Bases de Datos II
Bases de datos basadas en documentos

Diego Sevilla Ruiz & Fernando Terroso Sáenz

Universidad de Murcia & Universidad Politécnica de Cartagena

dsevilla@um.es, fernando.terroso@upct.es

2024

Índice

1. Introducción a las bases de datos de documentos

Bases de datos Documentales

- En general, organizadas como un conjunto de **Colecciones** que contienen **documentos**:
  - Tablas relacionales $\leftrightarrow$ Colecciones
  - Filas (*rows*) $\leftrightarrow$ Documentos
  - No restringen la estructura de cada documento y no tienen esquema
- Cada colección es un *array* donde cada documento tiene asociado un identificador
- La base de datos puede ver el contenido del documento, y utilizar su información como parte de las búsquedas y actualizaciones
- Documentos $\Rightarrow$ formatos jerárquicos tipo JSON o XML
- No utilizan SQL y tienen su lenguaje de consulta propio
- Utilizan normalmente **Map-Reduce** para cálculos distribuidos
- Algunas implementan otros lenguajes de consulta y procesado, como N1QL (Couchbase) y Aggregation Framework (MongoDB)
- Bases de datos de Documentos: Couchbase, MongoDB, OrientDB

2. Modelado de bases de datos de documentos

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:

- Se tiene que tener en cuenta el diseño **distribuido**
- **Optimización guiada por las consultas**

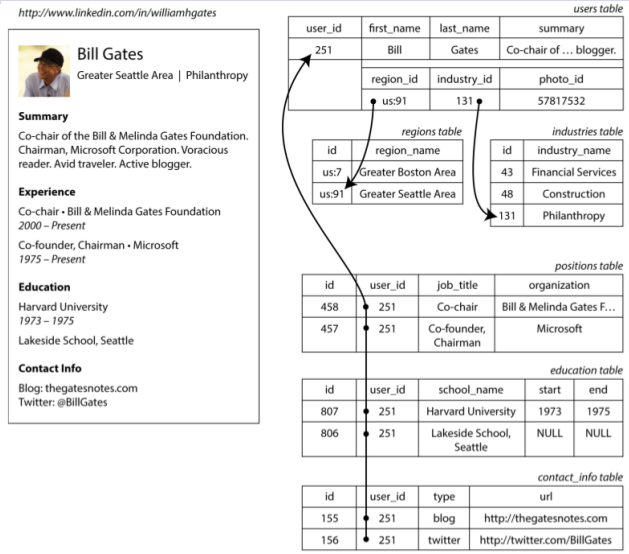
Modelado de datos en NoSQL (cont.)

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**
- Es “barato” **duplicar (desnormalizar)** los datos si con ello se consigue **mayor eficiencia de acceso**

Representación de CV como tablas

Kleppmann, 2016. *Designing Data Intensive Applications*



Representación de relaciones

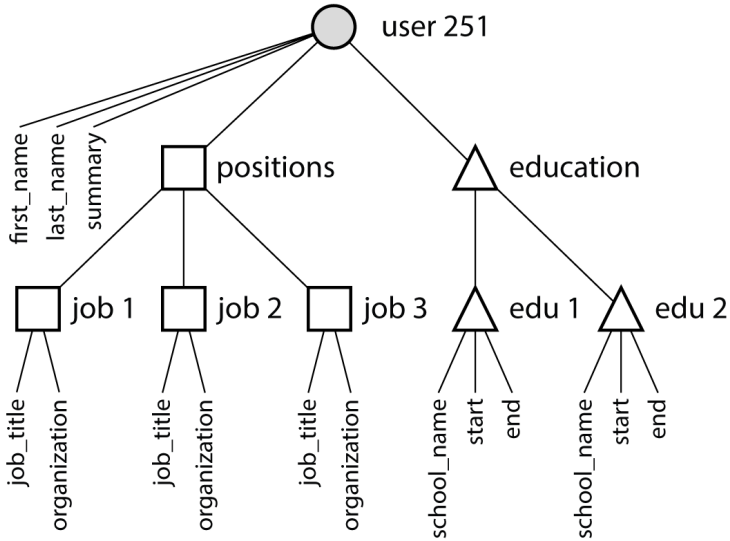
Relaciones uno a muchos

- 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 uso de `JOIN` $\Rightarrow$ ineficiencia
  - Algunos SGBDR ofrecen la posibilidad de tener tipos de datos estructurados y campos XML o JSON. (P. ej. PostgreSQL)
    - Usualmente no se pueden usar para buscar dentro
    - No son estándar

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)



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

Más flexibilidad que el modelo relacional:
(elección entre *referencia* y *agregación*)

Representación de Relaciones (iii)

Uno a muchos (ii) – NoSQL

- Relaciones **Uno a Muchos** (`positions`):
  - **Opción 1**: Agregando la tabla `positions`
  - **Opción 2**: Convertir las empresas en entidades, y utilizar una ***referencia***

Representación de Relaciones (iii)

Uno a muchos (ii) – NoSQL

- Relaciones **Uno a Muchos** (`positions`):
  - **Opción 1**: Agregando la tabla `positions`
  - **Opción 2**: Convertir las empresas en entidades, y utilizar una ***referencia***

¡Modelado guiado por el acceso a datos!

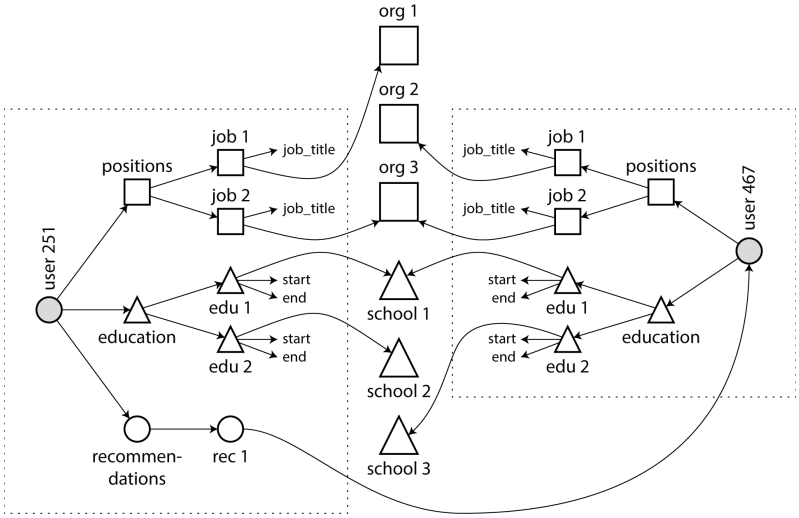
- Si los elementos “muchos” tienen una estructura sencilla ⇒ **Opción 1**
- Si los elementos “muchos” son usualmente **recuperados en una consulta** junto con el elemento “uno” ⇒ **Opción 1**
- Si los elementos “muchos” son relativamente grandes, o bien son recuperados siempre de forma separada ⇒ **Opción 2**

