

Bases de Datos a Gran Escala

Sesión 3 · Introducción a los sistemas NoSQL

Diego Sevilla Ruiz

Departamento de Ingeniería y Tecnología de Computadores

Facultad de Informática · Universidad de Murcia

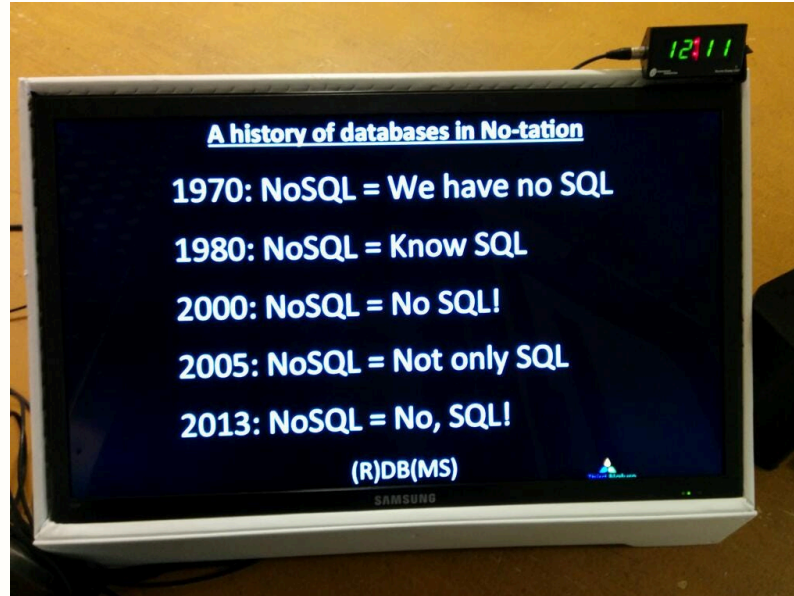
dsevilla@um.es

Introducción a los sistemas NoSQL

Por qué surgieron, qué tienen en común y cuánto se usan

Introducción a NoSQL. ¿Cómo surgió el nombre?

- **NoSQL** → *hashtag* llamativo elegido para una conferencia en 2009 (Johan Oskarsson, de Last.fm)
- Agrupa **modelos diferentes del relacional**, tradicionalmente con lenguajes de consulta propios
- Más recientemente, **NoSQL** → ***Not Only SQL*** (no sólo SQL) → persistencia políglota (*polyglot persistence*)



NoSQL



NoSQL: ¿por qué se plantearon?

1. Mayor escalabilidad horizontal

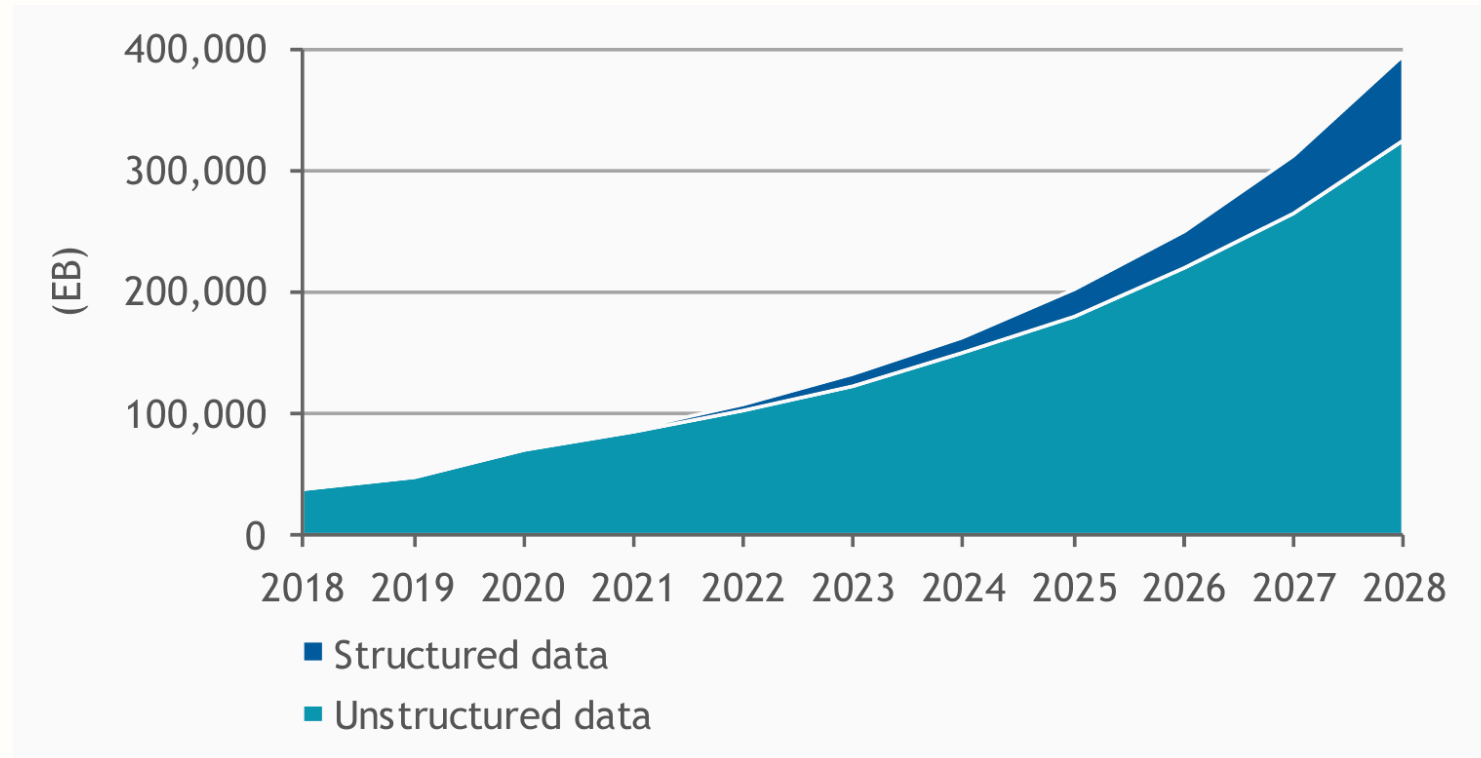
- Datos y peticiones que desbordan un único servidor
- Sistemas de alto volumen de escrituras (*streaming* de eventos, aplicaciones sociales)

2. Demanda de productos de software libre (crecimiento de las *start-ups*)

3. Patrones de acceso especializados: recuperar unidades completas sin recomponerlas con múltiples JOIN s

4. Expresividad, flexibilidad, dinamismo: frustración con la rigidez de las tablas normalizadas para ciertas aplicaciones

El volumen de datos sigue creciendo



Predominan los **no estructurados**; los estructurados crecen más en porcentaje

Fuente: IDC, septiembre de 2024 · figura 2

Volumen generado al año · Incluye semiestructurados · 2024–2028: previsión

NoSQL: características

Diferencias **tradicionales y generales** respecto de los SGBDR:

- Otros modelos y lenguajes de consulta, además del relacional y SQL
- Diseño orientado a **repartir los datos** y crecer horizontalmente
- En muchos sistemas, estructura flexible sin declarar todas las columnas
- Operaciones atómicas sobre una **clave o agregado**, evitando coordinar muchas entidades
- Accesos previstos influyen más en la forma de organizar los datos

Son tendencias de diseño: veremos excepciones y la convergencia posterior con SQL

Categorías de NoSQL

- Bases de datos *key-value*
- Bases de datos documentales
- Bases de datos columnares (*wide column*)
- Bases de datos de grafos
- Bases de datos de *arrays*

Evolución desde el modelo relacional

- El **modelo relacional** $\Rightarrow$ **predominante desde hace décadas**
- Tiene sus raíces en el denominado *business data processing*, procesamiento de transacciones y *batch*
- Propuesto por Codd en los 70, **de alto nivel**
- Actualmente los **sistemas SQL están muy optimizados**:
 - El **grado de implantación es mayoritario**
 - Para la gran mayoría de los problemas (que caben en un ordenador) es eficiente y adecuado

Adopción de NoSQL

Twitter adopta Cassandra hacia 2010. Cassandra, publicada por Facebook en 2008

The screenshot shows a web page from Computerworld. At the top, there's a navigation bar with the site logo, a 'Popular Now' dropdown, and social media icons. Below the navigation bar, a breadcrumb trail reads 'Home > Enterprise Applications > Database Management'. The main headline is 'Twitter growth prompts switch from MySQL to 'NoSQL' database'. Below the headline, a sub-headline reads 'Hopes open source technology can further boost improving Twitter uptime record'. The article is by Eric Lai, dated Feb 23, 2010. To the right of the article, there's a 'MORE LIKE THIS' section with three links: 'The end of SQL and relational databases? (part 3 of 3)', 'InfoWorld's top 10 emerging enterprise technologies', and 'InfoWorld review: Databases primed for social networks'. At the bottom left, there's a 'RELATED TOPICS' section with two tags: 'Database Management' and 'Enterprise Applications'.

TRENDING: What you need to know about Windows 10's new upgrade · First impressions: iPad Pro 'wicked fast' · Resources/White Papers

≡ **COMPUTERWORLD** Popular Now: [dropdown] [Twitter] [LinkedIn] [Facebook] [Google+] [RSS] [Search]

Home > Enterprise Applications > Database Management

NEWS

Twitter growth prompts switch from MySQL to 'NoSQL' database

Hopes open source technology can further boost improving Twitter uptime record

By Eric Lai

Computerworld | Feb 23, 2010 4:10 PM PT

RELATED TOPICS

- Database Management
- Enterprise Applications

Ryan King, an engineer at Twitter, today told the blog MyNoSQL that the social networking company plans to move from MySQL to the Cassandra database for what he called its resilience, scalability and large community of open-source developers.

