

Bases de Datos a Gran Escala

Sesión 4 · Bases de datos basadas en documentos

Diego Sevilla Ruiz

Departamento de Ingeniería y Tecnología de Computadores

Facultad de Informática · Universidad de Murcia

dsevilla@um.es

Del modelo documental a MongoDB

Lenguaje, consultas, índices y garantías de un producto concreto

Lo que traemos de la sesión 3

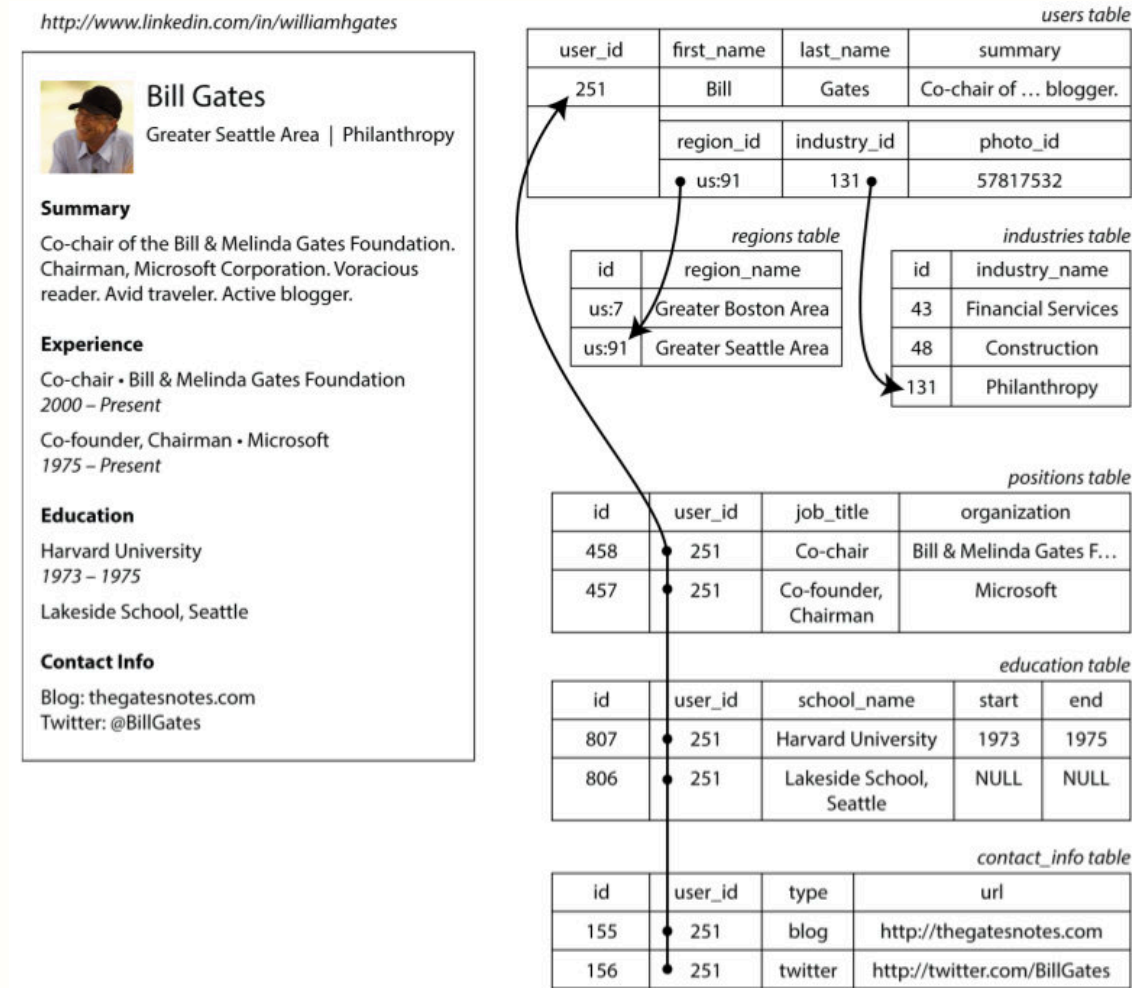
- Una colección es el **gran array** de documentos identificados por clave
- Cada documento representa un agregado; otras claves permiten referenciar entidades
- Agregar y referenciar decide qué se recupera y cambia junto
- La estructura puede evolucionar sin exigir las mismas celdas a todos los documentos

Ahora veremos cómo MongoDB expresa esas decisiones y qué garantías ofrece

Del agregado al documento: un CV

Retomamos el CV de Bill Gates de la sesión 3 y lo representamos como documento

Un mismo perfil, descompuesto en varias tablas



Kleppmann, 2017. *Designing Data-Intensive Applications*

Representación de relaciones uno a muchos

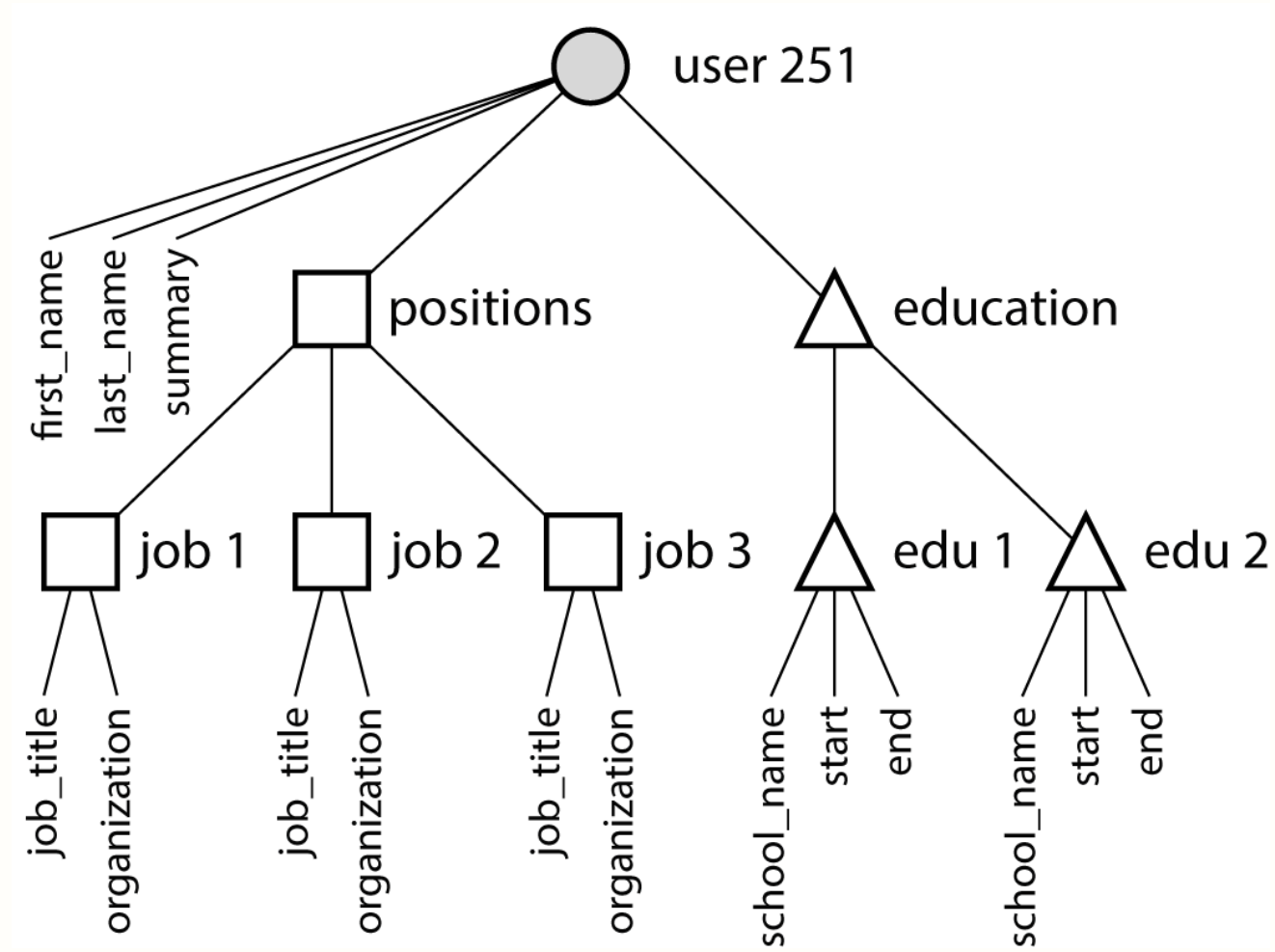
Las relaciones uno a muchos (por ejemplo, en el CV: `positions`) en el modelo relacional:

- Normalización usando varias tablas (`Positions` con `user_id`)
 - Necesidad de más de una tabla
 - Necesidad de recomponer el conjunto con `JOIN` : su coste depende de los índices y del volumen
- Los SGBDR actuales ofrecen además columnas JSON (sesión 2), con consultas e índices sobre sus campos
 - El estándar las recoge desde SQL:2016, pero cada gestor tiene su propia sintaxis
 - El documento es ahí un añadido a la fila, no la unidad del modelo

CV como un documento

```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates ... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    { "job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation" },
    { "job_title": "Co-founder, Chairman", "organization": "Microsoft" }],
  "education": [
    { "school_name": "Harvard University", "start": 1973, "end": 1975 },
    { "school_name": "Lakeside School, Seattle", "start": null, "end": null }],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

CV como un árbol (equivalente)



Del CV al foro: el mismo uno a muchos

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" }  
  ] }
```

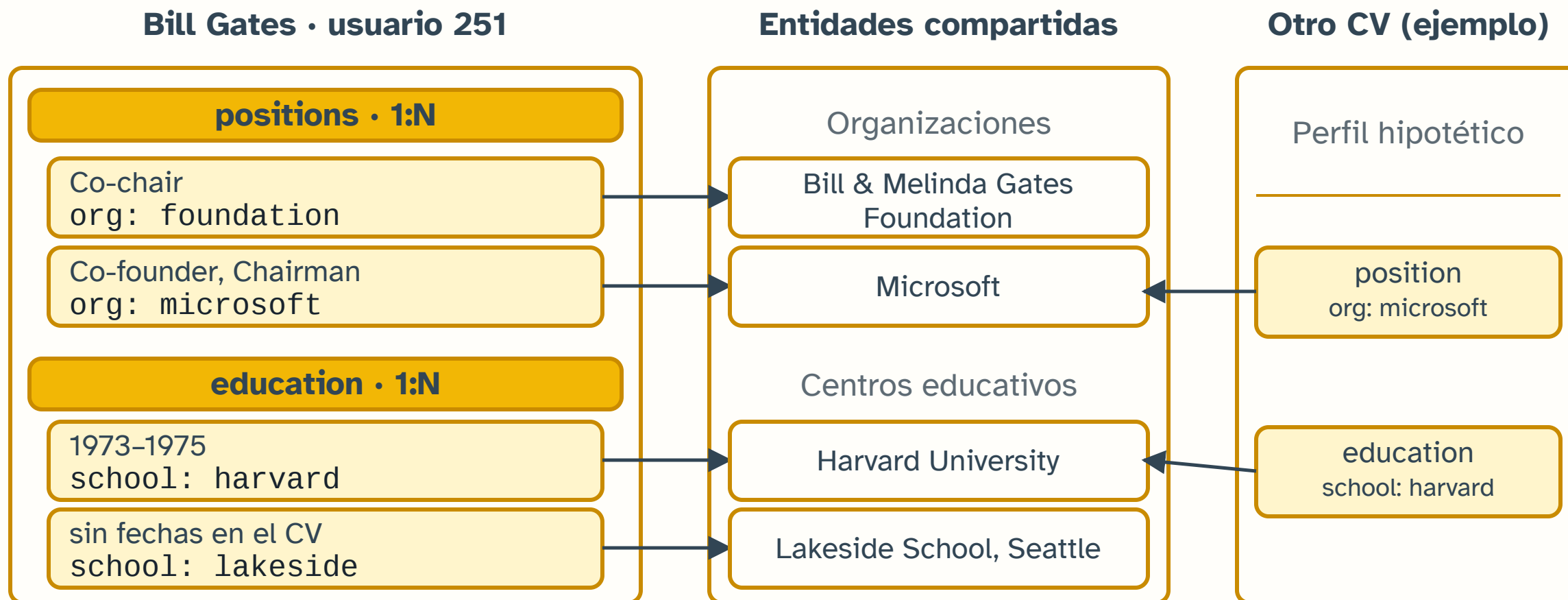
Leer la pregunta con sus comentarios

Qué queda fuera del documento

- Persona → puestos en el CV; pregunta → comentarios en el foro: ambas relaciones son **uno a muchos**
- Las posiciones y estudios forman parte del CV y se pueden recuperar con él
- Una región, una industria o una empresa pueden tener identidad propia y estar compartidas
- Referenciar esas entidades evita repetir su información en cada CV
- Incrustar una copia del nombre puede acelerar la presentación, pero requiere decidir cómo se actualiza

El formato admite ambas opciones; el modelado decide cuál necesitamos

EL CV: uno a muchos y muchos a muchos



Una alternativa al documento anterior: puestos y estudios dentro del CV; entidades por referencia

El segundo perfil es hipotético y muestra la relación N:M con Microsoft y Harvard

Posts y tags: dos sentidos de acceso

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

Esquema flexible no es ausencia de esquema

- Como vimos en la sesión 3, sin validación explícita el contrato vive en el código que escribe y lee
- Dos documentos de una colección pueden tener campos distintos, y un campo **ausente** no es lo mismo que un campo a `null`
- Cuando conviene, la base de datos también puede **validar al escribir**. En MongoDB:

```
db.createCollection("users", { validator: { $jsonSchema: {
  required: ["Id", "DisplayName"],
  properties: { Reputation: { bsonType: "int", minimum: 0 } }
} } })
```

- El esquema pasa de obligatorio a **gradual**: se exige sólo donde aporta

Introducción a MongoDB

BSON, consultas que son documentos y dónde se ejecutan

Introducción a MongoDB

- Base de datos documental: bases de datos → colecciones → documentos BSON
- Cada documento tiene un `_id` **único en la colección**; puede ser una clave elegida por la aplicación
- Si se omite, el cliente genera normalmente un `ObjectId`
- No exige declarar todos los campos antes de insertar; se puede añadir validación
- Consulta con `find` y *pipelines* con `aggregate` ; índices sobre campos simples, compuestos y anidados
- `mongosh` es un cliente JavaScript; hay *drivers* para Python y otros lenguajes

BSON: JSON con tipos

JSON sólo distingue cadenas, números, booleanos, `null`, *arrays* y objetos. BSON, el formato binario en que MongoDB guarda y transmite los documentos, añade tipos:

Tipo BSON	Para qué	En Python (pymongo)
ObjectId	Identificador único de 12 bytes: el <code>_id</code> por omisión	<code>bson.ObjectId</code>
Date	Instante en milisegundos (UTC)	<code>datetime.datetime</code>
Int32 , Int64 , Double	Números con tipo definido	<code>int</code> , <code>float</code>
Decimal128	Decimales exactos (importes)	<code>bson.Decimal128</code>
BinData	Datos binarios	<code>bytes</code>

Un documento ocupa como máximo **16 MB**: un límite que condiciona cuánto se puede agregar

¿Dónde está el SQL de MongoDB?