Representación de Relaciones (iv)

Muchos a uno y muchos a muchos

- Las relaciones muchos a uno y muchos a muchos:
  - Personas que viven en una región
  - Preguntas que refieren a Tags
- El modelo de documentos no aporta ventajas con respecto al modelo relacional
- Al haber muchas entidades que refieren a otra entidad, la *agregación* daría lugar a mucha **duplicación** (y a problemas de sincronización)
- $\Rightarrow$ **Referencias** (sobre el ID), similar a una FK en el modelo relacional
  - **Sin embargo**, al no haber **JOINs** la aplicación tiene que hacer más de una petición a la BD

Muchos a muchos – referencia



3. Introducción a MongoDB

Introducción

Introducción a MongoDB

- Base de datos documental
- Basado en una representación binaria de JSON, llamada BSON
- Cada base de datos se divide en un conjunto de *colecciones*
- Una colección es un conjunto arbitrariamente grande de documentos
- Cada documento contiene un campo especial `_id`, que guarda un objeto de tipo especial `ObjectID` de BSON
- Permite la definición de bases de datos, colecciones (tablas *hash* de documentos) y subcolecciones
- Permite realizar búsquedas secuenciales, programas MapReduce y un API propietario de consulta (llamado *aggregate*)
- Ofrece un *shell* que acepta JavaScript y permite realizar operaciones directamente sobre la base de datos
- También ofrece clientes en muchos lenguajes de programación

Uso básico

Uso básico de MongoDB

- MongoDB ofrece un *shell*, aparte de las posibles conexiones remotas a través de *drivers* usando otros lenguajes de programación
- El shell acepta código JavaScript
- Veremos inicialmente su uso
- Existen una serie de objetos predefinidos:

```
show databases;
use <database>;
db.<collection>; // Permite acceder a la colección
```

- Simplemente nombrando las colecciones se puede acceder a ellas para consulta y actualización
- Inserción de documentos:

```
db.colección.insert({< Objeto JSON >});
```

Actualización y consulta

- Se puede obtener los documentos de la base de datos con:

```
var objeto = db.colección.findOne({ atributos : valores})
// ó encontrar todos los que cumplen la condición:
var objs = db.colección.find({ atributos : valores})
```

- Y actualizarlos:

```
db.colección.update(
     { atributos: valores},// búsqueda
     objeto);              // nuevo objeto
```

- La selección del objeto se suele hacer por su `_id`, que es único

Actualización y consulta (cont.)

- También permite modificadores:

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

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

Consulta

- Las consultas se hacen con `find`:

```
db.users.find({<criterio>},
              {"username" : 1, "_id" : 0})
// =>
{
"username" : "joe",
}
```

- Y se pueden utilizar condicionantes para la búsqueda:

```
db.users.find({"age": {"$gte": 18, "$lte": 30}});
db.users.find({"$or": [ {"age": {"$gte": 18} },
                        {"permission": true} ] });
```

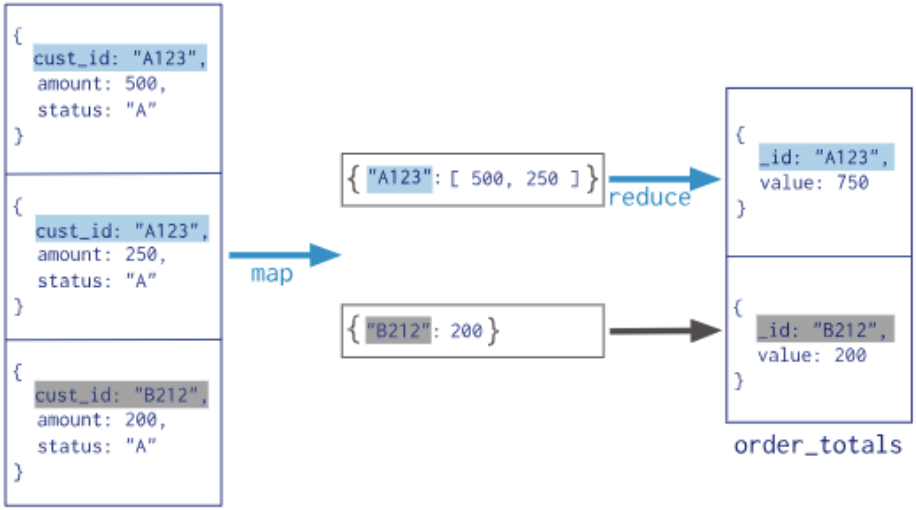
Consultas MapReduce

Consultas MapReduce

- *Map-reduce* es un paradigma de procesamiento de datos que permite condensar grandes volúmenes de datos en resultados agregados útiles.
- Una operación map-reduce consta de tres fases:
  1. Se aplica la función `Map (Map(k1,v1)` $\rightarrow$ `list(k2,v2))` a cada documento de entrada. La función Map se aplica en paralelo a cada documento (con clave `k1`) del conjunto de datos de entrada. Esto produce una lista de pares (codificados por `k2`) para cada llamada.
  2. La función `Reduce (Reduce(k2, list (v2))` $\rightarrow$ `list((k3, v3)))` se aplica entonces en paralelo a cada grupo.

