El caso para Base de Datos NO relacionales

- Grandes volúmenes de datos y sistemas basados en la Web
- Escalabilidad, elasticidad, eficiencia, tolerancia a fallas
- Replicación y particionado de datos.
- No existen Joins. El costo de joins en bases de datos relacionales llega a ser demasiado elevado cuando se trata de tablas con grandes cantidades de tuplas.
- Se aplica DES-normalización
- Clusters de procesadores y centros de datos
- Conceptos de paralelismo y sistemas distribuidos embebidos en el modelo de datos y el despliegue físico del modelo de datos.
- Diseño del modelo de datos basado en las CONSULTAS que debe soportar la aplicación, las tablas se diseñan para procesarlas de manera eficiente.

NoSQL = Not Only SQL databases

Existen al menos tres tipos de bases de datos NoSQL.

- Pares **llave-valor**, donde “valor” puede ser cualquier objeto, binario, JSON, se utilizan en situaciones donde es necesario almacenar y recuperar de manera eficiente objetos individuales tales como imágenes.

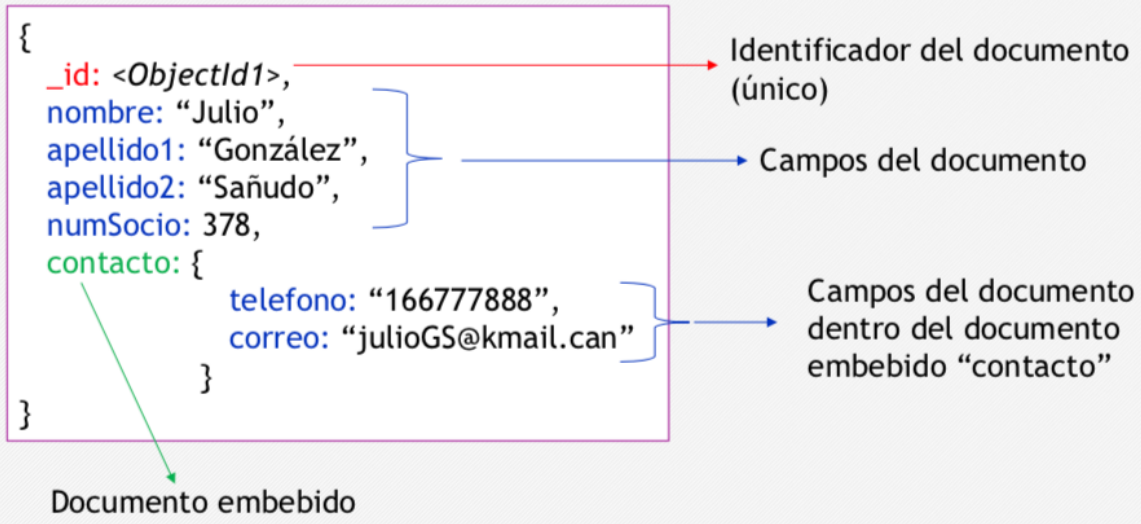
- **Documentales**, los objetos tienen mayor estructura, XML, JSON, texto, son documentos que pueden tener una estructura de árbol y/o pueden referenciar a otros documentos. Por ejemplo un documento que describe una orden de compra que contiene dirección de entrega del pedido, dirección de envío de la factura, y la lista de productos con la cantidad requerida de cada producto (en bases de datos relacionales se requieren varias tablas).

- **Columnares,** tienen la forma de tablas donde se especifican familias de columnas, muy flexibles, y se proporcionan operaciones que permitan ejecutar funciones sobre las familias de columnas. No tienen las restricciones de tablas con campos fijos del modelo relacional. Las filas pueden contener diferentes subconjuntos de columnas, y dentro de cada familia de columnas pueden haber varias filas. Explotan localidad de accesos en grandes volúmenes de datos.

- **Grafos,** permiten almecenar y recuperar eficientemente relaciones entre pares de datos ”Objeto → Relación → Objeto”.

- MongoDB es una Base de Datos NoSQL orientada a documentos en formato BSON.
  - Similar a JSON, con algunas particularidades:
    - Tipos de datos adicionales
    - Diseñado para ser más eficiente en espacio.
      - Aunque en ocasiones un JSON puede ocupar menos que un BSON.
    - Diseñado para que las búsquedas sean más eficientes.

- Ejemplo de documento:

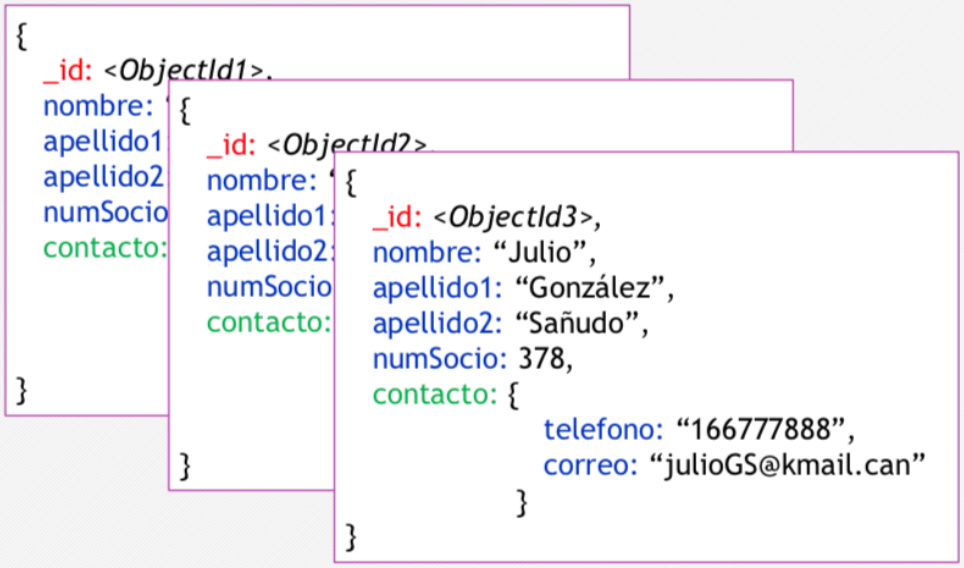


https://ocw.unican.es/pluginfile.php/2396/course/section/2473/NoSQL_Tema2_MongoDB.pdf