- A una base de datos relacional se le envía un **texto** en un lenguaje propio y declarativo: SQL
- MongoDB no tiene un lenguaje de consulta textual: su *shell* (`mongosh`) es un intérprete de **JavaScript**
- Lo que en SQL es sintaxis, aquí son **objetos y métodos** del lenguaje:
 - `db` es un objeto: la base de datos en uso
 - Cada colección es una **propiedad** de `db` : `db.posts` o `db["posts"]` , como la entrada de un diccionario
 - Cada documento es un **objeto** JavaScript: un diccionario de claves y valores
 - Cada operación es un **método** de la colección: `db.posts.find( ... )`

Una consulta es un documento

```
SELECT Title, Score FROM Posts
WHERE PostTypeId = 1 AND Score ≥ 40
ORDER BY Score DESC LIMIT 10;
```

```
db.posts.find({ PostTypeId: 1, Score: { $gte: 40 } }, // WHERE
               { Title: 1, Score: 1, _id: 0 })        // SELECT
               .sort({ Score: -1 })                  // ORDER BY
               .limit(10)                             // LIMIT
```

- El filtro, la proyección y el orden son **documentos**: datos, no texto
- El primero de `find` selecciona **filas/documentos**; el segundo, **columnas/campos**
- Los operadores (`$gte` , `$in` , `$or` ...) son claves que empiezan por `$`
- Una consulta se construye como cualquier otra estructura de datos, sin concatenar cadenas

Qué se ejecuta y dónde

- `mongosh` evalúa el JavaScript **en el cliente** y convierte cada método en un orden para el servidor, que es otro documento:

```
{ find: "posts", filter: { Score: { $gte: 40 } }, limit: 10 }
```

- El servidor (`mongod`) no interpreta JavaScript: recibe el documento, **planifica** la consulta y elige los índices, como el optimizador de SQL
- `find` es declarativo, como un `SELECT` ; `aggregate` describe una **secuencia de etapas**, más cercana a un plan escrito a mano
- Sólo `$where` , `$function` y el antiguo `mapReduce` ejecutan JavaScript en el servidor: están desaconsejados

Cualquier lenguaje puede ser cliente: basta con saber construir esos documentos

Consultar y modificar documentos

Las operaciones de `mongosh` y su equivalente en SQL

Uso básico de MongoDB

El *shell* permite elegir una base de datos y operar con sus colecciones:

```
db = db.getSiblingDB("pruebas");  
db.personas.insertOne({ _id: 1, nombre: "Ana" });  
db.personas.findOne({ _id: 1 });
```

- **OJO:** Esto es Javascript. Después veremos cómo se hace en Python
- La colección se crea al insertar por primera vez
- No hemos declarado una tabla ni las columnas del documento
- La clave `_id` es la que hemos elegido; si se omite, se genera normalmente un `ObjectId`

Consultar y reemplazar un documento

```
const persona = db.personas.findOne({ _id: 1 });  
// find devuelve un cursor, findOne un documento o null  
db.personas.find({ nombre: "Ana" });
```

```
db.personas.replaceOne(  
  { _id: 1 }, { _id: 1, nombre: "Ana", ciudad: "Murcia" }  
);
```

El reemplazo conserva la identidad y **sustituye el contenido completo**: los campos anteriores que no se incluyan desaparecen

Actualización: modificadores

Para cambiar sólo parte del documento, `updateOne` (o `updateMany`) con un **modificador**:

```
db.analytics.updateOne({"url" : "www.example.com"},  
                        {"$inc" : {"pageviews" : 1}})
```

También `$set`, `$unset`, `$push` (para *arrays*), `$pull` (para eliminar elementos de un *array* que cumplen un criterio)

Combinar condiciones: el mismo álgebra lógica

```
// Y implícito: preguntas que tengan al menos 40 votos
db.posts.find({ PostTypeId: 1, Score: { $gte: 40 } })
// O explícito: muchos votos o muchas visitas
db.posts.find({ $or: [
  { Score: { $gte: 40 } }, { ViewCount: { $gt: 1000 } }
] })
```

- Varias claves de un filtro se combinan con **Y**
- Varias condiciones sobre un mismo campo comparten subdocumento: `{ Score: { $gte: 40, $lte: 100 } }` es **BETWEEN 40 AND 100**
- `$or` y `$and` reciben listas de filtros; se pueden anidar
- El servidor decide cómo ejecutarlo y qué índices usar

Consultar dentro de un documento

```
{ _id: 1, nombre: "Ana", contacto: { ciudad: "Murcia" },  
  etiquetas: ["sql", "python"] }
```

```
db.personas.find({ "contacto.ciudad": "Murcia" })  
db.personas.find({ etiquetas: "sql" })
```

- La notación con punto permite buscar dentro de un subdocumento
- Un filtro por valor puede encontrarlo dentro de un array
- Esas consultas también pueden tener índices: el documento no obliga a leer la colección entera

Ausente no es lo mismo que `null`

```
{ _id: 1, nombre: "Ana" }  
{ _id: 2, nombre: "Luis", web: null }  
{ _id: 3, nombre: "Marta", web: "marta.dev" }
```

```
db.personas.find({ web: { $exists: false } }) // sólo Ana  
db.personas.find({ web: null })              // Ana y Luis
```

La igualdad con `null` incluye aquí el campo ausente; hay que decidir qué significado necesita la aplicación

Valores distintos: `distinct`

```
db.pedidos.distinct("cliente")
```

colección pedidos

```
{"cliente": "A123", "importe": 500, "estado": "A"}
```

```
{"cliente": "A123", "importe": 250, "estado": "A"}
```

```
{"cliente": "B212", "importe": 200, "estado": "A"}
```

```
{"cliente": "A123", "importe": 300, "estado": "D"}
```

distinct
→

resultado

```
["A123", "B212"]
```

4 documentos, 2 valores: cada uno aparece una sola vez en la lista (2 repeticiones descartadas)

De SQL a JavaScript: repaso y equivalencias

SQL	mongosh (JavaScript)
CREATE TABLE Posts (...)	Nada: la colección se crea al insertar
INSERT INTO Posts VALUES (...)	db.posts.insertOne({ ... }), db.posts.insertMany([...])
SELECT * FROM Posts WHERE Id = 4	db.posts.find({ Id: 4 })
SELECT Title FROM Posts	db.posts.find({}, { Title: 1 })
WHERE Score ≥ 40 AND PostTypeId = 1	{ Score: { \$gte: 40 }, PostTypeId: 1 }
ORDER BY Score DESC LIMIT 10	.sort({ Score: -1 }).limit(10)
UPDATE Posts SET Score = 0 WHERE Id = 4	db.posts.updateOne({ Id: 4 }, { \$set: { Score: 0 } })
DELETE FROM Posts WHERE Score < 0	db.posts.deleteMany({ Score: { \$lt: 0 } })
SELECT COUNT(*) FROM Posts WHERE ...	db.posts.countDocuments({ ... })
SELECT DISTINCT OwnerUserId FROM Posts	db.posts.distinct("OwnerUserId")
GROUP BY , HAVING , JOIN	db.posts.aggregate([...]) con \$group , \$match , \$lookup
CREATE INDEX ... ON Posts (Score)	db.posts.createIndex({ Score: 1 })
EXPLAIN SELECT ...	db.posts.find(...).explain()