Consultas MapReduce

Ejemplo de aplicación



Validación

Validación del esquema

- En sus últimas versiones MongoDB ha incorporado funcionalidades que permiten anotar y validar documentos JSON (*JSON Schema*)
- A la hora de crear una colección, podemos especificar en *JSON schema* qué configuración deben seguir los documentos almacenados en ella

Validación del esquema – Ejemplo

- Vamos a crear una colección `estudiantes` y vamos a especificar su esquema asociado

```
db.createCollection("estudiantes", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Validacion del objeto Estudiante",
         required: [ "direccion", "major", "nombre", "año" ],
         properties: {
            nombre: {
               bsonType: "string",
               description: "'nombre' debe de ser una cadena (string) y es
                   obligatorio"
            },
```

Validación del esquema – Ejemplo (cont.)

```
            año: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "'año' debe ser un entero entre [ 2017, 3017 ] y es
                   obligatorio"
            },
            gpa: {
               bsonType: [ "double" ],
               description: "'gpa' debe de ser un double si el campo existe"
            }
         }
      }
   }
} )
```

Validación del esquema – Ejemplo (cont.)

- Si ahora intentamos insertar un documento que no cumple con dicho esquema...

```
db.estudiantes.insertOne( {
   nombre: "Alice",
   year: Int32( 2019 ),
   major: "Historia",
   gpa: Int32(3),
   direccion: {
      city: "USA",
      street: "Fraggel Rock"
   }
} )
```

Validación del esquema – Ejemplo (cont.)

- …obtendremos un mensaje de error pues el campo `gpa` se ha insertado como `Int` cuando debía hacerse como `double`

```
MongoServerError: Document failed validation
Additional information: {
  failingDocumentId: ObjectId("630d093a931191850b40d0a9"),
  details: {
    operatorName: '$jsonSchema',
    title: 'Validacion del objeto Estudiante',
    schemaRulesNotSatisfied: [
      {
        operatorName: 'properties',
        propertiesNotSatisfied: [
          {
            propertyName: 'gpa',
            description: "'gpa' debe de ser un doble si el campo existe",
            details: [
              {
                operatorName: 'bsonType',
                specifiedAs: { bsonType: [ 'double' ] },
                reason: 'type did not match',
                consideredValue: 3,
                consideredType: 'int'
              }
            ] } ] } ]} }
```

Validación del esquema – Ejemplo (cont.)

- Sin embargo, la siguiente inserción no provoca ningún fallo

```
db.students.insertOne( {
   name: "Alice",
   year: NumberInt(2019),
   major: "History",
   gpa: Double(3.0),
   address: {
      city: "NYC",
      street: "33rd Street"
   }
} )
```

Índices y desnormalización

Data Access Hitting a Wall

Current practice based on data download (FTP/GREP)
Will not scale to the datasets of tomorrow

- You can GREP 1 MB in a second
- You can GREP 1 GB in a minute
- You can GREP 1 TB in 2 days
- You can GREP 1 PB in 3 years.

- Oh!, and 1PB ~5,000 disks

- At some point you need
  indices to limit search
  parallel data search and analysis
- This is where databases can help

- You can FTP 1 MB in 1 sec
- You can FTP 1 GB / min (~1$)
- … 2 days and 1K$
- … 3 years and 1M$

[slide src: Jim Gray]

Índices

- Sea con SQL o con datos *raw*, la búsqueda no se puede realizar secuencialmente
- Los ***índices*** se utilizan para acelerar esta búsqueda
- El concepto de índice está presente en casi todas las bases de datos (también es un eje principal en las NoSQL)
- En el ámbito del SQL, los índices se pueden aplicar a los valores de una o más columnas
- Se usan para acelerar la recuperación de información
- El planificador de consultas usa los índices disponibles
- Es **labor del usuario** definir los índices adecuados
- Los índices aceleran las búsquedas, pero también ocupan espacio
- En SQL se utiliza la construcción `CREATE INDEX`, `ALTER TABLE` o se incluyen en `CREATE TABLE`

Índices (cont.)

- EXPLAIN para ver si se usa:

```
mysql> explain select * from Posts where Id=5;
+----+-------------+-------+------+---------------+------+...
| id | select_type | table | type | possible_keys | key  |...
+----+-------------+-------+------+---------------+------+...
|  1 | SIMPLE      | Posts | ref  | Id            | Id   |...
+----+-------------+-------+------+---------------+------+...
```

- Obviamente **TAMBIÉN** se usan en las bases de datos NoSQL

Índices (cont.)

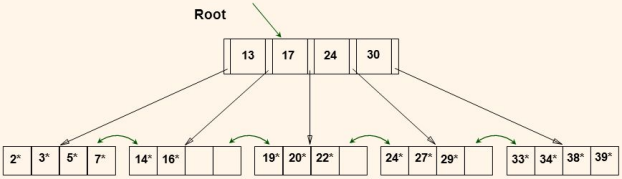
- En general, hay tres tipos de índices:
  1. Basados en árboles balanceados
  2. Basados en tablas *hash*
  3. Índices *full-text*

Índices de Árboles Balanceados

- Mantienen la clave ordenada dentro de un árbol balanceado (B-Tree, B+, etc.)
- Aceleran búsquedas con operadores =, < y > y rangos (`BETWEEN`)
- También `LIKE` que no empiecen con comodín
- Puede ayudar en `ORDER BY`

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



* *Based on the search for 15*, we know it is not in the tree!*

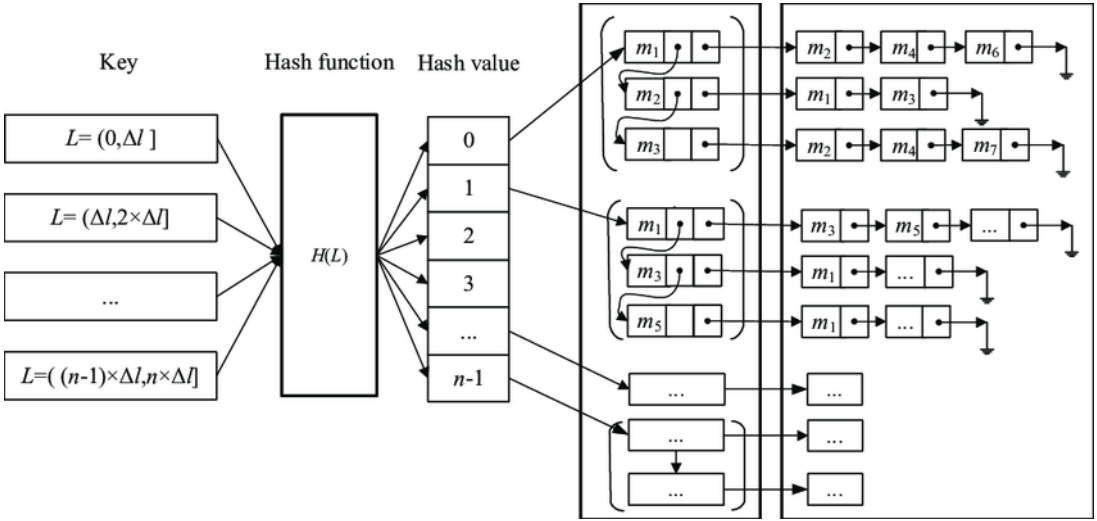
Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke 10

Índices (cont.)