- Los datos son accedidos mediante operaciones *CRUD* (*Create*, *Read*, *Update*, *Delete*):
  - Find: lectura (Read) de datos. Símil con el SELECT en SQL.
  - Insert: escritura (Create) de datos.
  - Update: actualización de valores en los datos.
  - Remove: borrado (Delete) de datos.
- La ejecución de cualquiera de estas instrucciones sigue el siguiente esquema:

```
db.<nombre_colección>.<nombre_operación>(<condiciones de la operación>)
```

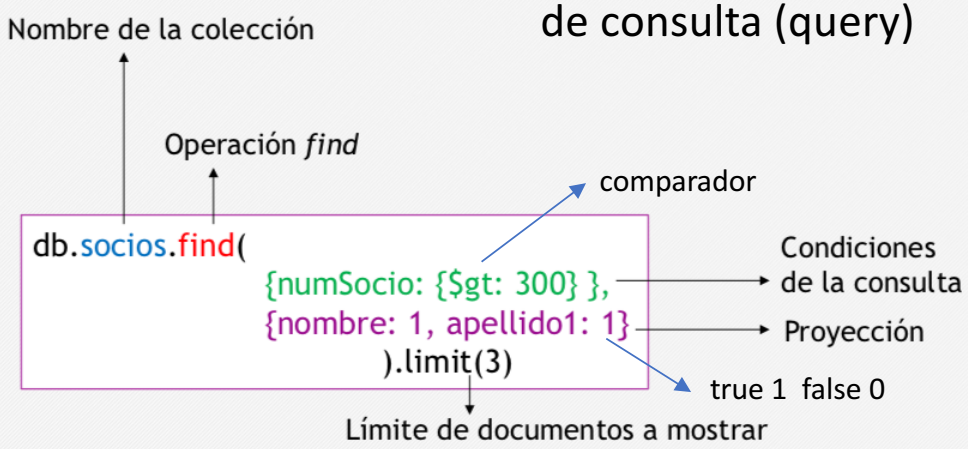
- Ejemplo de colección:



https://ocw.unican.es/pluginfile.php/2396/course/section/2473/NoSQL_Tema2_MongoDB.pdf

- Ejemplo de uso de *find*:

Operación principal de consulta (query)



- Para acceder a campos de documentos embebidos, se utiliza la notación con punto. En este caso, el uso de las comillas es necesario:

```
db.socios.find(
                {numSocio: {$gt: 300} },
                {nombre: 1, apellido1: 1, "contacto.teléfono": 1}
                ).limit(3)
```

Más sobre la operación find en https://docs.mongodb.com/manual/tutorial/query-documents/

https://ocw.unican.es/pluginfile.php/2396/course/section/2473/NoSQL_Tema2_MongoDB.pdf

SELECT item FROM inventory WHERE status=”A”	db.inventory.find({status:”A”}, {item:1,_id:0})
SELECT item FROM inventory WHERE status <>”A”	db.inventory.find({status:{$ne:”A”}}, {item:1,_id:0})
SELECT * FROM inventory WHERE status=“A” or item = “paper”	db.inventory.find({$or:[{status:"A"}, {item:"paper"}]})
SELECT * FROM inventory WHERE status=“A” ORDER BY item desc	db.inventory.find({status:"A"}).sort({item:-1})
SELECT count(*) FROM inventory WHERE status=“A”	db.inventory.count({status:"A"}) db.inventory.find({status:"A"}).count()

- Existen diferentes tipos de operaciones:

Forma implícita de la operación $eq

De igualdad (*$eq*):

```
db.socios.find({numSocio: {$eq: 300} })
```

=

```
db.socios.find({numSocio: 300 })
```

Mayor que ($gt):

```
db.socios.find({numSocio: {$gt: 300} })
```

Mayor o igual que ($gte):

```
db.socios.find({numSocio: {$gte: 300} })
```

Menor que ($lt):

```
db.socios.find({numSocio: {$lt: 300} })
```

Menor o igual que ($lte):

```
db.socios.find({numSocio: {$lte: 300} })
```

Diferente a ($ne):

```
db.socios.find({numSocio: {$ne: 300} })
```

Pertenece ($in) o no ($nin) a un conjunto

```
db.socios.find({numSocio: {$in: [300, 301]} })
db.socios.find({numSocio: {$nin: [300, 301]} })
```

- Operadores lógicos:

Ambas condiciones han de cumplirse($and):

```
db.socios.find({$and: [{nombre: "Juan"}, {apellido1: "Galindo" }  ]})
```

Alguna de las condiciones ha de cumplirse ($or):

```
db.socios.find({$or: [{nombre: "Juan"}, {apellido1: "Galindo" }  ]})
```

No debe de cumplir la condición ($not):

```
db.socios.find({numSocio: { $not: {$eq: 300} }})
```

Que no se cumpla ninguna condición ($nor):

```
db.socios.find({$nor: [{nombre: "Juan"}, {apellido1: "Galindo" }  ]})
```

- *Sort*: ordena los resultados según los campos dados. Un valor de 1 indica ordenamiento ascendente, y un valor de -1 ordenamiento descendente:

```
db.socios.find({numSocio: {$eq: 300} }, { nombre: 1, apellido1: 1 }).sort({nombre: 1})
```

- *Count*: cuenta el número de veces que aparece en los documentos de una colección un valor en un campo.