Índices

Qué consultas se optimizan y cómo se indexan los datos anidados

Índices: el mismo razonamiento que en SQL

```
db.users.find({ Reputation: 100 }).explain("executionStats")
db.users.createIndex({ Reputation: 1 })
db.users.find({ Reputation: 100 }).explain("executionStats")
```

- COLLSCAN : recorrer la colección; IXSCAN : recorrer un índice
- nReturned : documentos devueltos; totalDocsExamined : documentos leídos
- Un índice permite acotar el acceso, pero **mantenerlo cuesta escrituras y espacio**
- Los compuestos se ordenan por campos y prefijos, como vimos en SQL

En prácticas compararemos índices simples y compuestos, antes y después de crearlos

Índices compuestos y datos anidados

```
db.posts.createIndex({ PostTypeId: 1, CreationDate: -1 })
db.posts.find({ PostTypeId: 1 }).sort({ CreationDate: -1 }).limit(10)
```

- Como en SQL, el orden del índice determina sus **prefijos** y ordenaciones útiles
- También se pueden indexar rutas: `{ "contacto.ciudad": 1 }`
- Indexar un array crea entradas para sus elementos: un índice **multikey**
- Índice candidato no significa índice eficiente: mirar claves y documentos examinados

En prácticas mediremos el acceso y el coste de mantener esos índices

EL *framework* de agregación

Un *pipeline* de etapas que transforma los documentos

Framework de agregación: el *pipeline*

```
db.pedidos.aggregate([ Python, con pymongo
  etapa 1 → {"$match": {"estado": "A"}},
  etapa 2 → {"$group": {"_id": "$cliente", "total": {"$sum": "$importe"}}}
])
```

colección pedidos

```
{"cliente": "A123",
"importe": 500,
"estado": "A"}
```

```
{"cliente": "A123",
"importe": 250,
"estado": "A"}
```

```
{"cliente": "B212",
"importe": 200,
"estado": "A"}
```

```
{"cliente": "A123",
"importe": 300,
"estado": "D"}
```

\$match

tras \$match

```
{"cliente": "A123",
"importe": 500,
"estado": "A"}
```

```
{"cliente": "A123",
"importe": 250,
"estado": "A"}
```

```
{"cliente": "B212",
"importe": 200,
"estado": "A"}
```

\$group

tras \$group: resultado

```
{"_id": "A123",
"total": 750}
```

```
{"_id": "B212",
"total": 200}
```

Etapas del *pipeline* y su equivalente en SQL

Un *pipeline* es una lista de etapas: cada una recibe documentos y entrega otros

Etapas	Qué hace	Equivalente SQL
<code>\$match</code>	Filtra	<code>WHERE</code> ; después de agrupar, <code>HAVING</code>
<code>\$project</code>	Elige o calcula campos	Lista de <code>SELECT</code>
<code>\$set</code>	Añade o sustituye campos y conserva los demás	<code>SELECT *</code> más expresiones calculadas
<code>\$group</code>	Agrupar y calcular agregados	<code>GROUP BY</code> , <code>SUM</code> , <code>AVG</code>
<code>\$sort</code> , <code>\$skip</code> , <code>\$limit</code>	Ordena y pagina	<code>ORDER BY</code> , <code>OFFSET</code> , <code>LIMIT</code>
<code>\$lookup</code>	Traer documentos relacionados	<code>LEFT JOIN</code>
<code>\$unwind</code>	Un documento por elemento de un array	Expansión con <code>JSON_TABLE</code>

La idea de agrupar y reducir de la sesión 3 aparece ahora como `$group` y operadores de agregación

Añadir campos dentro del *pipeline*

`$set` añade campos o sustituye sus valores y conserva los demás; `$addField` es un alias

```
db.posts.aggregate([
  { $match: { PostTypeId: 1 } },
  { $set: { PuntuaciónAjustada: { $add: ["$Score", 1] } } },
  { $project: { _id: 0, Id: 1, Score: 1, PuntuaciónAjustada: 1 } },
  { $limit: 3 }
])
```

- `$project` deja sólo los campos indicados
- El `$set` de un *pipeline* transforma los documentos para las etapas siguientes; no escribe en la colección
- El modificador `$set` de `updateOne` sí cambia el documento almacenado

Agregación: seguir los datos etapa a etapa

```
db.posts.aggregate([
  { $match: { PostTypeId: 1 } },
  { $group: { _id: "$OwnerId", preguntas: { $sum: 1 } } },
  { $sort: { preguntas: -1 } },
  { $limit: 5 }
])
```

- Entrada: documentos de posts; tras `$match`, sólo preguntas
- Tras `$group`: un documento por usuario, con su número de preguntas
- Tras `$sort` y `$limit`: los cinco usuarios con mayor número de preguntas
- `"$OwnerId"` es el valor de un campo; `_id` identifica cada grupo

Un pipeline pequeño: entrada y filtro

Datos ilustrativos, reducidos a los campos que utiliza la consulta:

Id	PostTypeId	OwnerUserId
10	1	1
11	1	1
12	2	2
13	1	2

`{ $match: { PostTypeId: 1 } }` deja los posts **10, 11 y 13**

Todavía hay un documento por pregunta; la respuesta 12 ya no pasa a la siguiente etapa

El grupo cambia la forma de los documentos

```
{ $group: { _id: "$OwnerId", preguntas: { $sum: 1 } } }
```

_id (usuario)	preguntas
1	2
2	1

- Tres preguntas de entrada se convierten en dos documentos de salida
- Tras agrupar, `Title` e `Id` ya no están: sólo quedan la clave y los agregados
- Un `$sort` posterior ordena **usuarios agregados**, no los posts originales

Filtrar con `$match`

Se pueden añadir etapas; por ejemplo, **filtrar posts**:

```
{ $match: { Body: { $regex: "HBase" } } } // expresión regular
{ $match: { PostTypeId: 1, Score: { $gte: 10, $lt: 40 } } } // rango
{ $match: { $expr: { $gt: ["$ViewCount", { $multiply: ["$Score", 100] }] } } }
{ $match: { $expr: { $eq: [{ $year: "$CreationDate" }, 2020] } } }
```

- Las dos primeras son la sintaxis de `find`: `$match` admite el mismo álgebra de condiciones
- `$expr` evalúa una **expresión de agregación**: `"$ViewCount"` es el valor del campo y se puede comparar con otro campo o con un valor calculado
- La condición sobre `$year` no aprovecha un índice de `CreationDate`; un rango de fechas con `$gte` y `$lt` sí

Campos calculados con `$project`

```
// Entrada (reducida): { Id: 10, Title: "HBase y Hadoop", Score: 45,  
//                      ViewCount: 900, CreationDate: ISODate("2020-03-05") }  
{ $project: {  
  _id: 0, Id: 1,  
  etiqueta: { $concat: ["#", { $toString: "$Id" }, " · ", "$Title"] },  
  año: { $year: "$CreationDate" },  
  votosPorMil: { $multiply: [{ $divide: ["$Score", "$ViewCount"], 1000 } ],  
  nivel: { $cond: [{ $gte: ["$Score", 40] }, "alto", "normal"] } } }
```

```
{ Id: 10, etiqueta: "#10 · HBase y Hadoop", año: 2020, votosPorMil: 50, nivel: "alto" }
```