Índices de tablas hash

- Sólo para consultas de igualdad (=) y desigualdad (<=>)
- Son muy rápidos
- No se pueden usar para `ORDER BY`
- Sólo se pueden usar con valores completos de clave (no permiten prefijos, por ejemplo)

Índices (cont.)

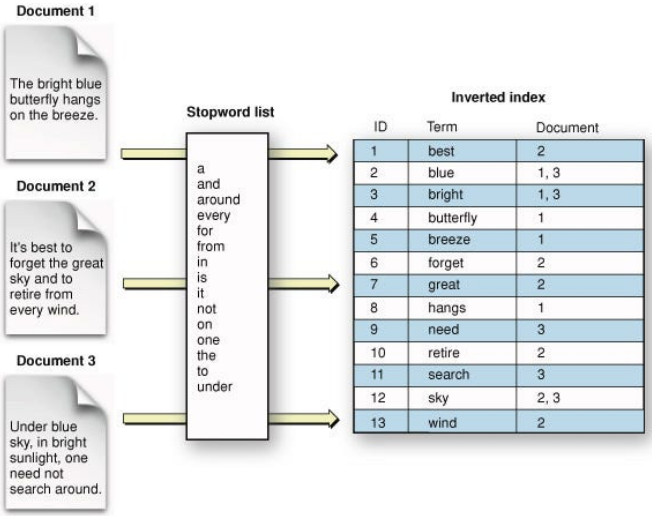


Índices (cont.)

Índices *full-text*

- Usados en búsquedas inversas sobre campos textuales
- Dada una palabra o conjunto de palabras, encontrar la fila o filas en las que el campo buscado (el del índice) contiene esas palabras
- Tiene limitaciones en `LIKE` con comodines (`%`)
- Por ello los SGBD proveen operadores especiales (p. ej. en MySQL "`MATCH(...) AGAINST ...`")

Índices (cont.)



Desnormalización

- El modelo relacional basa el modelado de datos en la **normalización**, en sus distintos niveles
- Esto permite aplicar el principio de “sólo almacenar un dato en un sitio”, lo que lleva a la eficiencia de datos
- Pero a veces necesitamos también **eficiencia de acceso**, y a veces está reñido con la normalización
- El principal escollo para la velocidad es la unión (JOIN) de datos de tablas

Desnormalización (cont.)

- ¿Por qué desnormalizar?
  - **Mantener la historia** – Los datos de los clientes, por ejemplo, pueden cambiar. Necesitamos almacenar los valores de los mismos, por ejemplo, cuando emitimos una factura en el pasado. Para eso hay que copiar los datos de un cliente en ese momento
  - **Mejorar la velocidad de consulta** – A veces, para hacer consultas recurrentes, tenemos que usar uniones de varias tablas. Evitar la unión, por ejemplo, añadiendo claves ajenas a las tablas finales. O bien, como en el caso de `Posts.Tags`, replicando la información
  - **Precalcular valores necesitados de antemano** – Si hay datos que se necesitarán, como medias o totales, podemos calcularlos y almacenarlos de antemano

Desnormalización (cont.)

- Desventajas de desnormalizar:
  - **Espacio en disco** – Al igual que ocurría con los índices, la copia y duplicación de datos ocupa más espacio
  - **Anomalías de datos** – Hay que ser conscientes de que ahora un cambio en un dato puede requerir un cambio en más de una parte de la base de datos para mantener la consistencia. Los SGBDR no están preparados para mantener este tipo de consistencia. Se tienen que utilizar mecanismos de *triggers*, transacciones, etc.
  - **Ralentización de las operaciones de actualización** – Dependiendo del *ratio* entre actualizaciones y consultas, este puede ser un punto importante
  - **Más código, más dependencias, más propenso a errores** – Al necesitar más coordinación, la base de datos requiere más código, hay más dependencias ocultas, lo que puede llevar a más errores y a un mayor coste de mantenimiento

Desnormalización (cont.)

- Ejemplos vistos en Stackoverflow:
  - En la tabla `Posts` tenemos, por ejemplo, el campo `Tags`, que contiene la lista de los tags. Evita tener que hacer un JOIN para obtener los nombres de los tags
  - También está el campo `Score`, que aglutina todos los votos positivos recibidos (también en la tabla `Votes`)

4. Uso de MongoDB desde pymongo

Métodos de búsqueda

Métodos de búsqueda

- El método de búsqueda principal es `find()`, que tiene muchas opciones
- En general permite especificar:
  - El filtro de búsqueda
  - Ordenación de resultados por algún campo
  - Proyección para no obtener todos los campos del documento
  - Número de resultados máximo (*limit*)
  - Número de elementos iniciales a ignorar (*skip*)
  - El tamaño del *batch*
- Como la variabilidad es muy grande, veremos ejemplos en la hoja Jupyter Notebook de prácticas y también en la documentación

Métodos de búsqueda

- La función `find()` tiene un gran número de posibilidades para especificar la búsqueda. Se pueden utilizar cualificadores complejos como:
  - `$and`
  - `$or`
  - `$not`
- Estos calificadores unen “objetos”, no valores

Métodos de búsqueda (cont.)

