQuestions.md` y discutido en el libro:
- 7 ejes (Q, O, L, T, N, K, M)
- hechos atómicos con signatura tipada
- situaciones reificadas (D4, D5)
- catálogo canónico de roles (D8)
- lexicon con resolución de polisemia (D9)
- vigencia temporal por reificación (D6)
- consultas como proyecciones sobre roles (preguntas-WH)

Se prioriza claridad sobre rendimiento. La meta es validar la arquitectura,
no servir producción.
"""

from .axes import Axis
from .individual import Individual, mint_id, category, quantity, time_point
from .fact import Fact
from .universe import Universe
from .catalog import Catalog, RoleSignature, SignatureError
from .lexicon import Lexicon, LexiconEntry
from .query import Pattern, Var, query, count
from .ingest import ingest_situation, IngestError

__all__ = [
    "Axis", "Individual", "mint_id", "category", "quantity", "time_point",
    "Fact", "Universe",
    "Catalog", "RoleSignature", "SignatureError",
    "Lexicon", "LexiconEntry",
    "Pattern", "Var", "query", "count",
    "ingest_situation", "IngestError",
]
```

Seis observaciones sobre el código completo

Visto de una sola pasada, de `axes.py` a `__init__.py`, el código deja seis impresiones que vale la pena nombrar. No son conclusiones nuevas: son la confirmación, en líneas de Python, de lo que el libro defiende en prosa.

1 POCO DE TODO

Nueve archivos, ~850 líneas, una clase principal por archivo. Sin infraestructura inútil ni abstracciones especulativas. Cada pieza existe porque un capítulo entero la justifica.

2 LAS DECISIONES SE VEN

D3 vive en `fact.py`, D4 en `ingest.py`, D6 en `is_valid_at`, D8 en `catalog.py`, D9 en `lexicon.py`. Una decisión, un módulo: el paralelismo es directo.

3 LA POLÍTICA LIBERAL ES EXPLÍCITA

En `catalog.validate`, un rol no declarado no se valida. Cuatro líneas que materializan la decisión de mantener el modelo extensible sin pelear con el catálogo central.

4 LA BITEMPORALIDAD ES LIGERA

No hay un sistema bitemporal completo. Hay `valid_from`/`valid_to` opcionales, un `tx_time` automático y un parámetro `at=` en las consultas. Suficiente para mudanzas, rediagnósticos y cláusulas que expiran.

5 NO HAY PARSER DE LENGUAJE

El prototipo asume el parser externo (un LLM o un módulo dedicado) que se enchufa por `ingest_situation`. Separar «entender» de «modelar» es lo que lo hace reutilizable en cualquier idioma.

6 EL MOTOR ES INGENUO

Un barrido con tres índices simples. Valida el modelo en dominios reales; no escala a producción. Los reemplazos serios (Datalog, SHACL, RDF, columnar) están en el capítulo 30.

Juntas, estas seis observaciones son la justificación del enfoque: **no construir infraestructura, sino prueba de concepto**. Lo que estas ochocientas cincuenta líneas demuestran es que el modelo es coherente y operable. La validación industrial es otro trabajo: el del capítulo 30 y de quien lo continúe.

Lo que queda en el repositorio

Para que este anexo siga siendo legible, incluí solo la librería núcleo. El resto del prototipo (alrededor de **2.400 líneas adicionales**) está publicado completo en el repositorio del proyecto. Este es el mapa de lo que encontrarás allí.

TEXT

```
github.com/joseabantomarin/WQuestions
└── prototipo/
    ├── wq/                      ← incluida en este anexo (850 líneas)
    │   ├── axes.py
    │   ├── individual.py
    │   ├── fact.py
    │   ├── catalog.py
    │   ├── lexicon.py
    │   ├── universe.py
    │   ├── ingest.py
    │   ├── query.py
    │   └── __init__.py
    │
    ├── ejemplos/                ← no incluida — está en el repo
    │   ├── spa.py               (532 líneas)
```

```
    │   ├── dominios_previos.py   (522 líneas — receta, gol, canción, noticia)
    │   ├── banco.py              (465 líneas)
    │   ├── clinica.py            (312 líneas)
    │   └── taxi.py               (256 líneas)
    │
    └── tests/                    ← no incluida — está en el repo
        └── test_wq.py            (349 líneas)
```

Cada archivo de `ejemplos/` es un script ejecutable que modela un dominio completo: declara los individuos, registra el lexicon, ingresa todas las situaciones, corre las consultas que el libro discute y valida los resultados. Ejecutar uno cualquiera (por ejemplo `python -m prototipo.ejemplos.banco`) reproduce las escenas de la Parte V tal como aparecen en el texto.