- Ejemplo: suponiendo que en una colección llamada "series" tenemos un campo puntuación, y los siguientes documentos:

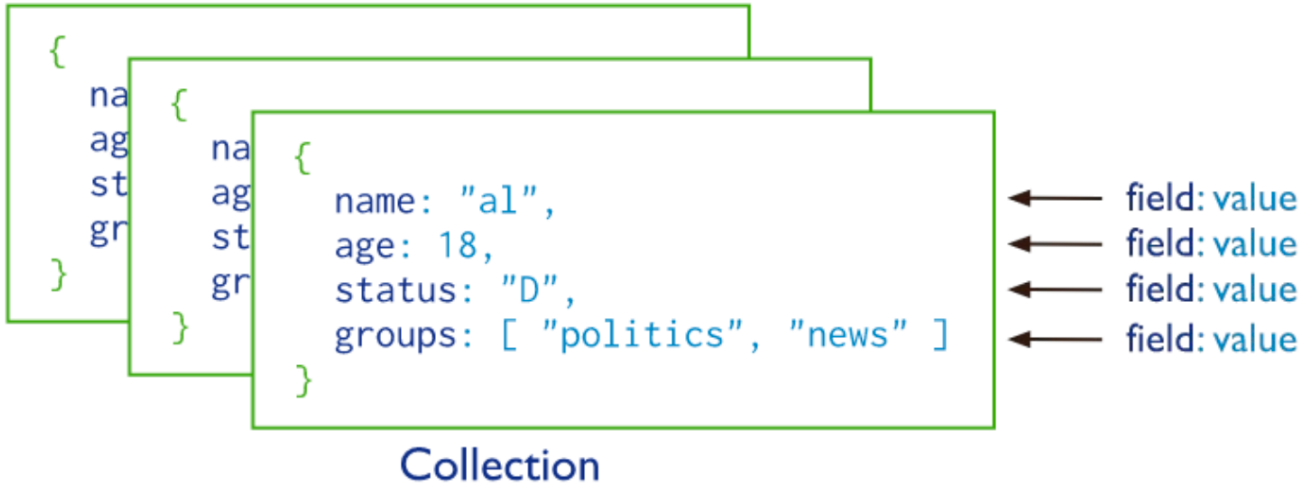
```
{puntuacion: 5}, {puntuacion: 3}, {puntuacion: 5}, {puntuacion: 5}, {puntuacion: 2},
```

La siguiente instrucción retornaría **3** (3 veces que se repite el valor 5 en el campo puntuación:

```
db.series.count({puntuacion: 5})
```

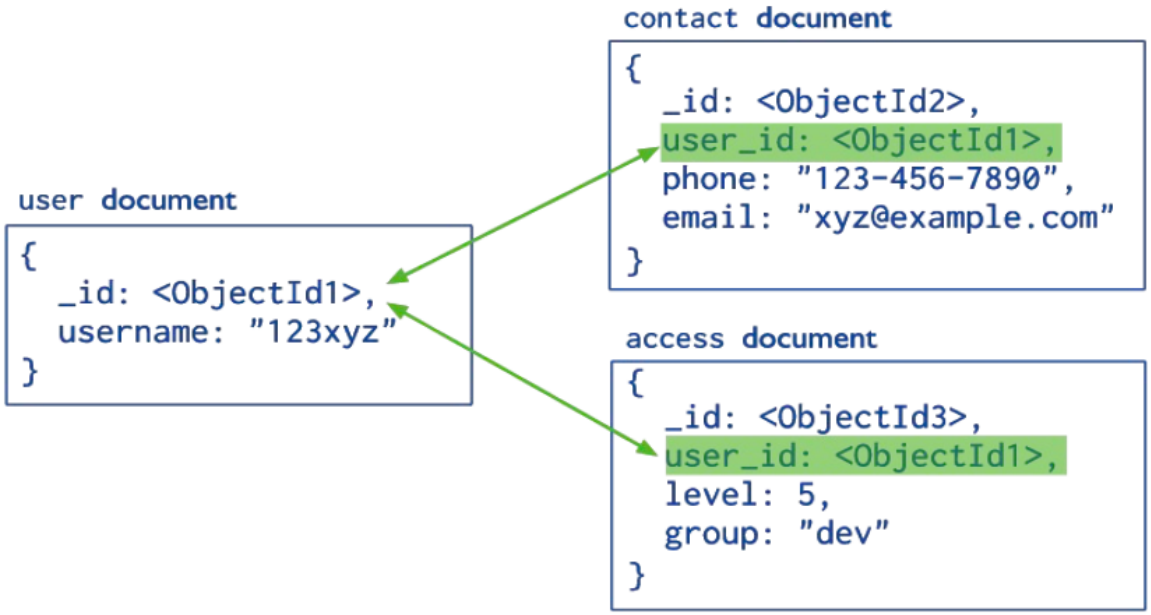
https://ocw.unican.es/pluginfile.php/2396/course/section/2473/NoSQL_Tema2_MongoDB.pdf

DOCUMENT AND COLLECTION




```
var mydoc = {
                _id: ObjectId("5099803df3f4948bd2f98391"),
                name: { first: "Alan", last: "Turing" },
                birth: new Date('Jun 23, 1912'),
                death: new Date('Jun 07, 1954'),
                contribs: [ "Turing machine", "Turing test", "Turingery" ],
                views : NumberLong(1250000)
              }
```