- Por otro lado, hay otros calificadores que se refieren a valores:
  - `$lt` (menor)
  - `$lte` (menor o igual)
  - `$gt` (mayor)
  - `$gte` (mayor o igual)
  - `$regex` (expresión regular)

```
posts.find_one({"body": {"$regex" : "[Mm]ongo"}})
```

Condición compuesta (limitando la salida a 10 documentos):

```
posts.find({"$and": [{"PostTypeId": 2},
                     {"Id" : {"$gte" : 100}} ]})
          .limit(10)
```

Métodos de búsqueda

Ejemplos introductorios

- Encontrar el primer documento en la colección `customers`

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.find_one()
```

- Encontrar el todos los documentos en la colección `customers`

```
for x in mycol.find():
  print(x)
```

- También podemos indicar qué campos queremos obtener de una determinada búsqueda

```
for x in mycol.find({},{ "_id": 0, "name": 1, "address": 1 }):
  print(x)
```

Métodos de búsqueda

Ejemplos básicos

- Dada la siguiente colección,

```
{ "_id": "manzanas", "cantidad": 5 }
{ "_id": "plátanos", "cantidad": 7 }
{ "_id": "naranjas", "cantidad": 8}
{ "_id": "aguacate", "cantidad": 14}
```

- Si quisieramos obtener aquellos documentos con un valor en su atributo `cantidad` superior a 4 deberísmos ejecutar el siguiente comando

```
db.collection.find( { "cantidad": { "$gt": 4 } } )
```

- ¿Qué comando deberíamos lanzar para obtener aquellos que tengan un valor inferior o igual a 5?

Métodos de búsqueda (cont.)

Ejemplos básicos

- Suponiendo una colección `bios` cuyos documentos tienen esta forma:

```
{
    "_id" : <valor>,
    "nombre" : { "nombre_propio" : <string>, "primer_apellido" : <string> },
    "nacimiento" : <ISODate>,
    "muerte" : <ISODate>,
    "profesion": <valor>,
    "trabajos": [<string>,...]
}
```

Métodos de búsqueda (cont.)

Ejemplos básicos

- Una opción interesante es el filtrado por rango de fechas en caso de que tengamos atributos de ese tipo

```
db.bios.find( { "nacimiento": { "$gt": new Date('1940-01-01'),
 "$lt": new Date('1960-01-01') } } )
```

- El operator `$in` nos permite buscar documentos con atributos dentro de una lista de valores

```
db.bios.find(
   { "profesion": { "$in": [ 'jedi', 'samurai' ] } }
)
```

Métodos de búsqueda (cont.)

Ejemplos básicos

- También podemos realizar búsquedas en campos anidados

```
db.bios.find( { "nombre.primer_apellido": "Skywalker" } )
db.bios.find(
    { "nombre": { "nombre_propio": "Akira", "primer_apellido": "Kurosawa" } }
)
```

- Las búsquedas pueden realizarse también sobre campos cuyos valores son arrays

```
db.bios.find({"trabajos": "star_wars"})
db.bios.find({"trabajos": { "$in": [ "star wars", "alien" ]} })
db.bios.find({"trabajos": { "$all": [ "star wars", "the clone wars" ]}})
db.bios.find({"trabajos": { "$size": 4 }}) #El array 'trabajos' debe tener 4
    elementos
```

Métodos de búsqueda (cont.)

Ejemplos básicos

- El operador `$not` es útil cuando queremos buscar elementos que NO cumplan una condición

```
# Buscar documentos cuyo precio no sea superior a 1,99€ o que no tengan el campo
    "precio"
db.inventory.find( { "precio": { "$not": { "$gt": 1.99 } } } )
```

- En el ejemplo de arriba, ¿cuál es ls diferencia respecto a usar la condición `{"$gt": 1.99}`?

Métodos de búsqueda (cont.)

Ejemplos básicos

- Por último, los operadores `$and`, `$or` y `$not` pueden combinarse para definir condiciones complejas

```
db.inventory.find( {
    "$and": [
        {"$or":[{"cantidad":{"$lt":10}}, {"cantidad":{"$gt":50} }]},
        {"$or":[{"vendido": true}, {"precio": {"$lt": 5}}]}
    ]
} )
```

- ¿Qué documentos devolverá la consulta de arriba?

Ejercicio métodos de búsqueda

- Dada la siguiente colección llamada `libros` en la base de datos `biblioteca`

```
{
  "_id": "1",
  "titulo": "Cien años de soledad",
  "autor": "Gabriel García Márquez",
  "genero": ["Realismo mágico", "Literatura latinoamericana"],
  "anio_publicacion": 1967,
  "precio": 15.99,
  "tags": ["Ciencia ficción", "Distopía", "Política"]
},
{
  "_id": "2",
  "titulo": "El Principito",
  "autor": "Antoine de Saint-Exupéry",
  "genero": ["Fantasía", "Literatura infantil"],
  "anio_publicacion": 1943,
  "precio": 10.99,
  "tags": ["Fantasía", "Aventura", "Amistad"]
},
...
```

Ejercicio métodos de búsqueda (cont.)

- Escribir el código Python que permita realizar las siguientes búsquedas:
  - **Ejercicio 1**: Encontrar todos los libros que sean de genero 'Fantasía' y tengan un precio mayor a 15 euros
  - **Ejercicio 2**: Encontrar todos los libros que tengan el tag 'Fantasía' y que se hayan publicado después de 1950
  - **Ejercicio 3**: Encontrar todos los libros que tengan el tag 'Fantasía' y 'Amistad' o que se hayan publicado a partir de 1982 (inclusive)

Métodos de inserción y actualización

Métodos de inserción y actualización

- `pymongo` ofrece métodos para inserción y actualización:
  - `insert_one()`, `insert_many()` (batch)
  - `update_one()` – Permite actualizar un objeto con nuevos campos. El objeto se crea si se pone el parámetro `upsert` a `True`
  - `update_many()` – Permite poner nuevos valores calculados a un conjunto de objetos

Métodos de inserción y actualización

Ejemplos básicos de inserción

- A la hora de insertar un elemento podemos obtener su `_id` asignado

```
micoleccion = db["clientes"]
midict = { "nombre": "Luke", "direccion": "Tatooine, 83" }
x = mycol.insert_one(midict)
print(x.inserted_id)
```

- Para insertar un conjunto de documentos a la vez deberemos incluirlos en un array JSON