MORE LIKE THIS

- The end of SQL and relational databases? (part 3 of 3)
- InfoWorld's top 10 emerging enterprise technologies
- InfoWorld review: Databases primed for social networks

Adopción NoSQL (septiembre 2020)

<https://db-engines.com/en/ranking/>

358 systems in ranking, September 2020

Rank			DBMS	Database Model	Score		
Sep 2020	Aug 2020	Sep 2019			Sep 2020	Aug 2020	Sep 2019
1.	1.	1.	Oracle +	Relational, Multi-model ⓘ	1369.36	+14.21	+22.71
2.	2.	2.	MySQL +	Relational, Multi-model ⓘ	1264.25	+2.67	-14.83
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model ⓘ	1062.76	-13.12	-22.30
4.	4.	4.	PostgreSQL +	Relational, Multi-model ⓘ	542.29	+5.52	+60.04
5.	5.	5.	MongoDB +	Document, Multi-model ⓘ	446.48	+2.92	+36.42
6.	6.	6.	IBM Db2 +	Relational, Multi-model ⓘ	161.24	-1.21	-10.32
7.	7.	↑ 8.	Redis +	Key-value, Multi-model ⓘ	151.86	-1.02	+9.95
8.	8.	↓ 7.	Elasticsearch +	Search engine, Multi-model ⓘ	150.50	-1.82	+1.23
9.	9.	↑ 11.	SQLite +	Relational	126.68	-0.14	+3.31
10.	↑ 11.	10.	Cassandra +	Wide column	119.18	-0.66	-4.22
11.	↓ 10.	↓ 9.	Microsoft Access	Relational	118.45	-1.41	-14.26
12.	12.	↑ 13.	MariaDB +	Relational, Multi-model ⓘ	91.61	+0.69	+5.54
13.	13.	↓ 12.	Splunk	Search engine	87.90	-2.01	+0.89
14.	14.	↑ 15.	Teradata +	Relational, Multi-model ⓘ	76.39	-0.39	-0.57
15.	15.	↓ 14.	Hive	Relational	71.17	-4.12	-11.93
16.	16.	↑ 18.	Amazon DynamoDB +	Multi-model ⓘ	66.18	+1.43	+8.36
17.	17.	↑ 25.	Microsoft Azure SQL Database	Relational, Multi-model ⓘ	60.45	+3.60	+32.91
18.	18.	↑ 19.	SAP Adaptive Server	Relational	54.01	+0.05	-2.09
19.	19.	↑ 21.	SAP HANA +	Relational, Multi-model ⓘ	52.86	-0.26	-2.53
20.	20.	↓ 16.	Solr	Search engine	51.62	-0.08	-7.35
21.	21.	↑ 22.	Neo4j +	Graph	50.63	+0.44	+2.41
22.	22.	↓ 20.	HBase +	Wide column	48.35	-0.76	-7.37

Adopción NoSQL: cómo leer estas imágenes

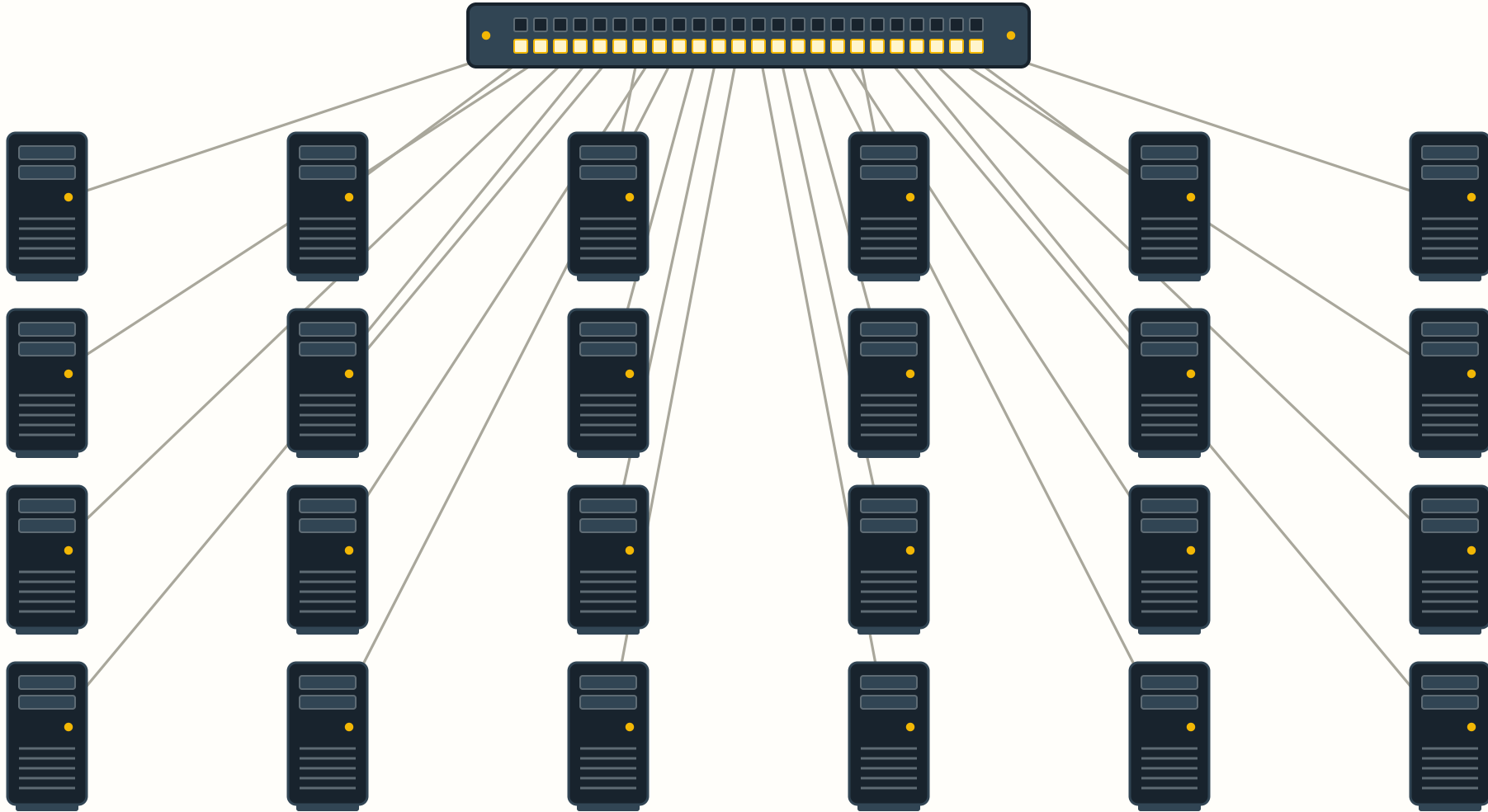
- La captura anterior corresponde a **septiembre de 2020**: es contexto histórico
- DB-Engines combina indicadores de **popularidad**, no mide cuota de instalaciones ni rendimiento
- Conviven grandes SGBDR y productos documentales, de clave-valor y de grafos
- La posición en un ranking no decide qué sistema resuelve mejor una aplicación