DOCUMENT



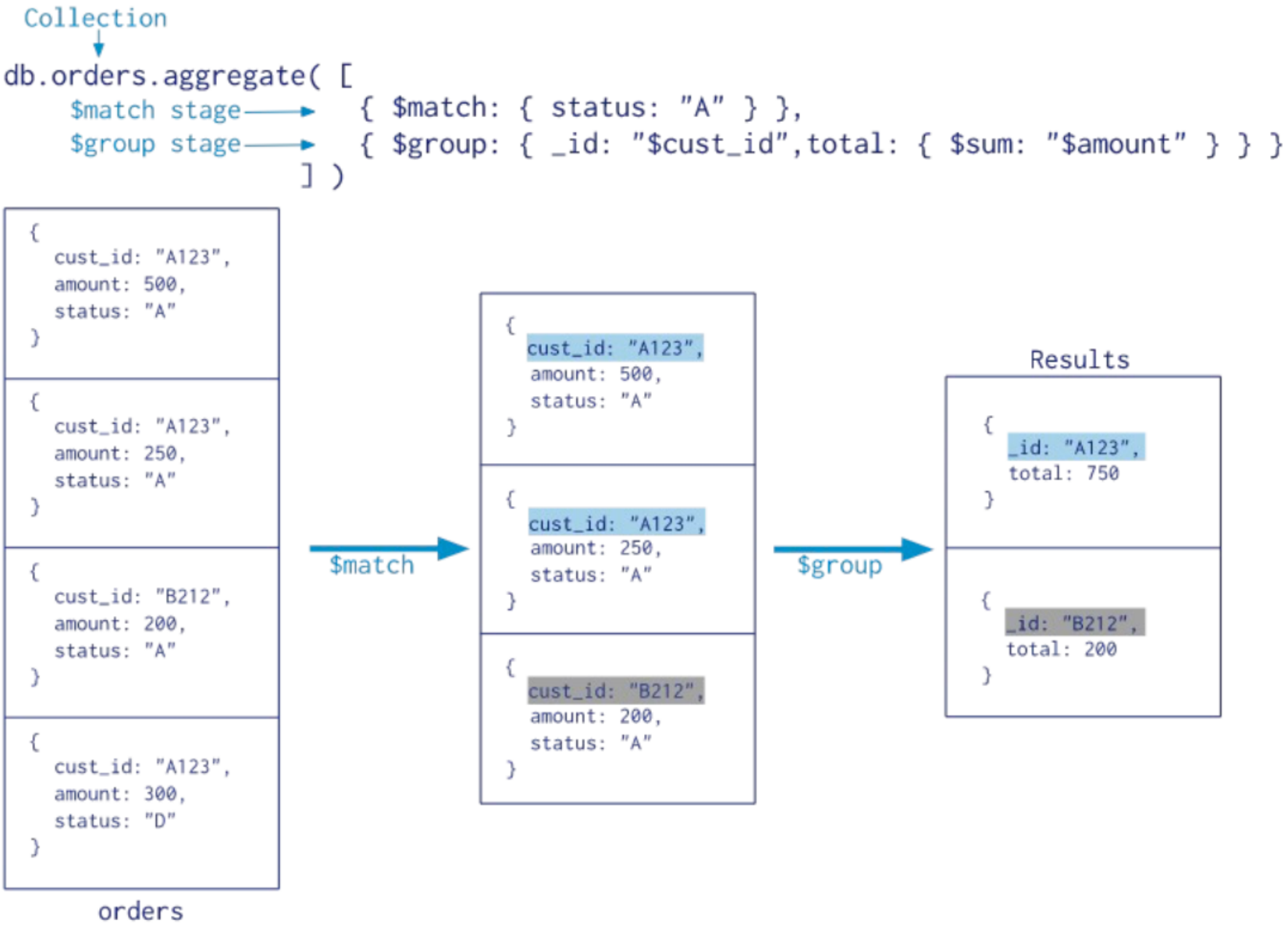
Embedded Data

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
              level: 5,
              group: "dev"
            }
}
```

Embedded sub-document

Embedded sub-document

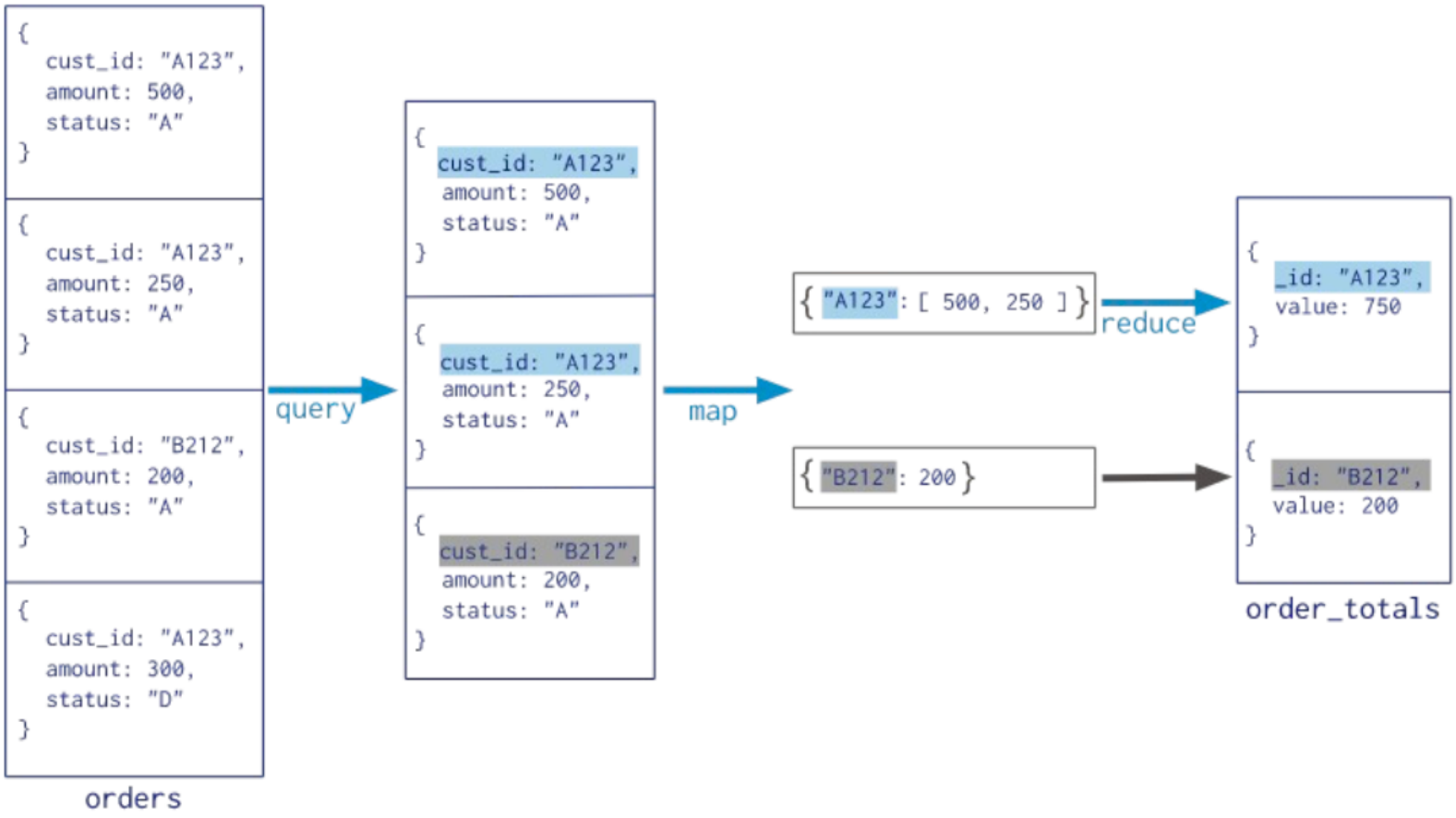
AGGREGATION PIPELINE



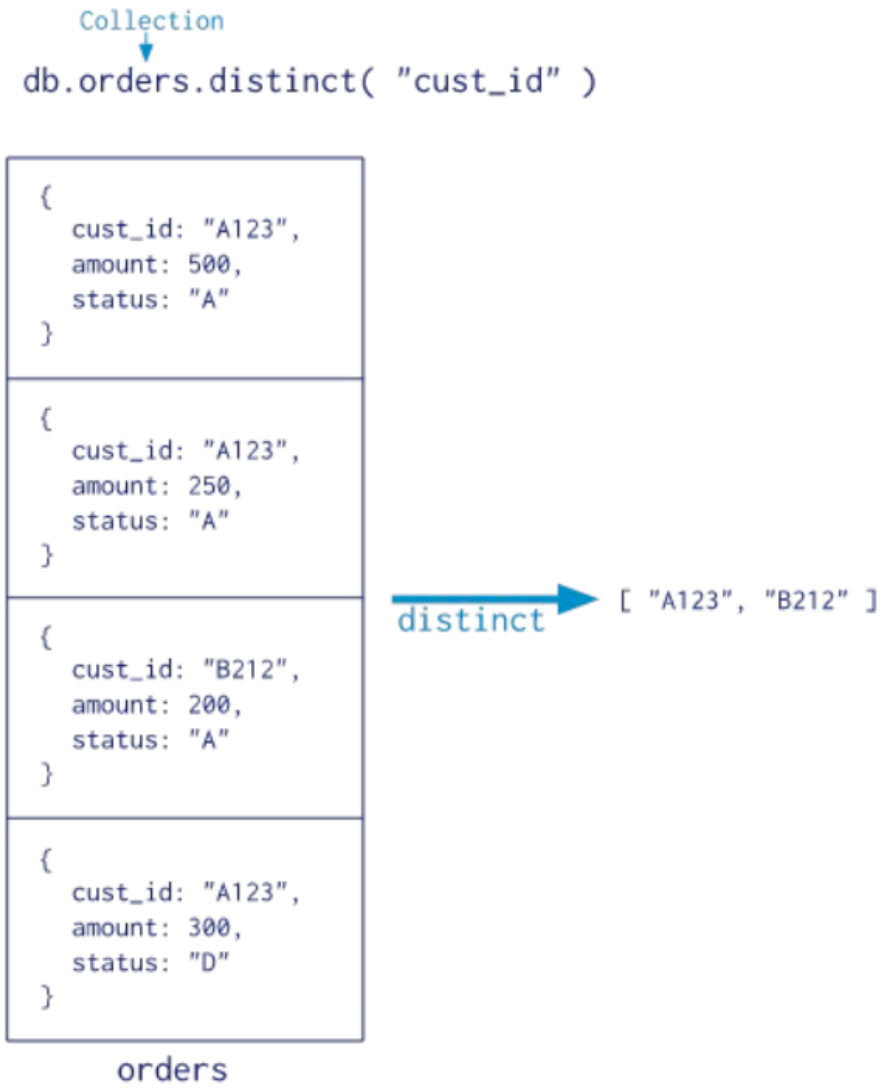
AGGREGATION MAP-REDUCE

Collection

```
db.orders.mapReduce(
          map      →   function() { emit( this.cust_id, this.amount ); },
          reduce   →   function(key, values) { return Array.sum( values ) },
                       {
          query    →     query: { status: "A" },
          output   →     out: "order_totals"
                       }
                     )
```



AGGREGATION SINGLE PURPOSE



SEARCH TEXT

```
db.stores.insert(
   [
     { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
     { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
     { _id: 3, name: "Coffee Shop", description: "Just coffee" },
     { _id: 4, name: "Clothes Clothes Clothes", description: "Discount clothing" },
     { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
   ]
)
```