El archivo `test_wq.py` ejercita las invariantes del modelo: que el catálogo valide signaturas, que las consultas bitemporales devuelvan lo correcto en distintos momentos, que la polisemia del lexicon elija la entrada más específica, que la reificación produzca el grafo esperado. Es el cinturón de seguridad detrás de la promesa del libro de que «todos los tests pasan».

CORRER EL PROTOTIPO EN CINCO MINUTOS

BASH

```bash
git clone https://github.com/joseabantomarin/WQuestions.git
cd WQuestions
python -m prototipo.tests.test_wq          # corre la batería de tests
python -m prototipo.ejemplos.spa           # ejecuta el dominio del Spa
python -m prototipo.ejemplos.banco         # ejecuta el dominio bancario
```

No hace falta instalar dependencias: el prototipo solo usa la librería estándar de Python. Los identificadores y las APIs (`Universe`, `ingest_situation`, `Pattern`, `query`) son los mismos que leíste aquí, así que pasar de leer a ejecutar es inmediato.

> *El prototipo, los ejemplos y los tests son el espejo operable del libro: lo que se afirma en cualquier capítulo de la Parte V se puede ejecutar línea por línea.*

DEL TEXTO A LA INFRAESTRUCTURA

Si encuentras una afirmación del libro que no puedas reproducir, el repositorio tiene un *issue tracker* abierto. Es exactamente ese tipo de retroalimentación la que el capítulo 30 reclama para que la propuesta deje de ser un texto y empiece a ser infraestructura de uso diario.

Clónalo, córrelo, rómpelo.

34
El autor

Detrás de toda arquitectura hay una biografía. Esta es, en pocas líneas, la de quien pasó tres décadas buscando la capa donde la información deja de depender de la aplicación que la consulta.

José Abanto Marín

José Abanto Marín nació en Celendín, en la sierra norte del Perú, en 1973. Lleva cerca de treinta y cinco años construyendo sistemas a la medida (el oficio paciente de traducir un negocio en estructuras que una máquina pueda sostener) y de esa práctica larga, más que de cualquier teoría previa, nace este libro.

Programó desde los tiempos en que las pantallas eran de color verde o ámbar y un vetusto intérprete de BASIC obraba el pequeño milagro de obedecer las órdenes de quien lo escribía. De allí pasó a las bases de datos (dBase primero, FoxPro después) y en ese tránsito quedó sellada una pasión que ya no lo abandonaría. Cuando conoció Clipper intuyó el potencial tremendo de un compilador y, antes que resignarse a recompilar a cada paso, construyó un solo núcleo: un sistema al que llamó **xsystem** y bautizó *generador de programas*, porque en él empezaba ya a separar la lógica, llevando las reglas del negocio a la base de datos. Delphi fue el siguiente gran salto, y con él levantó la primera encarnación de lo que sería su obra mayor.

Es el creador de **Ghenesis**, un *framework* de orientación *meta-driven* cuya idea rectora es sencilla de enunciar y difícil de llevar hasta el final: persistir la lógica de negocio en la propia base de datos. Formularios, reportes, módulos enteros viven allí como datos, no como código, de modo que pueden administrarse y reconfigurarse sin volver a tocar la aplicación. Esa obsesión (separar el *qué* del *cómo*, encontrar el estrato donde el conocimiento se vuelve independiente del programa que lo interroga) recorre toda su trayectoria y desemboca, casi por su propio peso, en las páginas anteriores.

Hoy tiene programada una versión web de Ghenesis en la que esa abstracción llega a su punto más alto: los campos, los reportes, las validaciones e incluso las APIs se consumen sin necesidad de compilar una sola línea. El código deja de ser el lugar donde el sistema se decide; pasa a serlo el dato que lo describe.

> *Toda mi vida profesional ha sido la misma pregunta repetida en dominios distintos: ¿dónde está la capa en la que el dato deja de pertenecerle a la aplicación?*
>
> JOSÉ ABANTO MARÍN

Apasionado de las bases de datos, de los modelos de inteligencia artificial y de la ciencia en general, encuentra en la física cuántica y en los enigmas del universo algo más que un pasatiempo: la confirmación de una sospecha que también gobierna su trabajo, la de que aquello que parece irreductiblemente complejo esconde, en algún nivel, una estructura más simple. La filosofía, lejos de ser ornamento, es para él una herramienta de taller: la disciplina con que se reconocen los patrones, las abstracciones y los paradigmas que se asoman, una y otra vez, en cada problema técnico.