Fuente: **metodología de DB-Engines**

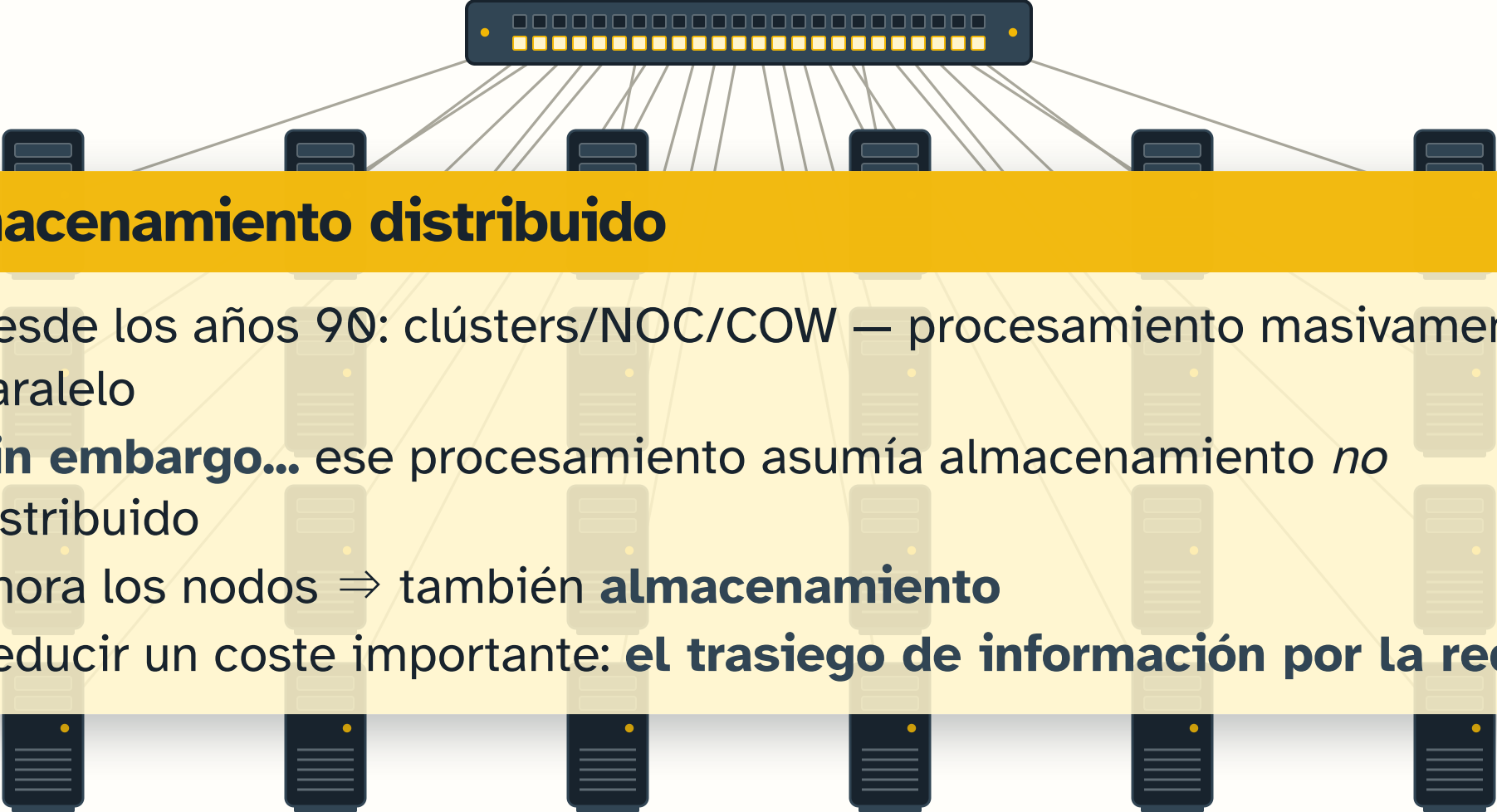
La importancia de la escalabilidad

Del servidor único a un gran array repartido y replicado entre nodos

Cambio de perspectiva: red



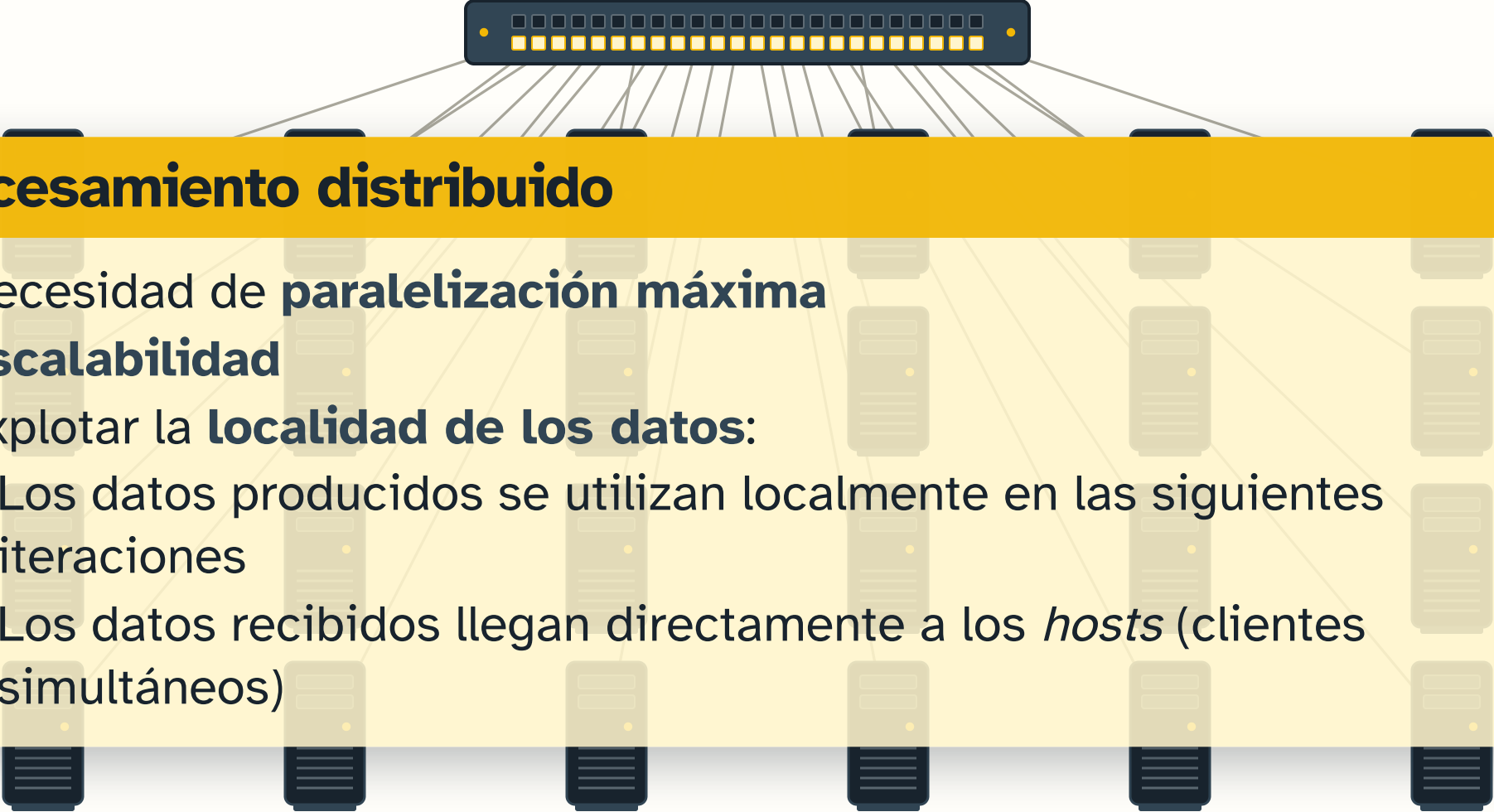
Cambio de perspectiva: red



Almacenamiento distribuido

- Desde los años 90: clústers/NOC/COW — procesamiento masivamente paralelo
- **Sin embargo...** ese procesamiento asumía almacenamiento *no* distribuido
- Ahora los nodos $\Rightarrow$ también **almacenamiento**
- Reducir un coste importante: **el trasiego de información por la red**

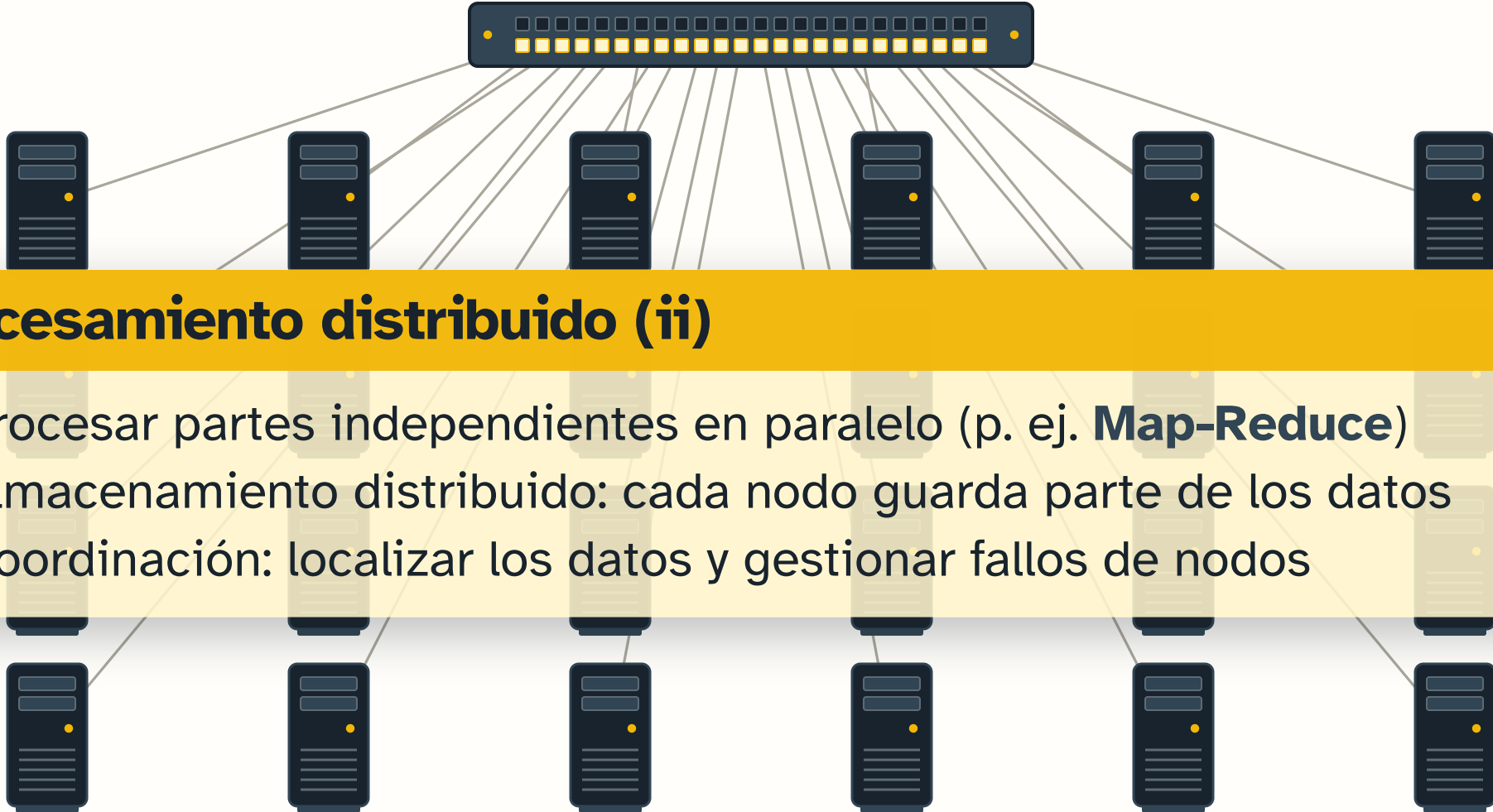
Cambio de perspectiva: red



Procesamiento distribuido

- Necesidad de **paralelización máxima**
- **Escalabilidad**
- Explotar la **localidad de los datos**:
 - Los datos producidos se utilizan localmente en las siguientes iteraciones
 - Los datos recibidos llegan directamente a los *hosts* (clientes simultáneos)