```
db.stores.createIndex( { name: "text", description: "text" } )
```

```
db.stores.find( { $text: { $search: "java coffee shop" } } )
```

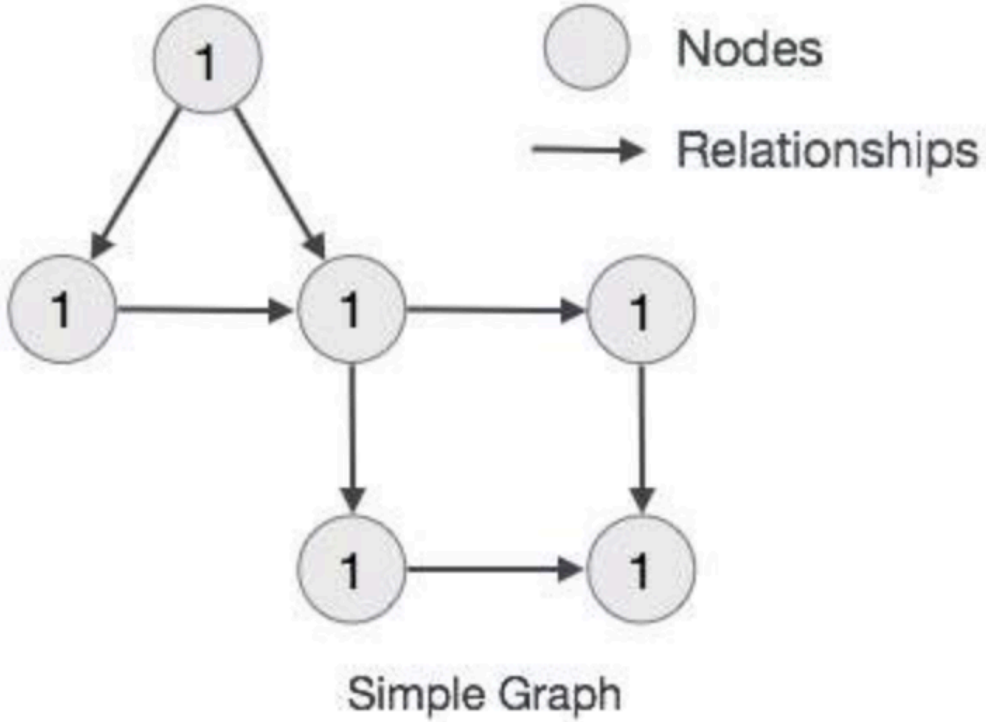
```
db.stores.find( { $text: { $search: "java \"coffee shop\"" } } )
```

```
db.stores.find( { $text: { $search: "java shop -coffee" } } )
```

Gestor de Bases de Datos en Grafos **Neo4J**

http://www.tutorialspoint.com/neo4j

Simple Property Graph example



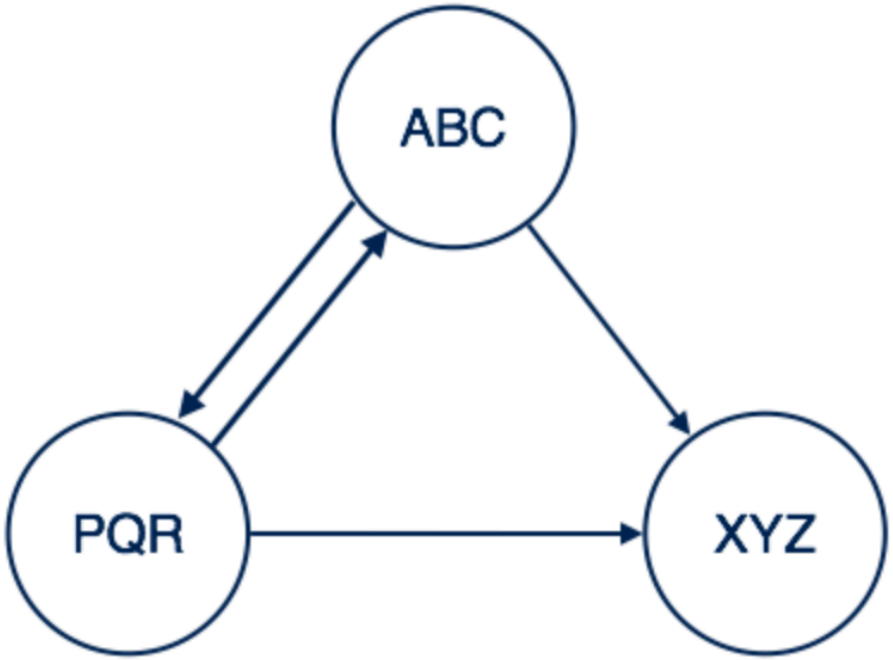
Profile Node

This node contains a set of properties. A Property is a name:value pair.

Relationships are created between two nodes

Here Relationship name "Follows" is created between two profiles. That means Profile-I follows Profile-II.

Complex Graph Example



Here nodes are connected with relationships. Relationships are unidirectional or bi-directional.

- Relationship from XYZ to PQR is a unidirectional relationship.
- Relationship from ABC to XYZ is a bidirectional relationship.

The following are frequently used Neo4j CQL Commands/Clauses:

S.No.	CQL Command/Clause	Usage
1.	CREATE	To create nodes, relationships and properties
2.	MATCH	To retrieve data about nodes, relationships and properties
3.	RETURN	To return query results
4.	WHERE	To provide conditions to filter retrieval data
5.	DELETE	To delete nodes and relationships
6.	REMOVE	To delete properties of nodes and relationships
7.	ORDER BY	To sort retrieval data
8.	SET	To add or update labels

Neo4j CQL Functions

The following are frequently used Neo4j CQL Functions:

S.No.	CQL Functions	Usage
1.	String	They are used to work with String literals.
2.	Aggregation	They are used to perform some aggregation operations on CQL Query results.
3.	Relationship	They are used to get details of Relationships like startnode,endnode etc.

CREATE command

CREATE (<node-name>:<label-name>)

Ejemplo: CREATE (emp:Employee)