```
milista = [
  { "nombre": "Amy", "direccion": "Apple st 652"},
  { "nombre": "Hannah", "direccion": "Mountain 21"},
  { "nombre": "Michael", "direccion": "Valley 345"}
]
x = micoleccion.insert_many(milista)
#imprimimos us _id asignados
print(x.inserted_ids)
```

Métodos de inserción y actualización

Ejemplos básicos de actualización

- A la hora de actualizar un documento debemos especificar un criterio de búsqueda y los campos y valores a actualizar

```
micoleccion = db["clientes"]
mifiltro = { "direccion": "Valley 345" }
nuevosvalores = { "$set": { "direccion": "Rue del percebe, 13" } }
micoleccion.update_one(mifiltro, nuevosvalores)
```

- Para actualizar múltiples documentos la estructura es la misma

```
micoleccion = db["clientes"]
mifiltro = { "direccion": { "$regex": "^S" } }
nuevosvalores = { "$set": { "nombre": "Rey" } }
x = micoleccion.update_many(mifiltro, nuevosvalores)
print(x.modified_count, "documentos actualizados.")
```

Índices, `explain()`

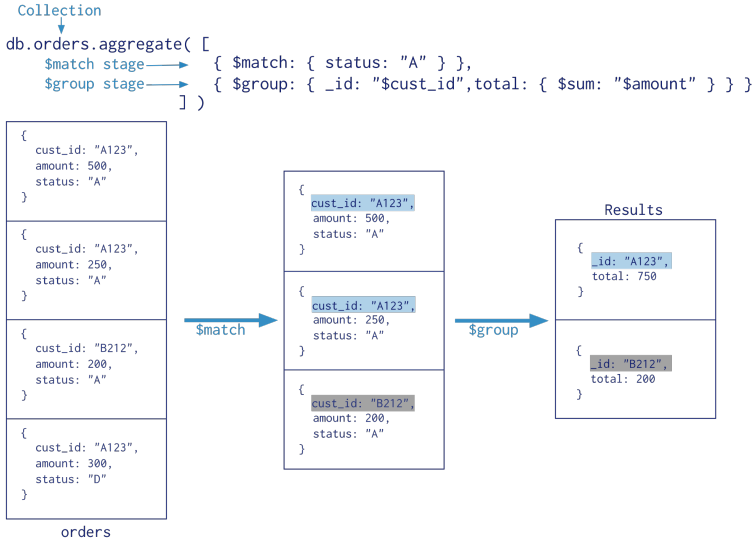
- La llamada `.explain()` a cualquier búsqueda muestra el plan de ejecución
- Se puede crear un índice si la búsqueda por ese campo va a ser crítica
- Se pueden crear más índices:
  - `ASCENDING`
  - `DESCENDING`
  - `HASHED`
  - y geoespaciales

Framework de agregación

Framework de agregación

- El framework de agregación de MongoDB es una de sus características más potentes
- Permite escribir expresiones, divididas en una serie de etapas, que realizan operaciones como agregaciones, transformaciones y uniones en los datos de las bases de datos MongoDB
- Esto le permite realizar cálculos y análisis a través de documentos y colecciones dentro de su base de datos MongoDB

Framework de agregación



Framework de agregación

Framework de agregación:
https://docs.mongodb.com/manual/reference/operator/aggregation/

```
posts.aggregate( [
 {'$project' : { 'Id' : 1 }},
 {'$limit': 20}
])
```

Framework de agregación (cont.)

```
by_length = posts.aggregate( [
  {'$project': {
    'Body' : {'$ifNull' : ['$Body', '']}
  }},
  {'$project' : {
    'id' : {'$strLenBytes': '$Body'},
    'value' : {'$literal' : 1}
  }},
  {'$group' : {
    '_id' : '$id',
    'count' : {'$sum' : '$value'}
  }},
  {'$sort' : { '_id' : 1}}
  ])
```

Framework de agregación (cont.)

Se pueden añadir etapas. En particular, por ejemplo, se puede **filtrar** inicialmente añadiendo al principio:

```
{'$match': { 'Body' : {'$regex': 'HBase'}}}
```

La construcción `$lookup` permite el acceso a otra (o a la misma) colección. Es equivalente a un **JOIN**

Framework de agregación (cont.)

P. ej., listar los posts con Score mayor o igual a 40 junto con el usuario que lo ha hecho:

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

Framework de agregación

Colección de ejemplo

- Vamos a suponer que contamos con una colección de películas (`movies`) que contienen documentos que tienen esta estructura:

```
{
    '_id': ObjectId('573a1392f29313caabcdb497'),
    'awards': {'nominations': 7,
               'text': 'Won 1 Oscar. Another 2 wins & 7 nominations.',
               'wins': 3},
    'cast': ['Janet Gaynor', 'Fredric March', 'Adolphe Menjou', 'May Robson'],
    'countries': ['USA'],
    'directors': ['William A. Wellman', 'Jack Conway'],
    'fullplot': 'Esther Blodgett is just...
    'genres': ['Drama'],
    'imdb': {'id': 29606, 'rating': 7.7, 'votes': 5005},
    'languages': ['English'],
    'lastupdated': '2015-09-01 00:55:54.333000000',
    'plot': 'A young woman ...',
    'poster': 'https://m.media-amazon.com/images/M/MV5....jpg',
    'rated': 'NOT RATED',
    'released': datetime.datetime(1937, 4, 27, 0, 0),
    'runtime': 111,
    'title': 'A Star Is Born',
    'tomatoes': {'critic': {'meter': 100, 'numReviews': 11, 'rating': 7.4},
                 'dvd': datetime.datetime(2004, 11, 16, 0, 0),
```

Framework de agregación (cont.)

Colección de ejemplo

```
            'fresh': 11,
            'lastUpdated': datetime.datetime(2015, 8, 26, 18, 58, 34),
            'production': 'Image Entertainment Inc.',
            'rotten': 0,
            'viewer': {'meter': 79, 'numReviews': 2526, 'rating': 3.6},
            'website': 'http://www.vcientertainment.com/Film-Categories?product_id=73'},
 'type': 'movie',
 'writers': ['Dorothy Parker (screen play)',
             'Alan Campbell (screen play)',
             'Robert Carson (screen play)',
             'William A. Wellman (from a story by)',
             'Robert Carson (from a story by)'],
 'year': 1937}
```

Framework de agregación – Ejemplo (i)

- En primer lugar vamos a realizar una primera búsqueda sobre dicha colección filtrando por el nombre de la película (`title`) y ordenando por su año (`year`):

```
movie_collection = db["movies"]
pipeline = [
   {
      "$match": {
         "title": "A Star Is Born"
      }
   },
   {
      "$sort": {
         "year": pymongo.ASCENDING
      }
   },
]
results = movie_collection.aggregate(pipeline)
```

Framework de agregación – Ejemplo (i) (cont.)

- Si ahora quisieramos limitar la salida a un único documento...