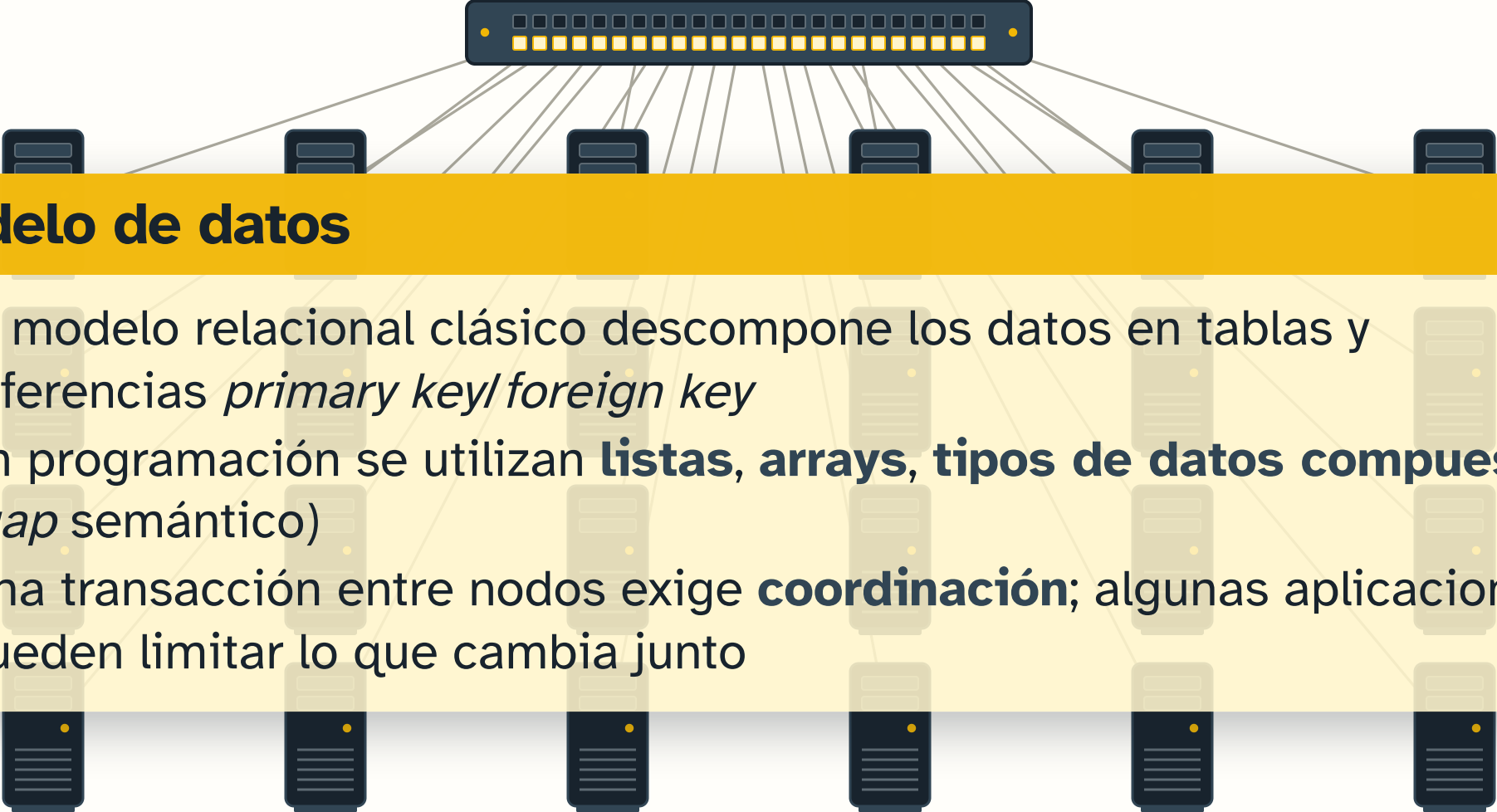
Cambio de perspectiva: red



Procesamiento distribuido (ii)

- Procesar partes independientes en paralelo (p. ej. **Map-Reduce**)
- Almacenamiento distribuido: cada nodo guarda parte de los datos
- Coordinación: localizar los datos y gestionar fallos de nodos

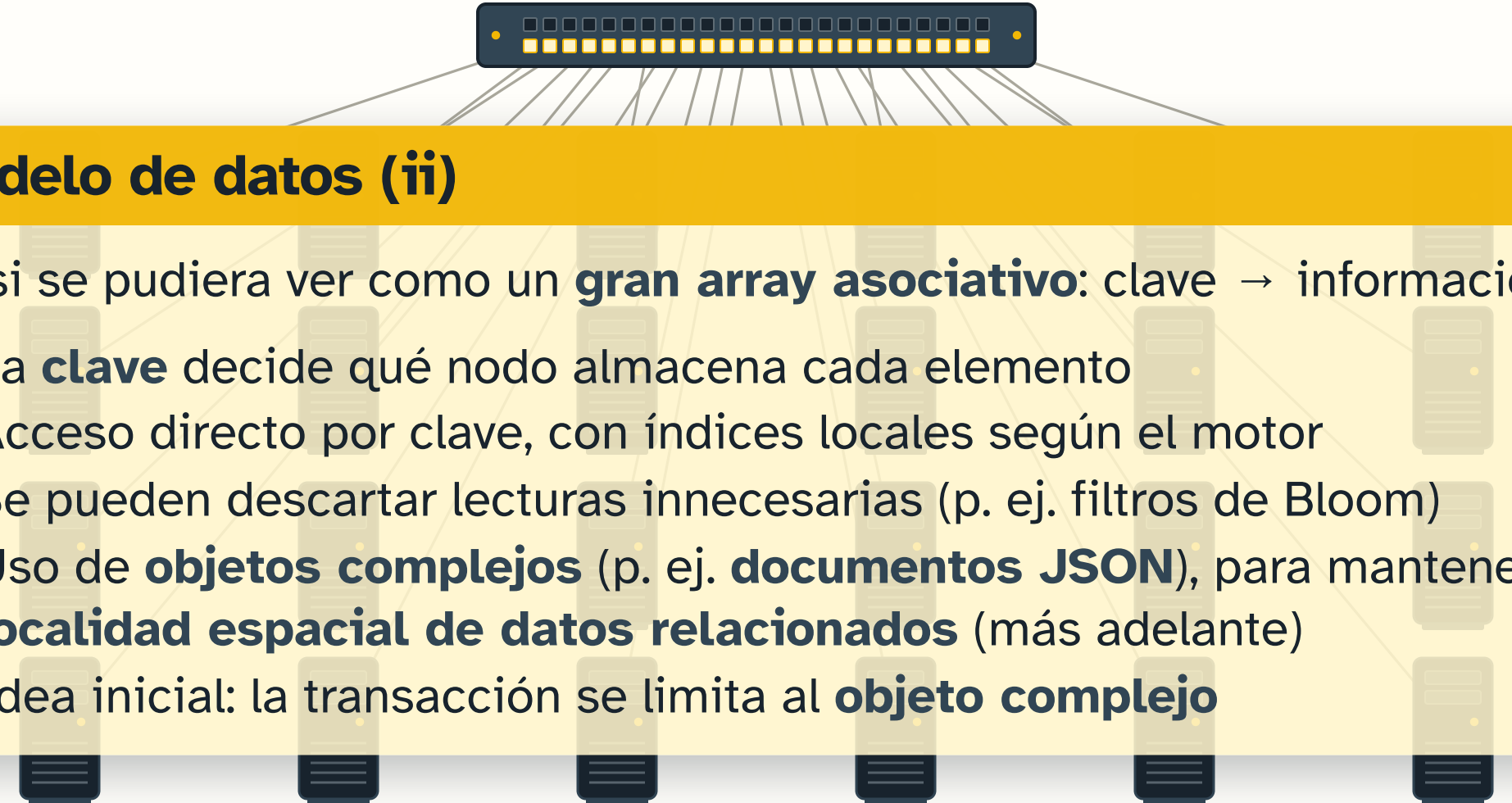
Cambio de perspectiva: red



Modelo de datos

- El modelo relacional clásico descompone los datos en tablas y referencias *primary key/foreign key*
- En programación se utilizan **listas, arrays, tipos de datos compuestos** (*gap* semántico)
- Una transacción entre nodos exige **coordinación**; algunas aplicaciones pueden limitar lo que cambia junto

Cambio de perspectiva: red



Modelo de datos (ii)

¿Y si se pudiera ver como un **gran array asociativo**: clave → información?

- La **clave** decide qué nodo almacena cada elemento
- Acceso directo por clave, con índices locales según el motor
- Se pueden descartar lecturas innecesarias (p. ej. filtros de Bloom)
- Uso de **objetos complejos** (p. ej. **documentos JSON**), para mantener la **localidad espacial de datos relacionados** (más adelante)
- Idea inicial: la transacción se limita al **objeto complejo**

El gran array se reparte y se replica

usuarios[clave] → información



Entre grupos se reparte · Dentro de cada grupo se replica

Particionado y replicación: dos decisiones

- **Particionar** reparte elementos distintos entre grupos de nodos: aumenta la capacidad
- **Replicar** mantiene varias copias de cada partición: ayuda a soportar fallos
- Una búsqueda por clave puede dirigirse a su grupo; una consulta global puede necesitar varios
- Añadir nodos no evita una **clave muy solicitada** ni una distribución desigual
- Mantener copias añade otra decisión: cuándo una escritura se considera confirmada

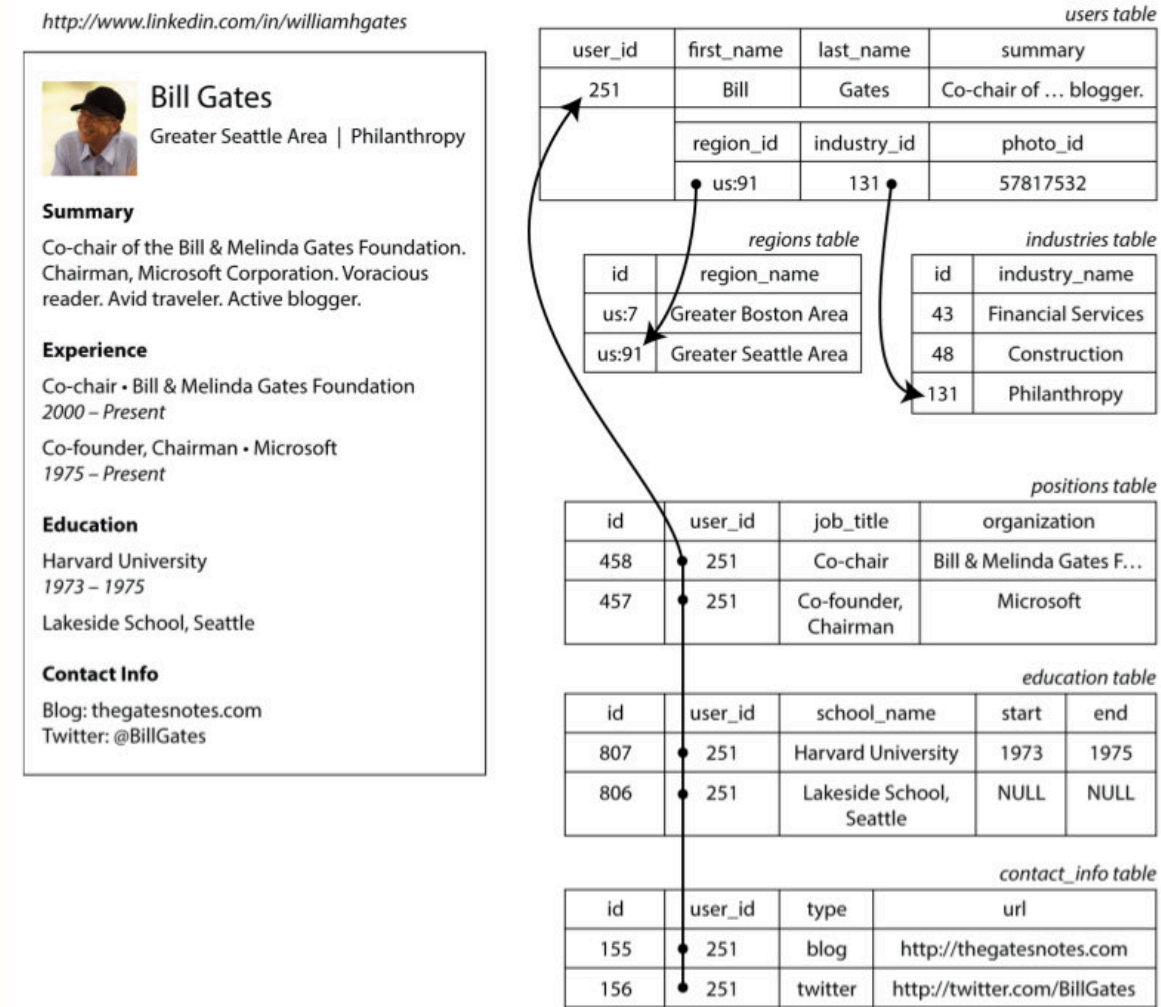
En la sesión 4 veremos el reparto y la replicación en MongoDB; en la 5, el reparto de Cassandra