```
CREATE (
   <node-name>:<label-name>
   {
      <Property1-name>:<Property1-Value>
      ........
      <Propertyn-name>:<Propertyn-Value>
   }
)
```

```
CREATE (emp:Employee{ id:123,name:"Lokesh",sal:35000,deptno:10 })
```

```
CREATE (dept:Dept{ deptno:10,dname:"Accounting",location:"Hyderabad" })
```

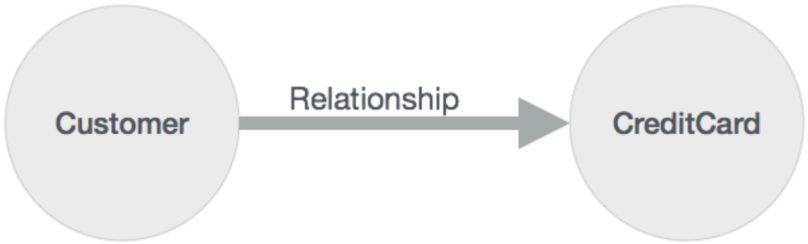
Neo4j CQL MATCH command is used -

- To get data about nodes and properties from database
- To get data about nodes, relationships and properties from database

```
MATCH
(
   <node-name>:<label-name>
)
RETURN
   <node-name>.<property1-name>,
   ........
   <node-name>.<propertyn-name>
```

Ejemplo

```
MATCH (dept:Dept)
RETURN dept.deptno, dept.dname

MATCH (dept:Dept)
RETURN dept
```



Relaciones entre nodos (relationships)

```
CREATE (e:Customer{id:"1001",name:"Abc",dob:"01/10/1982"})

CREATE
(cc:CreditCard{id:"5001",number:"1234567890",cvv:"888",expiredate:"20/17"})
----

MATCH (e:Customer),(cc:CreditCard)
CREATE (e)-[r:DO_SHOPPING_WITH ]->(cc)

MATCH (cust:Customer),(cc:CreditCard)
CREATE (cust)-[r:DO_SHOPPING_WITH{shopdate:"12/12/2014",price:55000}]->(cc)
RETURN r

MATCH (cust:Customer),(cc:CreditCard)
WHERE cust.id = "1001" AND cc.id= "5001"
CREATE (cust)-[r:DO_SHOPPING_WITH{shopdate:"12/12/2014",price:55000}]->(cc)
RETURN r
```

```
CREATE
        (video1:YoutubeVideo1 {
                title:"Action Movie1",
                updated_by:"Abc",
                uploaded_date:"10/10/2010”
                }
        )-[movie:ACTION_MOVIES{rating:1}]->
        (video2:YoutubeVideo2 {
                title:"Action Movie2",
                updated_by:"Xyz",
                uploaded_date:"12/12/2012”
                }
        )

---------

MATCH (emp:Employee)
WHERE emp.name = 'Abc'
RETURN emp

MATCH (emp:Employee)
WHERE emp.name = 'Abc' OR emp.name = 'Xyz'
RETURN emp
```

Neo4j CQL REMOVE command is used

- To remove labels of a Node or a Relationship
- To remove properties of a Node or a Relationship

Main Difference between Neo4j CQL DELETE and REMOVE commands -

- DELETE operation is used to delete Nodes and associated Relationships.
- REMOVE operation is used to remove labels and properties.

```
CREATE (book:Book {id:122,title:"Neo4j Tutorial",pages:340,price:250})

MATCH (book { id:122 })
REMOVE book.price
RETURN book

Equivalente SQL:

ALTER TABLE BOOK REMOVE COLUMN PRICE;
SELECT * FROM BOOK WHERE ID = 122;
```

Agrega un nuevo campo (par propiedad-valor), si ya existe lo actualiza al nuevo valor.

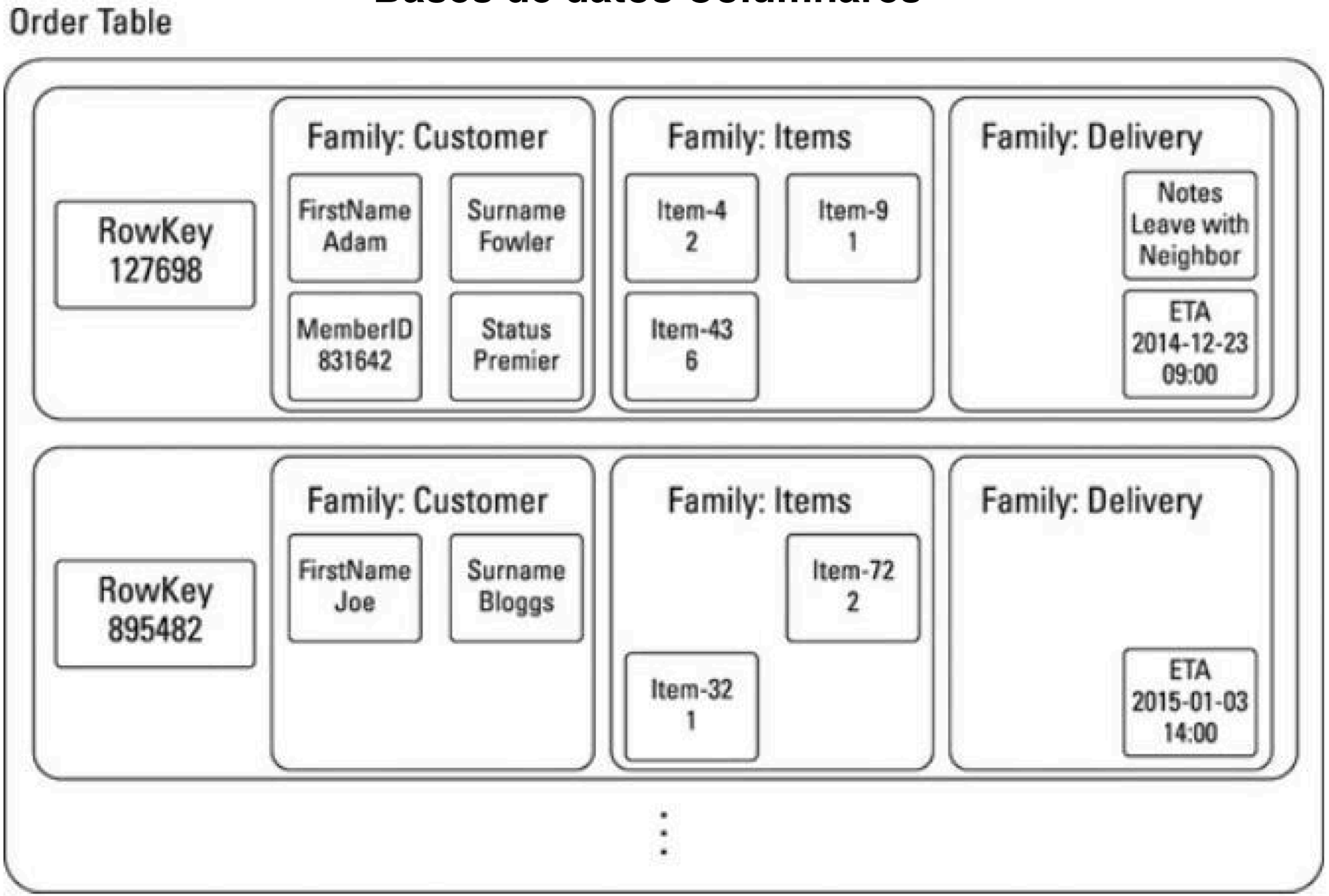
```
MATCH (dc:DebitCard)
SET dc.atm_pin = 3456
RETURN dc
-------