UNA LÍNEA

Tres décadas de práctica, una curiosidad científica que nunca se apagó y el hábito filosófico de mirar bajo la superficie: las tres vertientes confluyen en una sola tesis.

Estas tres líneas (la práctica de tres décadas modelando datos, la curiosidad científica y la mirada filosófica) transcurrieron durante años en cauces paralelos. *WQuestions, Las preguntas como coordenadas* es el primer libro en el que el autor las reúne deliberadamente alrededor de una única idea arquitectónica: que las preguntas que cualquier niño domina antes de aprender a leer bastan para organizar, con precisión técnica, la información de cualquier dominio del mundo.

UNA SOLA OBSESIÓN, MUCHOS DISFRACES

Si algo une a Ghenesis con WQuestions no es una tecnología, sino una convicción: que existe un nivel anterior a toda aplicación donde la información se describe a sí misma. Encontrar ese nivel, y volverlo infraestructura utilizable, ha sido el trabajo de una vida.

Gracias por haber llegado hasta aquí.

APARATO CRÍTICO

Referencias

Las fuentes y precedentes citados a lo largo del libro. En el texto, cada referencia se marca con su número entre paréntesis —por ejemplo, (4) remite a CIDOC CRM—. La numeración es estable y se asigna por orden de primera aparición.

Filosofía y retórica clásica

1 Aristóteles. *Ética a Nicómaco*, libro III. Discusión de las circunstancias del acto voluntario como criterio para evaluar la responsabilidad moral.

2 Cicerón. *De inventione*, libro I (siglo I a.C.). Lista canónica de las *circumstantiae* romanas: *quis, quid, ubi, quibus auxiliis, cur, quomodo, quando*.

25 Quintiliano. *Institutio Oratoria*, siglo I d.C. Refinamiento retórico del esquema de las circunstancias heredado de Cicerón y Hermágoras.

26 Tomás de Aquino. *Summa Theologica*, I-II, q. 7 (siglo XIII). Retoma las circunstancias del acto como categorías para evaluar su moralidad.

Periodismo

3 Bleyer, W. G. *Newspaper Writing and Editing*. Houghton Mifflin, 1913. Texto fundacional que codifica los 5W1H como obligación de toda nota informativa.

Ontologías de dominio

4 **CIDOC CRM** — *Conceptual Reference Model* del International Council of Museums. Norma ISO 21127:2014. cidoc-crm.org

5 **Biolink Model** — Modelo unificador de tipos y predicados para datos biomédicos. biolink.github.io/biolink-model

30 **Schema.org** — Vocabulario estandarizado para datos estructurados en la web. Iniciativa conjunta de Google, Microsoft, Yandex y Yahoo. schema.org

19 **IFC** — *Industry Foundation Classes*. ISO 16739. Estándar de datos para arquitectura, ingeniería y construcción. buildingsmart.org

Estándares de intercambio

6 **HL7 FHIR** — *Fast Healthcare Interoperability Resources*. Estándar de intercambio en salud. hl7.org/fhir

7 **XBRL** — *eXtensible Business Reporting Language*. Estándar para reportes financieros corporativos. xbrl.org

20 **EDI** — *Electronic Data Interchange*. Familia de estándares (X12, EDIFACT) para intercambio comercial estructurado.

21 **ISO 20022** — Norma global para mensajes financieros (pagos, valores, comercio exterior).

Web semántica y grafos abiertos

8 **RDF** — *Resource Description Framework*. Recomendación W3C. w3.org/RDF

22 **OWL** — *Web Ontology Language*. Recomendación W3C. w3.org/OWL

23 **OpenIE** — Banko, M., Cafarella, M. J., Soderland, S., Broadhead, M. & Etzioni, O. «Open Information Extraction from the Web». *IJCAI*, 2007. Origen de las técnicas de extracción de relaciones abierta.

31 Berners-Lee, T., Hendler, J. & Lassila, O. «The Semantic Web». *Scientific American*, mayo de 2001. Artículo programático que formuló la visión de la web semántica.