```
pipeline = [
   {
      "$match": {
         "title": "A Star Is Born"
      }
   },
   {
      "$sort": {
         "year": pymongo.ASCENDING
      }
   },
   { "$limit": 1 }
]
results = movie_collection.aggregate(pipeline)
```

Framework de agregación – Ejemplo (ii)

- Vamos a suponer ahora que existe otra colección llamada `comments` cuyos documentos tienen este formato

```
{
    '_id': ObjectId('5a9427648b0beebeb69579d3'),
    'movie_id': ObjectId('573a1390f29313caabcd4217'),
    'date': datetime.datetime(1983, 4, 27, 20, 39, 15),
    'email': 'cameron_duran@fakegmail.com',
    'name': 'Cameron Duran',
    'text': '...'}
```

Framework de agregación – Ejemplo (ii) (cont.)

- Esto permitiría añadir un campo `related_comments` a cada película con sus críticas asociadas

```
stage_lookup_comments = {
   "$lookup": {
         "from": "comments",
         "localField": "_id",
         "foreignField": "movie_id",
         "as": "related_comments",
   }
}
# Limitamos a los primeros 5 documentos
stage_limit_5 = { "$limit": 5 }
pipeline = [stage_lookup_comments, stage_limit_5]
results = movie_collection.aggregate(pipeline)
```

Framework de agregación – Ejemplo (iii)

- Una funcionalidad muy interesante es la agrupación de documentos con el comando `$group`
- Vamos a usar dicho comando para contar el número de películas por año, ordendas de forma ascendente

```
stage_group_year = {
   "$group": {
         "_id": "$year",
         # contamos el número de películas en el grupo
         "movie_count": { "$sum": 1 },
   }
}
stage_match_years = {
   "$match": {
         "year": {
            "$type": "number",
         }
```

Framework de agregación – Ejemplo (iii) (cont.)

```
    }
}
stage_sort_year_ascending = {
  "$sort": {"_id": pymongo.ASCENDING}
}
pipeline = [stage_match_years, stage_group_year,stage_sort_year_ascending]
results = movie_collection.aggregate(pipeline)
```

Índices

Creación de índices

- En MongoDB, la creacion de índices se realiza mediante el método `create_index`

```
db.collection.create_index([(<key and index type specification>)], <options> )
```

- En el siguiente ejemplo se un indice con una sola clave (`nombre`) en orden desdendente

```
1 collection.create_index([("nombre", pymongo.DESCENDING)])
```

- Podemos listar los índices que tenemos creados en una colección con

```
1 collection.index_information()
```

- En el caso de las colecciones con una alta proporción de escritura por lectura, los índices son caros porque cada inserción y actualización también debe actualizar cualquier índice.

Nombrado y borrado de índices

- Al crear un índice, puede darle un nombre personalizado.
- Ayuda a distinguir los distintos índices de la colección.
- Para especificar el nombre del índice, incluya la opción de nombre al crear el índice:

```
db.collection.create_index(
   { "<campo>": "<valor>" },
   { "name": "<NombreDelIndice>" }
)
```

- Para borrar un índice podemos hacerlo con el método `.drop_index`

```
collection.drop_index("<nombreDelIndice>")
```

Tipos de índices en MongoDB

Indice de campo único (*single field index*)

- Recogen y ordenan los datos de un solo campo de cada documento de una colección
- Por defecto, todas las coleciones tienen un índice de este tipo sobre su campo "`_id`"
- Es posible crear un *single field index* en cualquier campo de un documento (incluyendo los campos de documentos incrustados)
- Cuando se crea un índice, se especifica
  - El campo sobre el que crear el índice
  - El orden de ordenación de los valores indexados, ascendente (1) o descendente (-1)

```
db.<collection>.create_index( { "<campo>": <tipoDeOrdenacion> } )
```

- Por ejemplo, si el departamento de recursos humanos necesita a menudo buscar empleados por su apellido. Puede crear un índice en el `apellido` de empleado para mejorar el rendimiento de esa consulta

```
db.empleados.create_index( { "apellido": 1 } )
```

Tipos de índices en MongoDB

Índices compuestos (*compound indexes*)

- Los índices compuestos recopilan y ordenan datos de dos o más campos de cada documento de una colección
- Los datos se agrupan por el primer campo del índice y, a continuación, por cada campo subsiguiente
- Para su creación se sigue un formato parecido al de los *single field index*

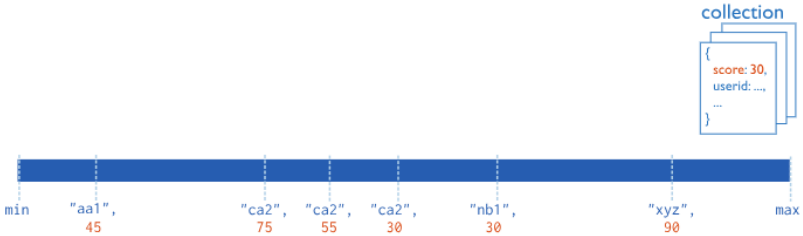
```
db.<collection>.create_index( {
   <campo1>: <tipoDeOrdenacion>,
   <campo2>: <tipoDeOrdenacion>,
   ...
   <campoN>: <tipoDeOrdenacion>
}  )
```

Tipos de índices en MongoDB

Indices compuestos (*compound indexes*) (ii)

- Ejemplo: si quisieramos crear un índice sobre los campos `userid` (de forma ascendente) y `score` (de forma descendente) de los documentos de una colección `usuarios` deberíamos ejecutar