- Cada campo calculado es `nombre: expresión`, y las expresiones leen los campos con `"$Campo"`
- `$project` no modifica la colección: sólo da forma a los documentos que pasan a la etapa siguiente

\$project con condiciones y valores ausentes

```
// users (reducido): { Id: 1, DisplayName: "Ana", Reputation: 250, Location: null }
{ $project: {
  _id: 0, DisplayName: 1,
  categoría: { $switch: {
    branches: [ { case: { $gte: ["$Reputation", 1000] }, then: "veterano" },
                 { case: { $gte: ["$Reputation", 100] }, then: "activo" } ],
    default: "nuevo" } },
  ciudad: { $ifNull: ["$Location", "desconocida"] },
  iniciales: { $toUpper: { $substrCP: ["$DisplayName", 0, 1] } } },
{ $match: { categoría: "activo" } }
```

```
{ DisplayName: "Ana", categoría: "activo", ciudad: "desconocida", iniciales: "A" }
```

- `$switch` generaliza `$cond` a varias ramas, como un `CASE` de SQL
- `$ifNull` sustituye tanto `null` como un campo ausente
- El `$match` final filtra por `categoría`, un campo que sólo existe tras el `$project`

El orden de las etapas cambia el trabajo

```
// Filtrar, ordenar y obtener la pregunta de java más antigua
db.posts.aggregate([
  { $match: { PostTypeId: 1, Tags: { $regex: "<java>" } } },
  { $sort: { CreationDate: 1 } },
  { $limit: 1 }
])
```

- `$match` antes de ordenar reduce los documentos que pasan a la etapa siguiente
- `$limit` antes de `$sort` elegiría primero un documento cualquiera: **cambia la respuesta**
- Un índice adecuado puede reducir lectura y ordenación; hay que contrastarlo con el plan

Agrupar y después filtrar: HAVING

```
SELECT OwnerUserId, COUNT(*) AS preguntas FROM Posts
WHERE PostTypeId = 1 GROUP BY OwnerUserId
HAVING COUNT(*) > 10;
```

```
db.posts.aggregate([
  { $match: { PostTypeId: 1 } },
  { $group: { _id: "$OwnerUserId", preguntas: { $sum: 1 } } },
  { $match: { preguntas: { $gt: 10 } } }
])
```

La etapa `$match` es la misma: antes del grupo filtra posts, después filtra usuarios agregados

Unir colecciones: \$lookup

Traer documentos de otra colección y desdoblar el array resultante

\$lookup: el JOIN de MongoDB

`$lookup` permite el acceso a otra (o a la misma) colección: equivalente a un `JOIN`.
Listar los *posts* con `Score` mayor o igual a 40 junto con el usuario que lo ha hecho:

```
db.posts.aggregate([
  { $match: { Score: { $gte: 40 } } },
  { $lookup: { from: "users", localField: "OwnerUserId",
               foreignField: "Id", as: "owner" } }
])
```

- `from` : colección de destino; `localField` , campo del documento actual; `foreignField` , campo con el que se compara en el destino
- `as` : nombre del nuevo campo, siempre un array

\$lookup: resultado y \$unwind

Con una pregunta de Ana y su usuario en otra colección:

```
{ Id: 10, OwnerUserId: 1, owner: [
  { Id: 1, DisplayName: "Ana", Reputation: 250 }
] }
```

- \$lookup añade un **array** de coincidencias; si no encuentra ninguna, es []
- Una referencia no garantiza por sí sola que exista el documento de destino
- Para obtener un documento por coincidencia se añade { \$unwind: "\$owner" }

\$unwind: resultado

```
db.posts.aggregate([
  { $match: { Id: 10 } },
  { $lookup: { from: "users", localField: "OwnerId",
               foreignField: "Id", as: "owner" } },
  { $unwind: "$owner" }
])
```

```
{ Id: 10, OwnerUserId: 1,
  owner: { Id: 1, DisplayName: "Ana", Reputation: 250 } }
```

- Cada coincidencia del array `owner` produce un documento con un subdocumento
- Un array vacío no produce salida; `preserveNullAndEmptyArrays: true` conserva el documento, como un `LEFT JOIN`
- La unión puede multiplicar documentos y alterar agregados posteriores

\$lookup con dos comentarios y con ninguno

Los comentarios están en otra colección, con `PostId` apuntando al post. Datos ilustrativos: el post 10 tiene dos y el 11 ninguno

```
db.posts.aggregate([
  { $match: { Id: { $in: [10, 11] } } },
  { $lookup: { from: "comments", localField: "Id",
               foreignField: "PostId", as: "comments" } }
])
```

```
{ Id: 10, comments: [ { Id: 71, PostId: 10, Text: "¿Qué versión usas?" },
                      { Id: 72, PostId: 10, Text: "Ya funciona, gracias" } ] }
{ Id: 11, comments: [] }
```

- Hay un documento por post, pero el array tiene tantos elementos como coincidencias: dos, cero...
- El array vacío es el resultado normal de «no hay comentarios», no un error

\$unwind con dos documentos y con ninguno

Se añade al *pipeline* anterior una etapa:

```
{ $unwind: "$comments" }
```

```
{ Id: 10, comments: { Id: 71, PostId: 10, Text: "¿Qué versión usas?" } }  
{ Id: 10, comments: { Id: 72, PostId: 10, Text: "Ya funciona, gracias" } }
```

- Dos entradas (posts 10 y 11) producen **dos** salidas, y las dos son del post 10
- El post 10 se repite, una vez por comentario; el post 11, sin comentarios, **desaparece**
- Un `$group` posterior por `Id` cuenta comentarios, pero el 11 no aparece ni con un 0

\$unwind sin perder los posts sin comentarios

```
{ $unwind: { path: "$comments", preserveNullAndEmptyArrays: true } }
```

```
{ Id: 10, comments: { Id: 71, PostId: 10, Text: "¿Qué versión usas?" } }  
{ Id: 10, comments: { Id: 72, PostId: 10, Text: "Ya funciona, gracias" } }  
{ Id: 11 }
```

- Con `preserveNullAndEmptyArrays: true` el post 11 pasa sin el campo `comments`: es el `LEFT JOIN`
- Para contar con un 0, un `$group` posterior no debe contar los documentos, sino los que tienen `comments`

```
{ $group: { _id: "$Id", n: { $sum: { $cond: ["$comments", 1, 0] } } } }
```

\$lookup con un sub-pipeline

```
// Sólo los comentarios con Score positivo: el 72 tiene Score 0
{ $lookup: {
  from: "comments",
  let: { postId: "$Id" },
  pipeline: [ { $match: { $expr: { $eq: ["$PostId", "$$postId"] } } },
              { $match: { Score: { $gt: 0 } } },
              { $project: { _id: 0, Text: 1 } } ],
  as: "comments" } }
```

```
{ Id: 10, comments: [ { Text: "¿Qué versión usas?" } ] }
{ Id: 11, comments: [] }
```

- `let` pasa campos del post al sub-*pipeline*, que los lee como `$$postId`
- Se ejecuta sobre `comments`, una vez por post, con las etapas de siempre
- Es un `JOIN` que filtra y da forma a lo que se une

¿Tiene el array algún elemento?