El agregado: una unidad de información

- El **valor** del gran array puede ser un objeto complejo: un perfil, una pregunta con comentarios o un CV
- Un **agregado** reúne los datos que suelen leerse y cambiar juntos
- Una clave permite recuperar el conjunto sin recomponerlo con varias tablas
- El agregado es la unidad natural de **acceso**, **distribución** y **atomicidad** en muchos diseños NoSQL

Elegir qué se guarda junto es elegir el alcance de la operación local

EL CV como unidad de información



Kleppmann, 2017. *Designing Data-Intensive Applications*

Referencias entre los grandes arrays

```
usuarios[1] → { nombre: "Ana", reputación: 250 }  
preguntas[10] → { título: " ... ", usuario: 1 }  
preguntas[11] → { título: " ... ", usuario: 1 }
```

- `usuario: 1` apunta a otra clave, como la **clave ajena** que conocemos de SQL
- Referenciar no implica que el servidor compruebe la existencia del usuario ni aplique un borrado en cascada
- La aplicación puede seguir la referencia; un producto también puede ofrecer una unión
- Incrustar el perfil evita esa búsqueda, pero duplica datos si el usuario tiene varias preguntas

Schemaless

Insertar sin declarar antes el esquema: quién lo comprueba y cuándo

Schemaless

- Muchos sistemas NoSQL **no exigen declarar un esquema completo** antes de insertar
- **Flexibilidad:** posibilidad de almacenar entidades con una estructura diferente
 - Tratar información incompleta
 - Evolucionar la base de datos/esquema
 - Añadir nuevas características a las aplicaciones

El código sigue esperando campos y tipos: **flexibilidad no significa ausencia de modelo**

Schema-on-write frente a schema-on-read

Comparación **tradicional y general**, por analogía con los lenguajes estáticos y dinámicos:

	<i>Schema-on-write (SQL)</i>	<i>Schema-on-read (NoSQL)</i>
Cuándo	Los datos conforman al esquema al escribir	Los datos leídos conforman a un esquema implícito
Tipado (analogía)	Estricto (estático)	<i>Duck typing</i> (dinámico)
Datos	Homogéneos	Heterogéneos
Proceso	Analítico, a través de consultas	<i>Use as read.</i> usar los datos conforme se leen

Duck typing: el lector utiliza los campos que necesita, si están presentes y tienen tipos compatibles

Schemaless: convivir con dos versiones

```
// Documento anterior: el campo no existe
{ _id: 1, nombre: "Ana" }
// Documento nuevo: el usuario ha indicado una preferencia
{ _id: 2, nombre: "Luis", idioma: "es" }
```

```
idioma: str | None = user.get("idioma", "es")
```

- Los documentos nuevos incorporan el campo sin reescribir los anteriores
- Hay que distinguir **campo ausente**, `null` (`None` en Python) y valor no válido
- `get` devuelve `"es"` si falta la clave; si existe con valor `None`, conserva `None`

Schemaless: el mismo cambio en SQL

```
ALTER TABLE users ADD COLUMN idioma VARCHAR(10) NULL;  
UPDATE users SET idioma = 'es' WHERE idioma IS NULL;
```

- En tablas grandes, añadir la columna y rellenar los valores puede ser muy costoso
- La migración modifica muchas filas y puede mantener la tabla bloqueada

¿Cuándo es apropiado *schemaless*?

- **Objetos heterogéneos**
- Estructura de los datos **impuesta externamente**
- Si intuimos que los datos **cambiarán en el futuro**

¿Dónde se comprueba el esquema?

- En SQL, columnas, tipos y restricciones forman parte del esquema del servidor
- En un almacén clave-valor, el servidor puede tratar el valor como opaco: lo interpreta la aplicación
- En MongoDB, el uso habitual permite insertar documentos sin declarar toda su estructura
- Si se necesita, un ODM como Mongoose o el propio servidor pueden validar campos
- En Cassandra CQL se declaran columnas y tipos, aunque una fila pueda no guardar todas las celdas

Al flexibilizar el esquema, parte de la responsabilidad pasa a quien escribe y lee los datos

Modelado: qué se agrega y qué se referencia

Relaciones entre agregados: incrustar o apuntar a otra clave

Modelado de datos en NoSQL

El modelado de datos debe ser:

- Realizado al mayor nivel de abstracción posible
- Independiente de la tecnología subyacente

Sin embargo, en NoSQL:

- Siempre se tiene que tener en cuenta el diseño **distribuido**
- **Optimización y organización guiada por las consultas**

Modelado de datos en NoSQL (ii)

Con respecto al modelo de datos:

- Se mantienen los conceptos de entidad, relación, cardinalidades, etc.
- El modelado relacional se centra en especificar **qué datos tenemos y podemos ofrecer**
- El modelo NoSQL se centra en **optimizar qué consultas vamos a servir**
- Puede compensar **duplicar (desnormalizar)** datos para optimizar accesos, a cambio de mantener coherentes las copias
- **Agregar o referenciar** —la decisión que planteamos en la sesión 1— según qué consultas queramos tener optimizadas

Representación de relaciones: modelo de documentos

- **Modelo de documentos** → analogía del **array/mapa gigante**
- **Conjunto de documentos** (objetos complejos):
 - Un **identificador único**, campo *id*
 - Búsqueda aleatoria eficiente por clave (**referencia**)
 - Estructura jerárquica de subdocumentos contenidos → **agregación**

Frente a las tablas normalizadas, el documento permite elegir qué relaciones se agregan y cuáles se referencian

Uno a muchos: agregar o referenciar

Una pregunta tiene varios comentarios:

- **Agregar:** guardar los comentarios dentro de la pregunta
- **Referenciar:** guardar cada comentario con la clave de su pregunta

Si se leen juntos, un agregado evita recomponer el hilo

Si crecen sin límite, se actualizan de forma independiente o se consultan por separado, las referencias pueden encajar mejor

Son los conceptos de entidad y cardinalidad de SQL, pero con otras restricciones de diseño

Una pregunta y sus comentarios

Separar y referenciar

Tablas SQL: Posts y Comments

Id	Title
10	Ordenar por fecha

PostId → Posts.Id · 1:N

Comments · dos filas

Id	PostId	Text
101	10	¿Fecha o texto?
102	10	Incluye un ejemplo

Recomponer con JOIN o buscar por PostId

Ejemplo didáctico · En documentos también se puede separar y referenciar por PostId

Agregar en un documento

Colección posts: comentarios incrustados

```
{ "_id": 10,  
  "Title": "Ordenar por fecha",  
  "Comentarios": [  
    { "Id": 101,  
      "Text": "¿Fecha o texto?" },  
    { "Id": 102,  
      "Text": "Incluye un ejemplo" }  
  ] }
```

Leer la pregunta con sus comentarios

Uno a muchos: leer y modificar

Operación	Separar y referenciar	Agregar en el documento
Mostrar la pregunta y sus comentarios	Recuperar la pregunta y sus comentarios por <code>PostId</code>	Recuperar un documento con el conjunto
Añadir un comentario	Insertar otra fila o documento con <code>PostId: 10</code>	Añadir un elemento al array de comentarios de la pregunta 10
Consultar comentarios de muchas preguntas	Consultar la tabla o colección de comentarios	Buscar dentro de los arrays de las preguntas

La relación sigue siendo **uno a muchos**; cambia dónde se guarda y qué unidad se lee o modifica

Muchos a uno y muchos a muchos

- **Muchos a uno:** varios perfiles pueden apuntar a una misma región
- **Muchos a muchos:** un post puede tener varios *tags* y un *tag* aparecer en varios posts
- Si una entidad compartida tiene identidad propia, referenciar su `id` evita duplicarla y facilita sus cambios
- Seguir la referencia puede exigir otra petición; algunos productos ofrecen uniones, como `$lookup` en MongoDB

Posts y tags: ¿desde qué clave buscamos?

posts

Leer los tags de un post

```
_id: 10  
title: "Ordenar por fecha"  
tags: ["sql", "fechas"]
```

```
_id: 11  
title: "Contar filas"  
tags: ["sql"]
```

tags

Leer los posts de un tag

```
_id: "sql"  
posts: [10, 11]
```

```
_id: "fechas"  
posts: [10]
```

Mantener ambas construcciones permite dos lecturas: post → tags y tag → posts

Si se mantienen las dos, añadir o quitar una asociación exige dos escrituras

Cada documento parte de una clave y contiene un array 1:N; la relación completa es N:M

Posts y tags: elegir qué guardar

Si necesitamos...	Podemos guardar...	Consecuencia
Tags de un post	Documento en <code>posts</code> con <code>_id: 10</code> y <code>tags: [...]</code>	Para hallar posts con <code>sql</code> hace falta otro acceso o índice
Posts de un tag	Documento en <code>tags</code> con <code>_id: "sql"</code> y <code>posts: [...]</code>	La lista puede crecer mucho; sólo contiene IDs
Consultar en ambos sentidos	Ambas colecciones	Cada asociación se escribe dos veces y hay que mantenerlas coherentes

Son dos construcciones para las **mismas asociaciones**; las consultas determinan si conviene mantener una o ambas

Agregar también tiene límites

- Si el agregado crece sin límite, leerlo o actualizarlo puede resultar costoso
- Incrustar datos compartidos evita búsquedas, pero obliga a actualizar varias copias
- Referenciar mantiene una identidad común; seguir la referencia puede requerir otra petición
- Si dos agregados deben cambiar juntos, hace falta coordinación adicional (**transacciones multidocumento**)

La decisión se toma con las cardinalidades y los accesos que conocemos, no sólo con el formato JSON

Map-Reduce

De las funciones `map` y `reduce` a las consultas sobre datos repartidos

Map-Reduce

Origen en **lenguajes funcionales**:

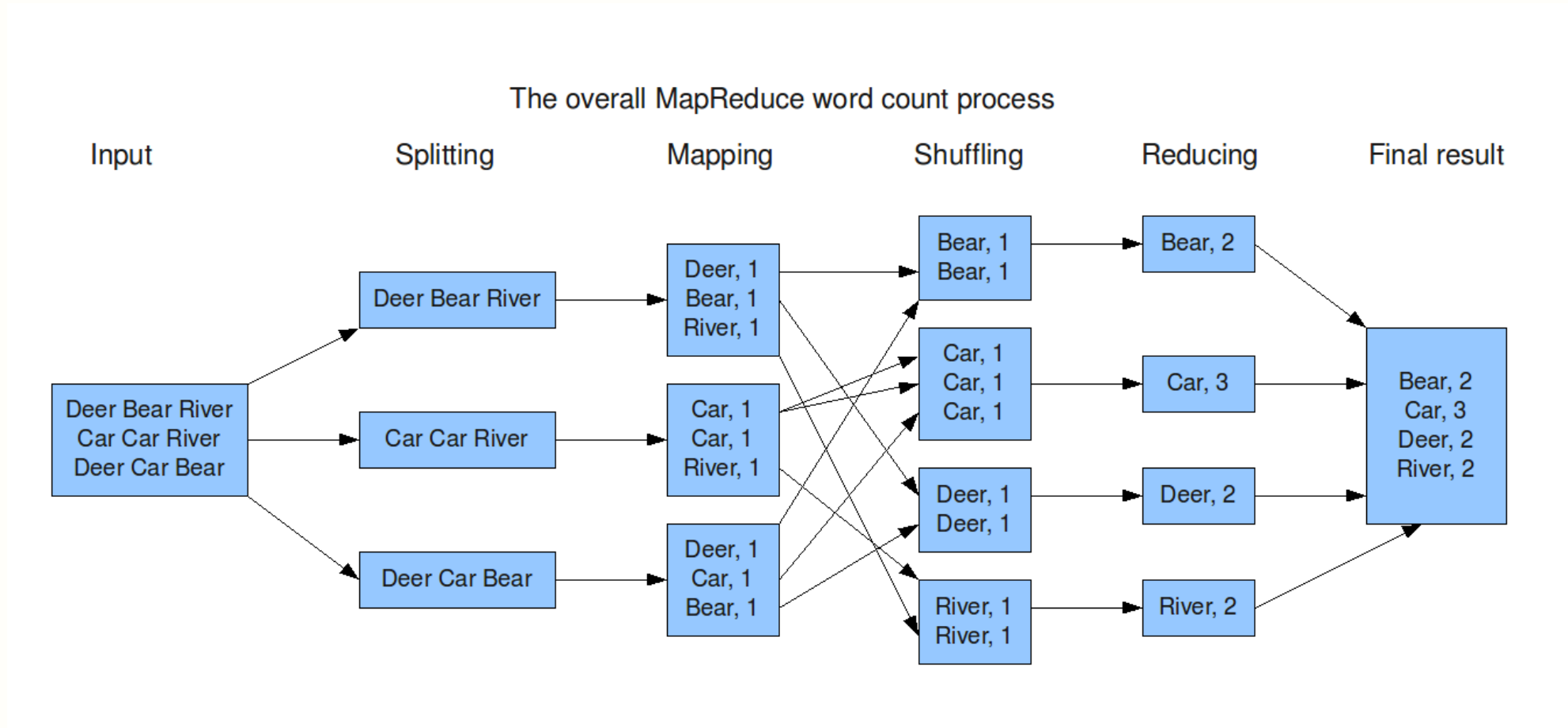
- `map()` : ejecuta una misma función sobre todos los elementos de un conjunto
- `reduce()` : sumariza un conjunto de valores para producir un valor de salida

Map-Reduce: combinando ambas operaciones

- Una misma operación `map()` sobre cada dato residente en un nodo se realiza en paralelo en **todos** los nodos simultáneamente
- Con los resultados parciales, una función `reduce()` genera el resultado final
- Proceso intermedio de *shuffle*, para agrupar valores **con la misma clave** antes del `reduce()`
- Resultados parciales en el mismo nodo (localidad) $\Rightarrow$ procesamientos **en cadena**

Map-Reduce — *Word Count*

<http://www.milanor.net/blog/an-example-of-mapreduce-with-rmr2/>



Map-Reduce: particularidades

- Se supone que los datos de entrada son siempre pares $\langle key, value \rangle$
- La función `map()` produce otro conjunto de valores $\{\langle key1, value1 \rangle, \langle key2, value2 \rangle, \dots\}$
- El *shuffle* agrupa los valores con la misma clave: $\{\langle key1, \{val1, val3, \dots\} \rangle, \langle key2, \{val2, val4, \dots\} \rangle, \dots\}$
- `reduce()` procesa cada lista de valores con la misma clave, y produce otros elementos $\langle key', value' \rangle$
- Hay procesamientos difíciles de expresar en Map-Reduce $\Rightarrow$ operaciones M/R **en cadena**
- El procesamiento distribuido se desarrolla en Tecnologías de Computación de Datos Masivos

Map-Reduce y consultas

Map-Reduce puede usarse no sólo para computación distribuida, sino también como una **generalización de consultas**. Ejemplo: un biólogo marino anota cada animal que ve en el océano y quiere saber cuántos tiburones ha visto por mes:

```
SELECT YEAR(observation_timestamp) AS observation_year,  
       MONTH(observation_timestamp) AS observation_month,  
       sum(num_animals) AS total_animals  
FROM observations  
WHERE family = 'Sharks'  
GROUP BY observation_year, observation_month;
```

Map-Reduce y consultas (ii)

El mismo agrupamiento con el API **histórico** de MongoDB, deprecado desde 5.0:

```
db.observations.mapReduce(  
  function map() {  
    var year = this.observationTimestamp.getUTCFullYear();  
    var month = this.observationTimestamp.getUTCMonth() + 1;  
    emit(year + "-" + month, this.numAnimals);  
  },  
  function reduce(key, values) {  
    return values.reduce((total, value) => total + value, 0);  
  },  
  {  
    query: { family: "Sharks" },  
    out: "monthlySharkReport"  
  }  
);
```

Map-Reduce y consultas (iii)

Hoy se usa `aggregate` : `$match` filtra, `$group` elige la misma clave y `$sum` reduce los valores

```
db.observations.aggregate([
  { $match: { family: "Sharks" } },
  { $group: {
    _id: {
      year: { $year: "$observationTimestamp" },
      month: { $month: "$observationTimestamp" }
    },
    totalAnimals: { $sum: "$numAnimals" }
  }
}]
```

La misma entrada debe producir los mismos totales por **año y mes**; en la sesión 4 seguiremos este *pipeline* etapa a etapa

mySQL

```
SELECT
  Dim1, Dim2,
  SUM(Measure1) AS MSum,
  COUNT(*) AS RecordCount,
  AVG(Measure2) AS MAvg,
  MIN(Measure1) AS MMin
  MAX(CASE
    WHEN Measure2 < 100
    THEN Measure2
  END) AS MMax
FROM DenormAggTable
WHERE (Filter1 IN ('A','B'))
  AND (Filter2 = 'C')
  AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8
```

- ① Grouped dimension columns are pulled out as keys in the map function, reducing the size of the working set.
- ② Measures must be manually aggregated.
- ③ Aggregates depending on record counts must wait until finalization.
- ④ Measures can use procedural logic.
- ⑤ Filters have an ORM/ActiveRecord-looking style.
- ⑥ Aggregate filtering must be applied to the result set, not in the map/reduce.
- ⑦ Ascending: 1; Descending: -1

MongoDB

```
db.runCommand({
  mapreduce: "DenormAggCollection",
  query: {
    filter1: { '$in': [ 'A', 'B' ] },
    filter2: 'C',
    filter3: { '$gt': 123 }
  },
  map: function() { emit(
    { d1: this.Dim1, d2: this.Dim2 },
    { msum: this.measure1, recs: 1, mmin: this.measure1,
      mmax: this.measure2 < 100 ? this.measure2 : 0 }
  ); },
  reduce: function(key, vals) {
    var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 };
    for(var i = 0; i < vals.length; i++) {
      ret.msum += vals[i].msum;
      ret.recs += vals[i].recs;
      if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
      if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax))
        ret.mmax = vals[i].mmax;
    }
    return ret;
  },
  finalize: function(key, val) {
    val.mavg = val.msum / val.recs;
    return val;
  },
  out: 'result1',
  verbose: true
});
db.result1.
  find({ mmin: { '$gt': 0 } }).
  sort({ recs: -1 }).
  skip(4).
  limit(8);
```

Revision 4, Created 2010-03-06
Rick Osborne, rickosborne.org

Eficiencia *raw*

MongoDB frente a MySQL con los mismos datos y las mismas operaciones

¿Es viable? La primera pregunta es la eficiencia

- Hasta aquí hemos visto una forma distinta de guardar los datos: claves, agregados y estructura flexible
- SQL ya existe y está muy optimizado: si el modelo nuevo fuese **mucho más lento** con los mismos datos, no sería viable, por mucha flexibilidad o distribución que ofreciera
- Es una condición **necesaria, no suficiente**: no demuestra que sea mejor, sólo que no tiene un precio de entrada en rendimiento
- Lo comprobamos con la misma carga en MySQL y en MongoDB

Primero, ¿pierde frente a SQL? Después, ¿qué ventajas compensan sus compromisos?