32 **Wikidata** — Base de conocimiento colaborativa de la Fundación Wikimedia, sobre RDF. wikidata.org

33 **DBpedia** — Extracción estructurada de información de Wikipedia. dbpedia.org

5W1H operativo en computación

9 Yang, X. & Hu, J. Trabajos sobre extracción de información basada en 5W1H aplicados a representación de eventos.

10 Mahmood, A. Representación de eventos sobre el esqueleto 5W1H.

Semántica formal y representación del conocimiento

11 Barwise, J. & Perry, J. *Situations and Attitudes*. MIT Press, 1983. Teoría matemática de la información situacional.

12 Davidson, D. «The Logical Form of Action Sentences». En Rescher, N. (ed.), *The Logic of Decision and Action*. University of Pittsburgh Press, 1967. Origen de la semántica neo-davidsoniana de eventos.

13 Gärdenfors, P. *Conceptual Spaces: The Geometry of Thought*. MIT Press, 2000.

24 Fillmore, C. J. «The Case for Case». En Bach, E. & Harms, R. T. (eds.), *Universals in Linguistic Theory*. Holt, Rinehart and Winston, 1968. Origen de la teoría de roles temáticos.

Recursos léxicos

14 **FrameNet** — Proyecto del International Computer Science Institute, Universidad de California, Berkeley. framenet.icsi.berkeley.edu

15 **VerbNet** — Léxico de verbos del inglés organizado por clases de Levin. Universidad de Colorado, Boulder. verbs.colorado.edu/verbnet

Extracción y modelado de relaciones

16 Riedel, S., Yao, L., McCallum, A. & Marlin, B. «Relation Extraction with Matrix Factorization and Universal Schemas». *NAACL HLT*, 2013.

Psicolingüística y desarrollo

27 Brown, R. *A First Language: The Early Stages*. Harvard University Press, 1973. Estudio longitudinal seminal sobre la adquisición del lenguaje en niños.

Universales lingüísticos

28 Wierzbicka, A. *Semantic Primitives* (1972) y *Semantics: Primes and Universals* (Oxford University Press, 1996). Programa del Natural Semantic Metalanguage: identifica primitivos semánticos compartidos por todas las lenguas humanas.

29 Greenberg, J. H. (ed.). *Universals of Human Language* (4 vols.). Stanford University Press, 1978. Marco fundacional del estudio comparado de universales lingüísticos.

Bases de datos temporales

17 Snodgrass, R. T. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999. Fundamento del modelado bitemporal.

Unidades de medida

18 **QUDT** — *Quantities, Units, Dimensions and Types Ontology*. qudt.org

APARATO CRÍTICO

Las decisiones de diseño

*A lo largo del libro, cada vez que el modelo elige un camino y descarta otro, la elección queda fijada en una caja de **decisión de diseño**. Son nueve, de **D1** a **D9**, y juntas forman las reglas que distinguen al modelo de un grafo cualquiera. Aquí están reunidas en un solo lugar: el enunciado de cada una y el capítulo donde se argumenta.*

Los identificadores `D1` … `D9` son *estables*: nombran la regla, no el capítulo. El orden en que se numeran coincide con el orden en que aparecen al leer, pero el número de capítulo puede cambiar entre ediciones sin que el identificador de la regla se mueva. Por eso este índice traduce de uno a otro.

Las nueve reglas y el capítulo donde se introduce cada una.

REGLA	ENUNCIADO BREVE	CAPÍTULO
`D1`	**La plantilla y la instancia** Los tipos (las categorías) viven aparte de las cosas concretas que creas a partir de ellos.	03 · Cuál: el zócalo categórico
`D2`	**Predicados unificados** Propiedades y relaciones son lo mismo: cables tipados. Solo cambia si aceptan uno o varios destinos.	05 · Cómo: los predicados
`D3`	**El átomo de la información** Todo se guarda como tripletas (sujeto, cable, objeto); lo complejo se arma apilando tripletas.	07 · El hecho atómico
`D4`	**Cuándo reificar una situación** Un hecho se vuelve «situación» con identidad propia solo si lo necesita; si no, basta una tripleta.	09 · Situaciones, contextos y agencia
`D5`	**Agencia contextual** Quien actúa puede ser persona, empresa, software o sensor; lo decide el verbo.	09 · Situaciones, contextos y agencia
`D6`	**Vigencia temporal** Lo que cambia con el tiempo se guarda con fecha de inicio y fin; nunca se sobreescribe.	09 · Situaciones, contextos y agencia
`D7`	**El porqué se divide en cuatro cables** No hay eje «por qué»: se separa en causa, motivo, finalidad y justificación.	10 · El «por qué» no es un eje
`D8`	**El catálogo es invisible; el lexicon es la interfaz** Por dentro hay un catálogo exacto de roles, pero nadie lo toca de fuera: todo pasa por el lexicon.	14 · El lexicon como compilador
`D9`	**El usuario nunca toca etiquetas canónicas** El usuario escribe en su propio idioma; el lexicon traduce a las etiquetas oficiales en silencio.	14 · El lexicon como compilador

Parte II · Las siete coordenadas

D1 **La plantilla y la instancia.** En el eje **K** habitan exclusivamente los conceptos atemporales y categóricos (las plantillas). En el eje **O** (y en el resto de los pilares) habitan las entidades creadas, situadas geográficamente e instanciadas (los objetos derivados).