Tras un `$lookup`, `owner` es `[]` sin coincidencia. Para quedarse con los arrays **no** vacíos:

```
{ $match: { $expr: { $gt: [{ $size: "$owner" }, 0] } } }  
{ $match: { "owner.0": { $exists: true } } }  
{ $unwind: "$owner" }
```

- `$size` exige una longitud exacta: sin `$expr` sólo sirve para `{ $size: 0 }`
- `"owner.0"`: la posición 0 existe sólo si hay algún elemento
- `$unwind` descarta los vacíos, pero devuelve un documento por elemento

Para filtrar basta `$size` o `"owner.0"`; `$unwind`, si se usan los elementos

Arrays dentro del *pipeline*

Calcular con arrays y conjuntos sin desdoblar los documentos

Calcular con un array sin desdoblarlo

```
// Hilo ilustrativo; la última respuesta es de un usuario eliminado
{ _id: 10, Answers: [ { AuthorId: 2 }, { AuthorId: 3 }, { AuthorId: 2 }, {} ],
  Comments: [ { UserId: 1 }, { UserId: 3 } ] }
```

```
{ $project: {
  respuestas: { $size: "$Answers" },           // 4
  autores:    "$Answers.AuthorId",             // [2, 3, 2]
  distintos:  { $setUnion: ["$Answers.AuthorId"] }, // [2, 3]
  numAutores: { $size: { $setUnion: ["$Answers.AuthorId"] } } } // 2
```

- Una ruta sobre un array reúne ese campo de cada elemento y salta los que no lo tienen
- `$setUnion` con un solo array lo trata como **conjunto**: sin repetidos ni orden garantizado
- Con `Answers: []`, `numAutores` vale 0 y el documento no se pierde

Dos arrays como conjuntos

```
// Mismo hilo: autores de las respuestas y de los comentarios
{ $set: { resp: "$Answers.AuthorId", // [2, 3, 2]
        coment: "$Comments.UserId" } }, // [1, 3]
{ $project: {
    participantes: { $setUnion: ["$resp", "$coment"] }, // [1, 2, 3]
    respondeYComenta: { $setIntersection: ["$resp", "$coment"] }, // [3]
    sóloResponde: { $setDifference: ["$resp", "$coment"] } } } // [2]
```

- Son **UNION**, **INTERSECT** y **EXCEPT** de SQL, pero entre dos arrays **del mismo documento**
- El resultado no tiene repetidos ni orden garantizado; **\$size** cuenta sus elementos

\$setUnion o \$addToSet

```
// Dentro de cada documento: un conjunto por hilo
{ $set: { autores: { $setUnion: ["$Answers.AuthorId"] } } }

// Entre documentos: desdoblar y volver a agrupar por hilo
{ $unwind: { path: "$Answers", preserveNullAndEmptyArrays: true } },
{ $group: { _id: "$_id", autores: { $addToSet: "$Answers.AuthorId" } } }
```

- Los dos dan el mismo conjunto por hilo; `$addToSet` también salta los valores ausentes
- El segundo multiplica los documentos y, sin `preserveNullAndEmptyArrays`, pierde los hilos sin respuestas

Dentro del documento, operadores de conjunto; entre documentos, `$group`

De `mongosh` a `pymongo`

Las mismas consultas desde Python, para las prácticas

De JavaScript a pymongo

Los *drivers* ofrecen las mismas operaciones que el *shell*, con las convenciones de cada lenguaje. En Python, los nombres pasan de *camelCase* a *snake_case*:

mongosh (JavaScript)	pymongo (Python)
insertOne , insertMany	insert_one , insert_many
findOne	find_one
updateOne , updateMany , replaceOne	update_one , update_many , replace_one
deleteOne , deleteMany	delete_one , delete_many
countDocuments	count_documents
createIndex , dropIndex	create_index , drop_index
find , aggregate , distinct , watch	Igual: son una sola palabra

Con cualquier ejemplo en JavaScript de la documentación o de un libro, basta aplicar la regla

De JavaScript a pymongo: qué cambia en los documentos

- Un documento es un `dict`. Los operadores y los nombres de los campos **no cambian**, pero las claves se escriben entre comillas: `{ $gte: 40 }` → `{"$gte": 40}`
- `true`, `false`, `null` → `True`, `False`, `None`
- `ISODate("2026-10-08")` → `datetime(2026, 10, 8)`
- Las opciones dejan de ser un documento y pasan a ser argumentos con nombre
- `find` y `aggregate` devuelven un cursor: se recorre con `for` o se convierte en lista con `.to_list()`

```
db.posts.updateOne({ Id: 4 }, { $inc: { Score: 1 } }, { upsert: true })
```

```
db.posts.update_one({"Id": 4}, {"$inc": {"Score": 1}}, upsert=True)
```

\$set en PyMongo

Las etapas del *pipeline* y sus operadores se conservan; `aggregate` recibe una lista de diccionarios Python

```
for documento in db.posts.aggregate([
    {"$match": {"PostTypeId": 1}},
    {"$set": {"PuntuaciónAjustada": {"$add": ["$Score", 1]}}},
    {"$project": {"_id": 0, "Id": 1, "Score": 1, "PuntuaciónAjustada": 1}},
    {"$limit": 3},
]):
    print(documento)
```

- `aggregate` devuelve los documentos transformados en un cursor; el `$set` no modifica la colección

Consistencia y transacciones

De ACID a BASE, y dónde se sitúa MongoDB

ACID frente a BASE

	ACID	BASE
Siglas	Atomicidad, Consistencia, Aislamiento (<i>Isolation</i>), Durabilidad	<i>Basically Available, Soft state, Eventual consistency</i>
Prioridad	Transacción: cambios atómicos y aislamiento según el nivel elegido	Disponibilidad y convergencia: tolerar estados temporales entre réplicas
Tras una escritura	La visibilidad depende del aislamiento y de la instantánea de lectura	Las réplicas convergen al cabo de un tiempo
Quién asume el coste	El sistema mantiene las garantías; entre nodos puede necesitar coordinación	La aplicación: debe tolerar lecturas antiguas y conflictos
En el curso	MySQL con InnoDB (sesión 2)	Cassandra leyendo y escribiendo a ONE (sesión 5)

No son dos bandos, sino los extremos de un eje: cada sistema, y a menudo cada operación, elige su punto

CAP: por qué hay que elegir

Con los datos replicados en varios nodos, si la red se **parte** (*Partition*), un nodo que no alcanza a los demás sólo puede:

- **C** (*Consistency*): rechazar la operación, para no devolver un dato quizá obsoleto ni aceptar una escritura que los demás no verán
- **A** (*Availability*): responder con lo que tiene, y reconciliar después

Y cada sistema toma partido:

- MongoDB se inclina por **C**: sólo el nodo primario acepta escrituras, y sin mayoría no hay primario
- Cassandra se inclina por **A**: cualquier réplica acepta escrituras (sesión 5)
- Mientras la red funciona no hay que renunciar a ninguna, pero se sigue eligiendo entre consistencia y **latencia**: esperar o no a las demás réplicas

MongoDB: el documento es la transacción

- Una escritura sobre **un** documento es siempre atómica, aunque cambie varios campos, subdocumentos y *arrays*
- Es la apuesta de la sesión 3: el agregado como alcance natural de una operación atómica

```
db.cuentas.updateOne(  
  { _id: 1, saldo: { $gte: 100 } },      // la condición y los dos cambios  
  { $inc: { saldo: -100 },                // se aplican juntos, o no se aplican  
    $push: { movimientos: { importe: -100, fecha: new Date() } } })
```

- Por eso lo que debe cambiar junto se **agrega** en un mismo documento
- La transferencia de la sesión 2, entre **dos** cuentas, toca dos documentos: ya no basta