Rendimiento: comparar operaciones completas

- Comparamos MySQL y MongoDB sobre **el mismo millón de medidas de sensores**
- Un solo cliente genera los mismos datos y mide operaciones completas en segundos
- Se comprueban filas iniciales, cambios, borrados, suma de lecturas y los cinco resultados del intervalo
- Un calentamiento y tres repeticiones: se muestran **mediana y rango**
- Dos servidores locales de un nodo: no es una prueba de escalabilidad distribuida

Código y resultados: `bdge/addendum/mysql_vs_mongo/`

Rendimiento: qué se está midiendo

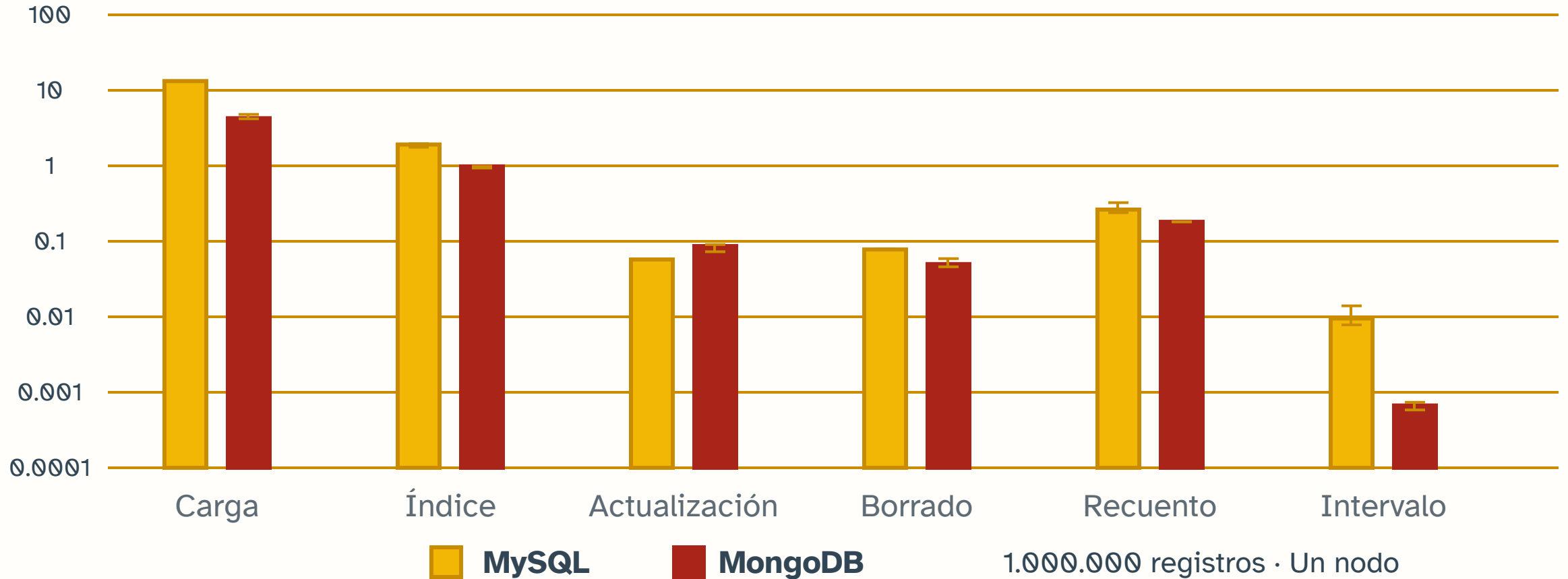
- MySQL 8 y MongoDB 8.3, con el mismo límite por servidor: **2 CPU y 2 GiB**
- Carga por lotes de 5.000, sin incluir la generación de datos en el cronómetro
- Índice en la fecha; mismos rangos y orden de operaciones en ambos productos
- `COUNT(*)` y `count_documents` : recuentos exactos, no estimaciones de metadatos
- La consulta de intervalo recibe los cinco resultados **antes de detener el tiempo**
- Escrituras confirmadas con persistencia de diario; la atomicidad sigue siendo distinta en ambos modelos

Eficiencia *raw*: las seis pruebas

1. Insertar un millón de elementos deterministas, con fechas que permitan la búsqueda por rango (**Fill**)
2. Crear un índice en la tabla para la fecha de la lectura (**Index**)
3. Actualizar los valores de un conjunto de entradas seleccionadas por rango de fechas (**Update**)
4. Eliminar un conjunto de filas seleccionadas por rango de fechas (**Delete**)
5. Obtener el número de filas restantes (**Count**)
6. Obtener cinco registros ordenados por fecha, tras saltar veinte, con el mismo filtro (**Interval**)

MySQL 8.4.11 · MongoDB 8.3.11

Tiempo completo (s), escala logarítmica · Mediana y rango de tres repeticiones



Rendimiento: interpretar el gráfico

- El eje Y es **logarítmico**: distancias iguales representan factores iguales
- Las barras son medianas; los segmentos muestran el mínimo y máximo observados
- Se comparan operaciones concretas, con estas versiones y esta configuración
- La carga incluye cliente, envío, almacenamiento y confirmación; no mide sólo el motor
- El resultado puede cambiar al variar índices, lotes, concurrencia y garantías

Antes de explicar una diferencia, comprobar que ambas operaciones hacen el mismo trabajo

Rendimiento: ¿es viable?

- Con la misma carga, ninguna operación queda **órdenes de magnitud** por detrás en MongoDB; la actualización es la única que favorece a MySQL
- Respuesta a la pregunta de partida: el rendimiento **no impide** adoptar el modelo
- Pero no lo justifica: es un solo nodo, así que la **distribución**, la otra mitad del argumento, sigue sin probarse
- Para decidir hay que preguntar:
 - ¿Qué operaciones predominan: carga, actualización o lectura?
 - ¿Necesitamos una consulta global o recuperar un agregado por clave?
 - ¿Qué garantías necesitamos cuando fallan un nodo o la red?
 - ¿Cuánto cuesta mantener los índices y las copias de datos?

Superado el listón de la eficiencia, la elección depende del patrón de acceso: no hay un ganador universal

Tipos de sistemas NoSQL

Clave-valor, documentos, familias de columnas, grafos y *arrays*

Tipos de sistemas NoSQL

- Las familias se distinguen por la **estructura del valor** y las **operaciones** que ofrece el servidor
- Clave-valor: acceso por clave, con un valor generalmente opaco
- Documentos: el servidor puede consultar campos y subdocumentos
- Familias de columnas: acceso a filas o particiones y a sus celdas
- Grafos: entidades y relaciones que se recorren
- *Arrays*: operaciones sobre datos multidimensionales

Vamos a compararlas con las tablas y operaciones SQL que ya conocemos

Key-Value Stores y documentales

- Cada pieza de datos tiene asignado un identificador
- La diferencia entre ambas es que:
 - En *key-value*, el valor es opaco: no se conoce nada de su interior (a todos los efectos es un *blob* de datos)
 - En las basadas en documentos, la base de datos puede ver el contenido del agregado, y utilizar su información como parte de las búsquedas y actualizaciones
- Documentos $\Rightarrow$ formatos jerárquicos tipo JSON o XML

Clave-valor y documentos frente a SQL

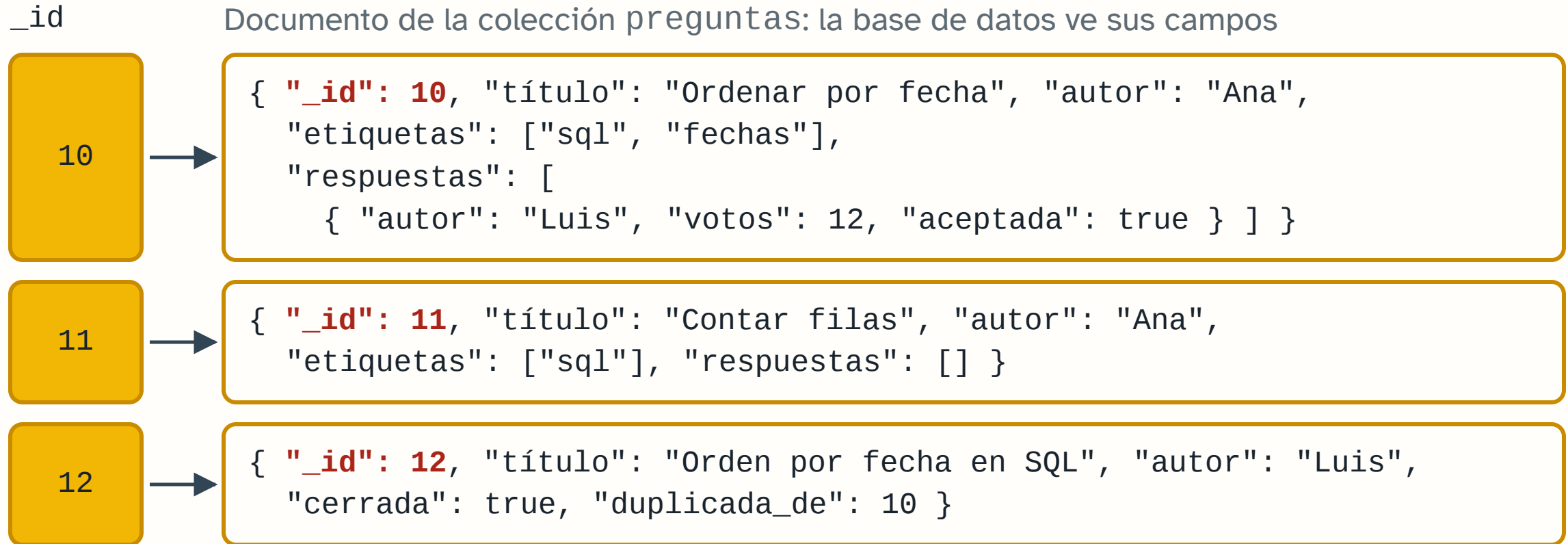
	Clave-valor	Documentos
Unidad de acceso	Valor completo por clave	Documento; también filtros por campos
Estructura	La interpreta principalmente el cliente	La conoce y consulta el servidor
Referencias	Otra clave, seguida por el cliente	Otro documento; algunas BD permiten uniones
Frente a SQL	Menos operaciones generales, acceso directo	Agregados jerárquicos frente a tablas normalizadas

Ejemplos: Redis o LevelDB para clave-valor; MongoDB o Couchbase para documentos

Key-value: a cada valor se llega por su clave

Clave	Valor	Tipo del valor
sesión:9f2c	eyJ1c3Vhcm1vIjogMX0...	string opaco para la base de datos
pregunta:10:visitas	1841	string contador: INCR
usuario:1	nombre Ana reputación 250	hash campo → valor
pregunta:10:respuestas	21 22 23	list en orden de llegada
ranking:reputación	80 Luis 250 Ana	sorted set ordenado por puntuación

Documentos: la base de datos ve el valor



Se puede buscar por cualquier campo: `db.preguntas.find({ "etiquetas": "fechas" })` → **`_id 10`**

Almacenar por filas o por columnas

Almacenamiento por filas

Una franja por fila

	id	lugar	lectura
fila 1	1	10	18
fila 2	2	20	25
fila 3	3	10	21

Cada fila queda contigua en disco

Almacenamiento por columnas

Una franja por columna

id	1	2	3	...
lugar	10	20	10	...
lectura	18	25	21	...