→ Capítulo 3 · Cuál: el zócalo categórico (K)

D2 **Predicados unificados.** Las propiedades y las relaciones se unifican bajo el mismo concepto: son simplemente cables (predicados) del eje *cómo* (M) con etiquetas de seguridad (signaturas tipadas). La única diferencia entre ellas es la cardinalidad —si aceptan uno o varios destinos simultáneos—, y esa cardinalidad es un atributo de la signatura de cada cable, no un eje aparte. Le indica a la base de datos cuándo debe «borrar y reemplazar» (cable funcional) y cuándo debe «acumular» (cable múltiple) nueva información.

→ Capítulo 5 · Cómo: los predicados (P y M)

Parte III · Cómo funcionan juntas

D3 **El átomo de la información.** Cualquier hecho de la realidad se representa siempre como una tripleta atómica de la forma `(sujeto, cable, objeto)`. El cable debe estar tipado (saber qué cajas conecta) para que el sistema pueda validar si el dato es lógico o absurdo. Las descripciones de cosas complejas jamás se construyen inventando estructuras nuevas ni tablas más anchas: se construyen apilando decenas de estas tripletas simples sobre un mismo sujeto.

→ Capítulo 7 · El hecho atómico

D4 **Cuándo reificar una situación.** Un evento o relación solo se transforma en una «situación reificada» (guardada en el eje O) si cumple al menos uno de cuatro requisitos: (1) tiene atributos propios como lugar o modo, (2) participan más de dos actores, (3) será referenciado por otros eventos en el futuro, o (4) su valor cambiará y necesitamos conservar un registro histórico. Si no cumple ninguna, usamos una tripleta simple.

→ Capítulo 9 · Situaciones, contextos y agencia

D5 **Agencia contextual.** El rol de `agente` puede ser ocupado por humanos, corporaciones, algoritmos de software o sensores físicos. Todo depende del verbo de la acción. Hay verbos que exigen un agente (como «vender») y otros que no necesitan a nadie («ocurrir», «llover»).

→ Capítulo 9 · Situaciones, contextos y agencia

D6 **Vigencia temporal.** Las propiedades que cambian con el paso del tiempo no se guardan directamente. Se reifican convirtiéndolas en situaciones, y se les añade una fecha de `inicio` y una de `fin` (su rango de vigencia).

→ Capítulo 9 · Situaciones, contextos y agencia

D7 **El porqué se divide en cuatro cables.** El modelo no tiene un eje «por qué». En su lugar, el sistema toma el «porque» del lenguaje natural y lo divide en cuatro cables distintos (relaciones canónicas) que conectan a las situaciones entre sí. Esos cables son: `causado_por` (para la física), `motivado_por` (para la intención humana), `con_finalidad` (para el propósito futuro) y `justificado_por` (para las reglas y leyes).

→ Capítulo 10 · El «por qué» no es un eje

Parte IV · Del lenguaje a los hechos

D8 **El catálogo es invisible; el lexicon es la interfaz.** El catálogo canónico de roles existe, es exacto y es la base sobre la que se valida cada hecho —pero nunca se expone directamente. La única superficie con la que el mundo exterior

interactúa es el lexicon. Por dentro, los ingenieros pueden renombrar un rol, fusionar dos o versionar el catálogo entero; por fuera, nada cambia, porque la cara visible del sistema es el diccionario, no el catálogo.

→ Capítulo 14 · El lexicon como compilador

D9 **El usuario nunca toca etiquetas canónicas.** Un humano jamás debe verse obligado a escribir `agente`, `beneficiario` o `experimentador` para que el sistema funcione. El usuario emplea su propio vocabulario («vendedor», «comprador», «cliente», «el que anota») y el lexicon lo traduce en silencio a la etiqueta canónica correspondiente. La complejidad del idioma se resuelve en el diccionario, no en la cabeza de quien escribe.

→ Capítulo 14 · El lexicon como compilador