MVCC: leer sin bloquear

- *Multi-Version Concurrency Control*: una escritura no sobrescribe el dato, crea una **versión** nueva
- Cada operación lee una **instantánea** (*snapshot*): las versiones confirmadas cuando empezó
- Los lectores no bloquean a los escritores, ni los escritores a los lectores
- Es la técnica de InnoDB (MySQL) y de PostgreSQL. En MongoDB la aplica su motor de almacenamiento, **WiredTiger**
- En WiredTiger el control de concurrencia es **optimista** y por **documento**:
 - Dos escrituras a documentos distintos no se estorban
 - Dos escrituras al mismo documento: una se aplica y la otra sufre un *write conflict*, que MongoDB reintenta sin que el cliente lo note
- Sobre esa base (atomicidad por documento e instantáneas) se construyen las transacciones

Transacciones multidocumento

- En el diseño inicial: el documento es la transacción
- MongoDB añadió transacciones **multidocumento**: en *replica sets* desde 4.0, en clústeres fragmentados (*sharded*) desde 4.2
- Transferir 100 entre dos cuentas exige debitar una y acreditar otra **en la misma transacción**
- Si falla el segundo cambio, se revierte el primero; si termina, se confirman ambos
- La sesión transaccional debe acompañar a todas las operaciones implicadas

Esta extensión acerca capacidades de SQL y NoSQL; no sustituye el diseño de agregados

Volvemos al hilo de Ana y Luis

La pregunta 10 de Ana (usuario 1), con las respuestas incrustadas (colección `preguntas` ilustrativa):

```
{ _id: 10, autor: 1, respuestas: [ { _id: 21, autor: 2, votos: 12 } ] }
```

Luis (usuario 2) añade otra respuesta:

```
db.preguntas.updateOne({ _id: 10 },  
  { $push: { respuestas: { _id: 22, autor: 2, votos: 0 } } })
```

- Una sola escritura sobre **un** documento: atómica, sin transacción
- Nadie puede leer la pregunta con la respuesta a medio añadir

Ana cambia su nombre: ¿referencia o copia?

- **Referencia:** la pregunta sólo guarda `autor: 1` y el nombre vive en un único documento de `usuarios`

```
db.usuarios.updateOne({ _id: 1 }, { $set: { nombre: "Ana M." } })
```

- **Copia:** cada pregunta de Ana guarda `autor: { _id: 1, nombre: "Ana" }`, y hay que cambiarlas todas

```
db.preguntas.updateMany({ "autor._id": 1 }, { $set: { "autor.nombre": "Ana M." } })
```

- `updateMany` es atómico **por documento**, pero no en conjunto:
 - Mientras dura, se pueden leer preguntas con el nombre nuevo y otras con el viejo
 - Si falla a mitad, las ya cambiadas no se deshacen

Que todas las copias cambien a la vez

```
db = client.foro
with client.start_session() as s:
    with s.start_transaction():
        db.usuarios.update_one({"_id": 1},
                                {"$set": {"nombre": "Ana M."}}, session=s)
        db.preguntas.update_many({"autor._id": 1},
                                   {"$set": {"autor.nombre": "Ana M."}}, session=s)
```

- Transacción multidocumento en `pymongo` : cada operación recibe la sesión (`session=s`); la que no la lleve queda fuera
- Al salir del bloque se confirman las dos; si salta una excepción, ninguna. Hasta entonces, los demás siguen leyendo el nombre viejo

Pregunta 3 de la sesión 3: la unidad atómica es el documento; un cambio que abarca varios necesita una transacción, o aceptar estados intermedios

Transacciones: disponibles, pero no por omisión

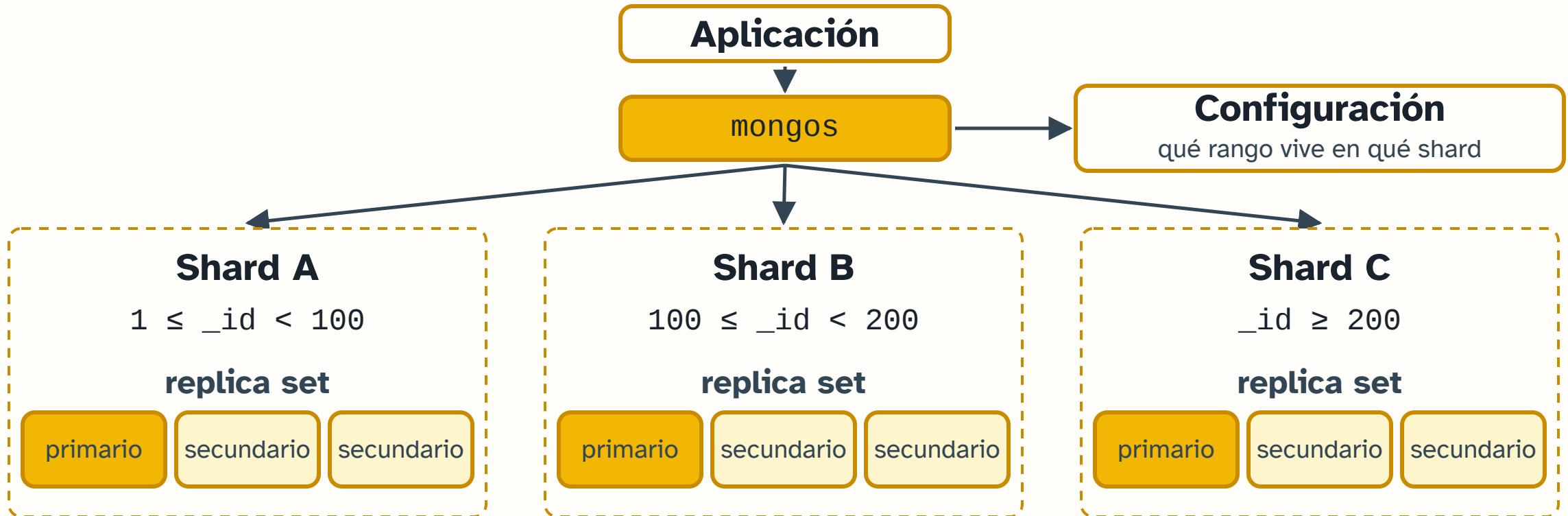
- Exigen un ***replica set*** (aunque sea de un nodo): las prácticas usan un servidor único, así que las vemos aquí y no las ejecutamos
- Son más costosas que escribir un documento, y más aún si abarcan varios *shards*, de los que hablamos a continuación (*commit* en dos fases)
- Están pensadas para ser cortas: por omisión se abortan a los 60 segundos
- Un conflicto de escritura aborta la transacción entera: el código debe poder repetirse

En el modelo relacional se normaliza y la transacción recompone la unidad. En el documental la unidad es el documento, y la transacción multidocumento es la excepción

Fragmentación y replicación

Repartir entre *shards*, replicar cada uno y seguir sus cambios

Fragmentación: repartir el gran array



`db.preguntas.find({ _id: 10 })` → **sólo Shard A**

`db.preguntas.find({ "respuestas.autor": 2 })` → **todos los shards**

Clave de fragmentación: `_id` · Se reparte entre shards, se replica dentro de cada uno