Cada columna queda contigua en disco

$\text{SUM}(\text{lectura}) = 64$ · Se necesitan sólo los valores de lectura

La BD conoce dónde empieza cada columna y puede leer varias en paralelo

Dos sentidos distintos de «columnar»

- **Columnar analítico:** agrupa los valores de una misma columna de muchas filas
 - Permite leer sólo las columnas de un análisis y comprimir valores parecidos
 - No cambia necesariamente el modelo relacional ni el uso de SQL
- **Familias de columnas / wide-column:** organiza celdas dentro de filas o particiones identificadas por clave
 - Agrupa datos relacionados, admite datos dispersos y favorece accesos por clave
 - HBase y Cassandra pertenecen a esta segunda familia

Comparten el interés por organizar celdas; no tienen la misma disposición física

Familias de columnas: del gran array a las celdas

- Influencia de **Bigtable, Google, 2006**
- Cada clave identifica una fila o partición; dentro se accede a celdas por sus nombres
- **HBase**: se declaran familias; cada fila puede tener columnas diferentes
- **Cassandra CQL**: se declaran columnas y tipos; no todas las filas guardan todas las celdas
- A diferencia de una tabla SQL normalizada, la clave se diseña para localizar y reunir lo que se consulta junto
- El almacenamiento de Cassandra organiza filas **por partición**, no una columna completa de todas las filas

Familias de columnas: el modelo de HBase

Clave de fila	Familia de columnas perfil	Familia de columnas actividad								
ana	<table><tr><td>nombre</td><td>ciudad</td></tr><tr><td>Ana</td><td>Murcia</td></tr></table>	nombre	ciudad	Ana	Murcia	<table><tr><td>reputación</td><td></td></tr><tr><td>250</td><td></td></tr></table>	reputación		250	
nombre	ciudad									
Ana	Murcia									
reputación										
250										
luis	<table><tr><td>nombre</td><td>web</td></tr><tr><td>Luis</td><td>luis.dev</td></tr></table>	nombre	web	Luis	luis.dev	<table><tr><td>reputación</td><td>respuestas</td></tr><tr><td>80</td><td>1</td></tr></table>	reputación	respuestas	80	1
nombre	web									
Luis	luis.dev									
reputación	respuestas									
80	1									
marta	<table><tr><td>nombre</td><td></td></tr><tr><td>Marta</td><td></td></tr></table>	nombre		Marta		sin columnas				
nombre										
Marta										

Una celda: (fila, familia:columna) → valor p. ej. (ana, actividad:reputación) → 250

Familias de columnas: las filas de Cassandra

Clave de partición

Celdas de la fila, por orden de columna: sólo las que tienen valor

ana

ciudad

Murcia

nombre

Ana

reputación

250

luis

nombre

Luis

reputación

80

respuestas

1

web

luis.dev

marta

nombre

Marta

```
ALTER TABLE usuarios ADD web text;
```

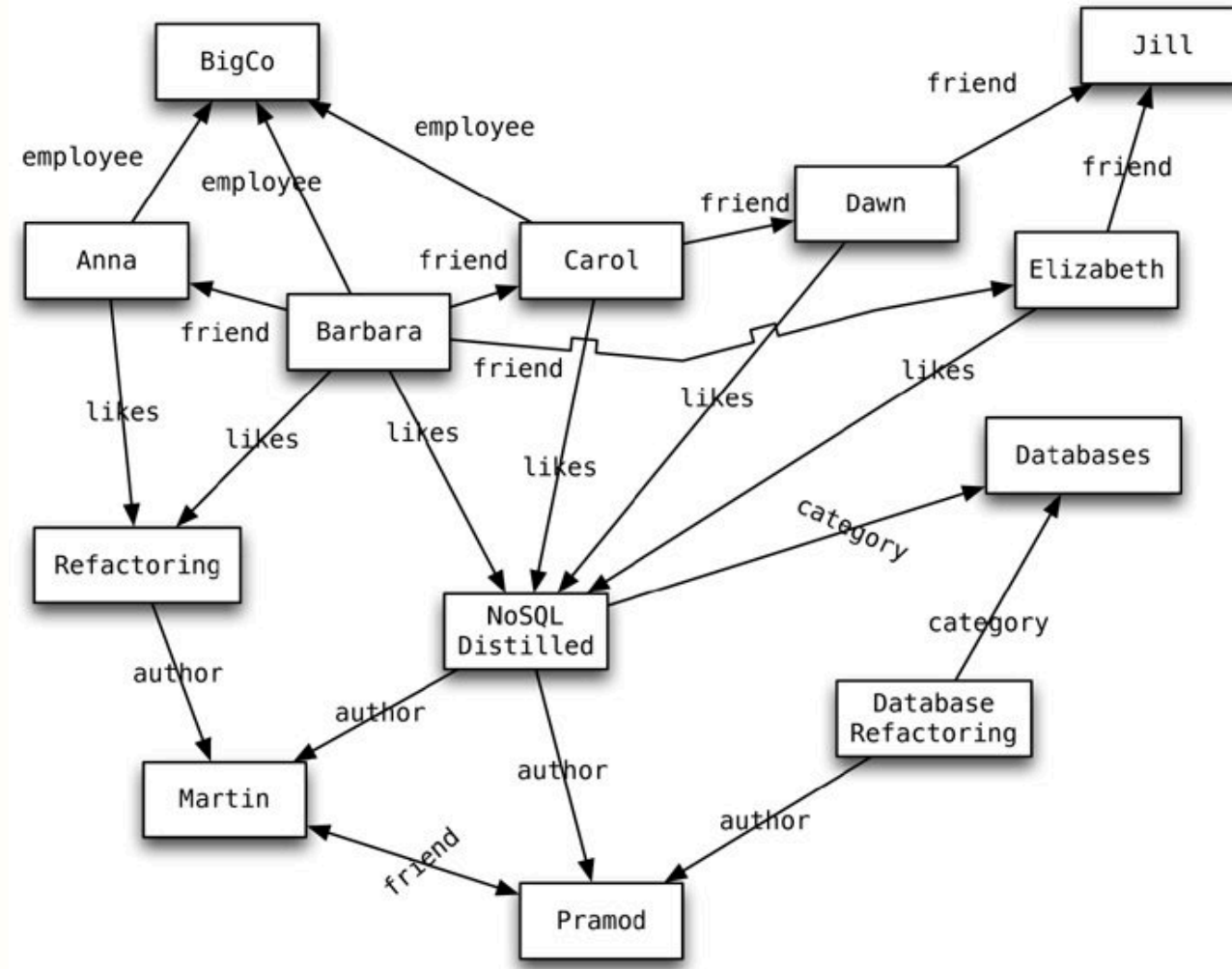
no reescribe ninguna fila

```
UPDATE usuarios SET web = 'luis.dev' WHERE usuario = 'luis';
```

sólo esta fila guarda la celda

Bases de datos de grafos

- Las bases de datos de grafos llevan el mecanismo *muchos a muchos* al extremo
- Datos en los que existen muchas relaciones entre sí, con un significado primordial
- Se basan en la construcción y consulta de grafos que constan de:
 - **Vértices**, también llamados *nodos* o *entidades*
 - **Aristas** (*edges*), también llamadas *relaciones*
- Pueden capturar relaciones complejas entre entidades y ofrecen lenguajes de búsqueda, actualización y creación que permiten trabajar con subconjuntos del grafo
- Ejemplos: Neo4j; OrientDB combina documentos y grafos



Grafos representados en SQL

Nota: usa la sintaxis PostgreSQL para `json`

```
CREATE TABLE vertices (  
    vertex_id integer PRIMARY KEY,  
    properties json  
);  
  
CREATE TABLE edges (  
    edge_id integer PRIMARY KEY,  
    tail_vertex integer REFERENCES vertices (vertex_id),  
    head_vertex integer REFERENCES vertices (vertex_id),  
    label text,  
    properties json  
);  
  
CREATE INDEX edges_tails ON edges (tail_vertex);  
CREATE INDEX edges_heads ON edges (head_vertex);
```

Grafo y consulta en Neo4j

CREATE

```
(NAmerica:Location {name:'North America', type:'continent'}),  
(USA:Location {name:'United States', type:'country' }),  
(Idaho:Location {name:'Idaho', type:'state' }),  
(Lucy:Person {name:'Lucy' }),  
(Europe:Location {name:'Europe', type:'continent'}),  
(Idaho)-[:WITHIN]→(USA)-[:WITHIN]→ (NAmerica),  
(Lucy)-[:BORN_IN]→(Idaho),  
(Lucy)-[:LIVES_IN]→(Europe)
```

Consulta:

MATCH

```
(person) -[:BORN_IN]→ ( ) -[:WITHIN*0..]→ (us:Location {name:'United States'}),  
(person) -[:LIVES_IN]→ ( ) -[:WITHIN*0..]→ (eu:Location {name:'Europe'})  
RETURN person.name
```

Bases de datos basadas en *arrays*

- Suelen presentarse como bases de datos que soportan SQL y añaden operaciones para trabajar con conjuntos de datos especiales (*arrays*)
- Utilizadas para tratamiento de grandes cantidades de datos de forma estadística, de modelado y OLAP
- Soportan también datos geográficos, ya que pueden definir rangos numéricos de una o varias dimensiones (2D para cálculos geográficos)
- Ejemplos: SciDB, rasdaman

Comparar y elegir

Qué queda de las diferencias y cómo combinar varios modelos

SQL y NoSQL: ideas que convergen

Diferencia tradicional	Hoy	Dónde lo veremos
Tablas frente a objetos complejos	Columnas JSON con consultas e índices en SQL	Sesión 2
Esquema obligatorio frente a flexible	Validación opcional en MongoDB; esquema declarado en CQL	Sesiones 4 y 5
Transacción entre entidades frente a agregado	Transacciones multidocumento en MongoDB	Sesión 4
Un servidor frente a datos distribuidos	Particionado de tablas en SQL; replicación y reparto en NoSQL	Sesiones 2, 4 y 5
Lenguajes y accesos diferentes	\$lookup y planes de consulta en MongoDB; índices secundarios en Cassandra	Sesiones 4 y 5

El modelo sigue importando: las capacidades se acercan, pero sus unidades naturales de acceso son distintas

Elegir y combinar

- Tres preguntas deciden: ¿qué consultas hay que **optimizar**?, ¿qué debe **cambiar junto**?, ¿cuánto cuesta **operar** el sistema?
- En el foro: SQL para usuarios y votos, documentos para el hilo, clave-valor para sesiones y contadores, grafo para «quién responde a quién»
- **Persistencia políglota**: cada parte con el modelo que le encaja
- Combinar añade trabajo: sincronización, fallos parciales y mantenimiento

Seguimos con los documentos: es el modelo más cercano al hilo que acabamos de ver, y MongoDB lo lleva a un producto