```
db.usuarios.create_index({"userid": 1, "score": -1})
```

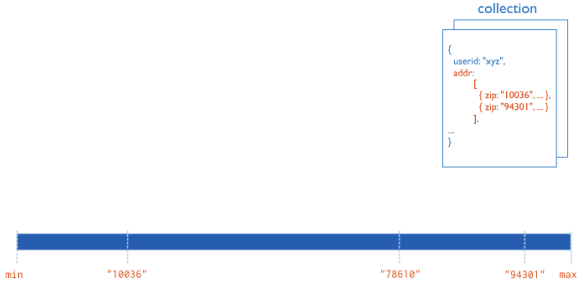


Tipos de índices en MongoDB

Indices multiclave (*multikey indexes*)

- Los índices multiclave recopilan y ordenan datos de campos que contienen un array como valor asociado
- Su creacion es igual a los índices anteriores pero MongoDB establece automáticamente que ese índice sea multiclave

```
1 db.empleados.create_index( { "addr.zip": 1 } )
```



Tipos de índices en MongoDB

Indices de texto (*text indexes*)

- Los índices de texto permiten realizar búsquedas sobre campos de tipo `string`
- Los índices de texto mejoran el rendimiento cuando se buscan palabras o frases específicas dentro del contenido de una cadena
- Una colección sólo puede tener un índice de texto, pero ese índice puede abarcar varios campos

```
db.<coleccion>.create_index( {
   <campo1>: "text",
   <campo2>: "text",
   ...
   <campoN>: "text"
} )
```

Tipos de índices en MongoDB

Indices de texto (*text indexes*) – Ejemplo

- Vamos a suponer que creamos una colección `blog` con los siguientes documentos

```
db.blog.insert_many( [
   {
     _id: 1,
     "contenido": "Esta mañana me he comado una taza de café.",
     "sobre": "bebida",
     "palabras_clave": [ "café" ]
   },
   {
     _id: 2,
     "contenido": "¿A quién le apetece un helado de chocolate de postre?",
     about: "comida",
     "palabras_clave": [ "encuesta" ]
   },
   {
```

Tipos de índices en MongoDB (cont.)

Indices de texto (*text indexes*) – Ejemplo

```
    _id: 3,
    "contenido": "Mis sabores favoritos son la fresa y el chocolate",
    "sobre": "helado",
    "palabras_clave": [ "comida", "postre" ]
  }
] )
```

- Puesto que vamos a querer realizar muchas búsquedas sobre el campo `contenido`, vamos a definir un índice sobre dicho campo

```
db.blog.create_index( { "content": "text" } )
```

Tipos de índices en MongoDB (cont.)

Indices de texto (*text indexes*) – Ejemplo

- Esto permite realizar, por ejemplo, una búsqueda para recuperar los documentos que contengan la palabra *café* en el campo `contenido`

```
db.blog.find(
   {
      "$text": { "$search": "café" }
   }
)
```

Tipos de índices en MongoDB

Indices hash (*hash indexes*)

- Recogen y almacenan los *hash* de los valores del campo indexado
- El campo que usemos para crear una clave hash de fragmento debe tener una cardinalidad alta, es decir, un gran número de valores diferentes
- Permiten la fragmentación en diferentes clusters (*sharding*) mediante claves hash

```
db.<coleccion>.create_index(
   {
      "<campo>": "hashed"
   }
)
```

La importancia de los índices en la eficiencia

- Los índices son cruciales para la eficiencia
- Y hay que conocer cuáles usar y si realmente una búsqueda los usará
- Recordemos la agregación sencilla de antes:

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

- Nótese cómo se busca al usuario por el campo “`Id`” (ojo, no “`_id`”)

La importancia de los índices en la eficiencia (cont.)

- Tiempo de ejecución sin índice: **24 segundos**
- Defino un índice para el campo “`Id`” (valdría también *hash* porque se hace una búsqueda exacta):

```
db.users.create_index({"Id": 1})
> 'Id_1'
```

- Ejecuto de nuevo: **67,2 milisegundos**

La importancia de los índices en la eficiencia (cont.)

- En pymongo hay que hacerlo de forma especial para saber que se ha usado el índice:

```
db.command('aggregate', 'posts', pipeline=pipeline, explain=True)
...
'winningPlan': {'queryPlan': {'stage': 'EQ_LOOKUP',
    'foreignCollection': 'stackoverflow.users',
    'localField': 'OwnerUserId',
    'foreignField': 'Id',
    'asField': 'owner',
    'strategy': 'IndexedLoopJoin',
    'indexName': 'Id_1',
    'indexKeyPattern': {'Id': 1},
    'inputStage': {'stage': 'COLLSCAN',
     'planNodeId': 1,
     'filter': {'Score': {'$gte': 40}},
     'direction': 'forward'}},
...
```

Transacciones

Transacciones en MongoDB

- En MongoDB, una operación en un solo documento es atómica
- El uso de documentos incrustados permite capturar relaciones entre datos en una sola estructura de documento
  - Elimina la necesidad de transacciones distribuidas para muchos casos
- Para situaciones que requieren atomicidad de lecturas y escrituras en varios documentos, MongoDB soporta transacciones distribuidas
- Pueden ser utilizadas en múltiples operaciones, colecciones, bases de datos y documentos

Transacciones en MongoDB

- En MongoDB ofrece una `TransactionAPI` que a a su vez hace uso de la `callback API`
- Esta API hace la siguiente secuencia de operaciones:
  1. Comienza una transacción
  2. Ejecuta las operaciones indicadas
  3. Hace *commit* del resultado (o aborta en caso de error)

Transacciones en MongoDB – Ejemplo

- 1. Nos conectamos a la BD

```
client = MongoClient(MONGODB_URI)
```

- 2. Definimos el *callback* que especifica la secuencia de operaciones a realizar

```
def callback(sesion):
    coleccion_1 = session.client.db.foo
    coleccion_2 = session.client.db.bar
    # Importante:: Se debe de pasar el parámetro de sesión a los operadores.
    coleccion_1.insert_one({"abc": 1}, session=sesion)
    coleccion_2.insert_one({"xyz": 999}, session=sesion)
```

Transacciones en MongoDB – Ejemplo (cont.)

- 3. Iniciamos la sesión de cliente y ejecutamos la transacción (el bloque `with` ejecuta el *commit*

```
with client.start_session() as session:
    session.with_transaction(callback)
client.close()
```

- (si algún paso falla, se puede llamar siempre a `session.abort_transaction()`, por ejemplo en un `catch` de una excepción)