La clave de fragmentación

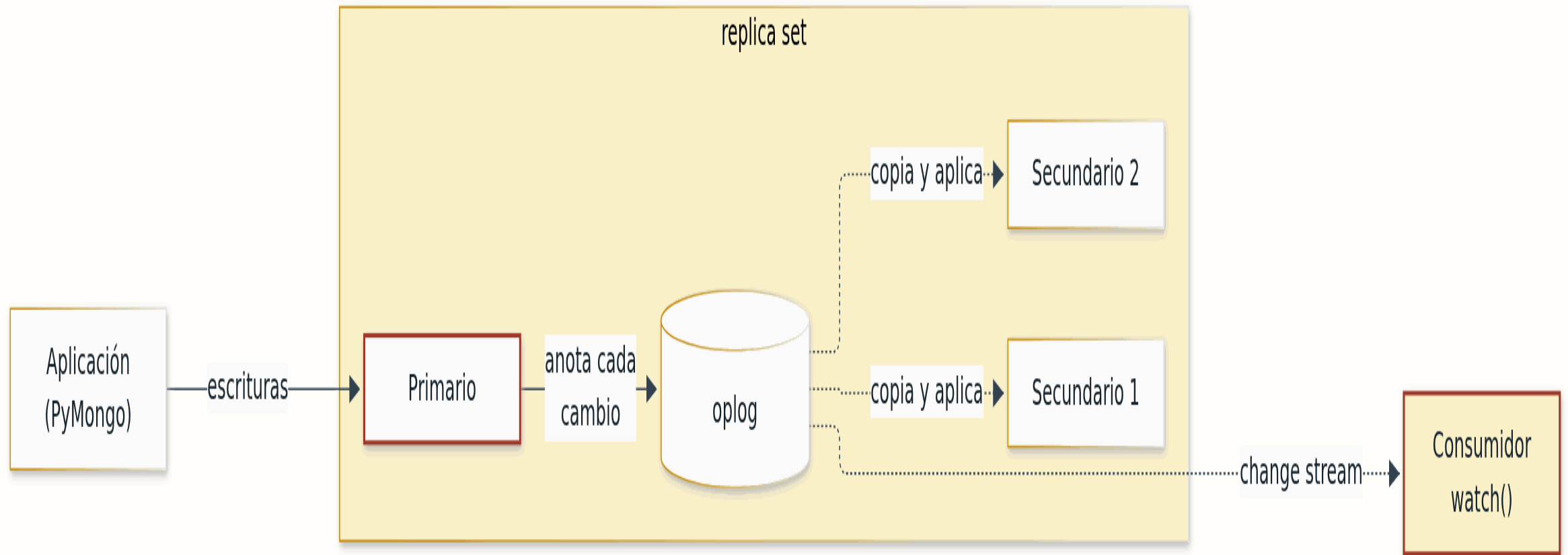
- Es la clave del gran array de la sesión 3: decide en qué *shard* vive cada documento
- Se reparte por **rangos** de la clave o por su **hash**; MongoDB lo divide en *chunks* y un **balanceador** los mueve entre *shards* al crecer el clúster
- Con la clave en el filtro, **mongos** dirige la consulta a un solo *shard*; sin ella, la envía a **todos** y combina los resultados
- Una clave que crece (**ObjectId** , una fecha) y se reparte por rangos manda todas las altas al último *shard*: un **punto caliente**; un *hash* reparte las escrituras, pero los rangos dejan de ser contiguos
- Cambiarla es posible desde la 5.0, pero costoso: se elige con los accesos, no con el formato del documento

Cassandra reparte por el *hash* de la clave de partición y sin enrutadores: lo veremos en la sesión 5

Replicación: el *replica set*

- Un nodo **primario** recibe todas las escrituras; los **secundarios** mantienen copias
- El primario anota cada cambio en el ***oplog***: una colección de tamaño limitado con las operaciones, en orden
- Los secundarios copian el *oplog* y lo aplican: replicación **asíncrona**
- El *oplog* anota efectos, no órdenes: un `$inc` se guarda como el valor resultante, para que aplicarlo dos veces dé lo mismo (**idempotencia**)
- Si el primario cae, los secundarios **eligen** otro por mayoría: por eso se despliegan en número impar

El *oplog*: un registro, varios lectores



Los secundarios y los *change streams* leen el mismo registro de cambios

Consistencia ajustable por operación

Ajuste	Qué decide	Valores habituales
<i>Write concern</i>	Cuántos nodos confirman la escritura antes de responder al cliente	<code>w=1</code> (sólo el primario), <code>w="majority"</code> (por omisión desde MongoDB 5.0)
<i>Read concern</i>	Qué datos se pueden leer	<code>"local"</code> (lo último del nodo; por omisión), <code>"majority"</code> (confirmado por la mayoría), <code>"snapshot"</code>
<i>Read preference</i>	A qué nodo se pregunta	<code>primary</code> (por omisión), <code>secondary</code> , <code>nearest</code>

- Con `w=1`, una escritura confirmada puede **perderse** si el primario cae antes de replicarla
- Leer de un secundario puede devolver un dato **antiguo**: consistencia eventual, por elección

Es el mismo mando que los niveles de consistencia de Cassandra (sesión 5)

Change streams: suscribirse a los cambios

- Una aplicación se **suscribe** a los cambios de una colección, de una base de datos o de todo el despliegue
- En lugar de preguntar cada cierto tiempo (*polling*), el servidor le envía cada `insert`, `update`, `replace` y `delete`
- Los cambios llegan **en orden** y sólo cuando la mayoría los ha confirmado: ya no se pueden deshacer
- `watch()` devuelve un cursor que no se agota: se queda esperando el siguiente cambio

```
with db.personas.watch() as cambios:
    for evento in cambios: # espera al siguiente cambio, de cualquier cliente
        print(evento["operationType"], evento["documentKey"])
```

Las prácticas no lo ejecutan: exige un *replica set*, que no montamos

Change streams: filtrar y reanudar

- `watch` admite etapas de filtrado: se puede escuchar sólo altas de documentos
- Cada evento trae en `_id` un **token de reanudación**
- El consumidor guarda el token después de procesar el evento y lo usa al reconectar
- Sólo puede reanudar si el punto sigue disponible en el *oplog*
- Si procesa un evento y cae antes de guardar el token, puede recibirlo otra vez

La aplicación debe tolerar repeticiones: escuchar cambios también exige diseñar la recuperación

Change streams y replicación

- Un *change stream* es la vista pública del *oplog*: por eso exige un *replica set*, conserva el orden y permite reanudar
- Su consumidor es, en la práctica, **una réplica más**, que decide qué hacer con cada cambio:
 - Mantener un índice externo: así se alimenta `mongot`, el proceso de búsqueda de texto y vectorial de MongoDB
 - Invalidar una caché, avisar a otra aplicación o mantener una colección derivada
 - Copiar los datos a otro sistema: *Change Data Capture* (CDC), que se estudia en TCDM
- Todo lo que cuelga de un *change stream* es **eventualmente consistente** respecto del primario

En Cassandra no hay primario ni un registro único y ordenado: cada nodo tiene su *commit log*. Compararemos su captura de cambios con ésta en la sesión 5

Preparados para las prácticas

- Traducir un filtro y una proyección SQL a documentos de consulta
- Distinguir reemplazar un documento de actualizar algunos campos
- Construir un *pipeline* y explicar cómo cambia su entrada en cada etapa
- Leer el plan antes y después de crear un índice
- Reconocer qué garantiza una escritura atómica por documento y cuándo haría falta una transacción entre documentos

En el boletín trabajaremos con `MongoClient` y cursores; las operaciones y los documentos son los que hemos visto aquí

Referencias

- M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017. Capítulos 2 (modelos de datos), 5 (replicación) y 7 (transacciones)
- D. Pritchett. *BASE: An ACID Alternative*. ACM Queue, 2008
- E. Brewer. *CAP Twelve Years Later: How the "Rules" Have Changed*. IEEE Computer, 2012
- MongoDB. Transacciones, replicación, *change streams* y *pipeline* de agregación
- MongoDB. Equivalencias entre SQL y MongoDB
- PyMongo: referencia del API