MATCH (emp:Employee)
RETURN emp.empid,emp.name,emp.salary,emp.deptno
ORDER BY emp.name
--------

MATCH (emp:Employee)
RETURN emp.empid,emp.name,emp.salary,emp.deptno
ORDER BY emp.name DESC
-------

MATCH (cc:CreditCard) RETURN cc.id,cc.number
UNION
MATCH (dc:DebitCard) RETURN dc.id,dc.number
```

```
MATCH (cc:CreditCard)
RETURN cc.id as id,cc.number as number,cc.name as name,
   cc.valid_from as valid_from,cc.valid_to as valid_to
UNION
MATCH (dc:DebitCard)
RETURN dc.id as id,dc.number as number,dc.name as name,
   dc.valid_from as valid_from,dc.valid_to as valid_to
```

```
MATCH (e:Employee)
WHERE e.id IS NOT NULL
RETURN e.id,e.name,e.sal,e.deptno
```

```
MATCH (e:Employee)
WHERE e.id IN [123,124]
RETURN e.id,e.name,e.sal,e.deptno
```

Bases de datos Columnares



Query Driven Methodology: process

- Entities and relationships: map to tables
- Key attributes: map to primary key columns
- Equality search attributes: must be at the beginning of the primary key
- Inequality search attributes: become clustering columns
- Ordering attributes: become clustering columns

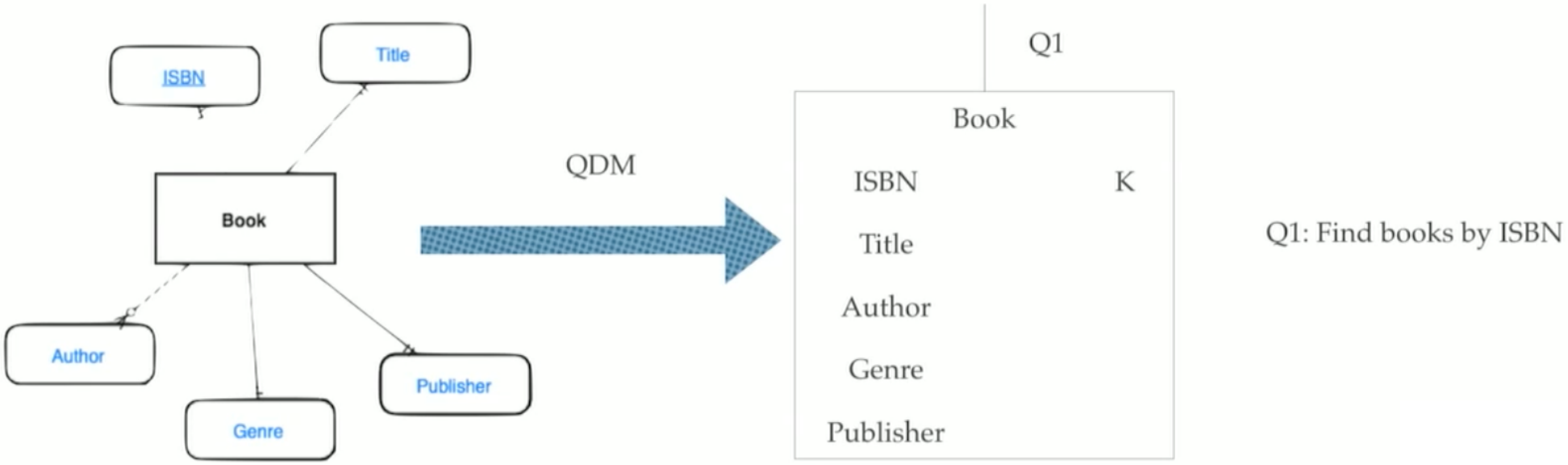
PARTITION KEY + CLUSTERING COLUMN(S)

(dos elementos para obtener rendimiento eficiente)

El diseño del modelo de datos está condicionado por las consultas que realiza la aplicación en la base de datos (Column Families).

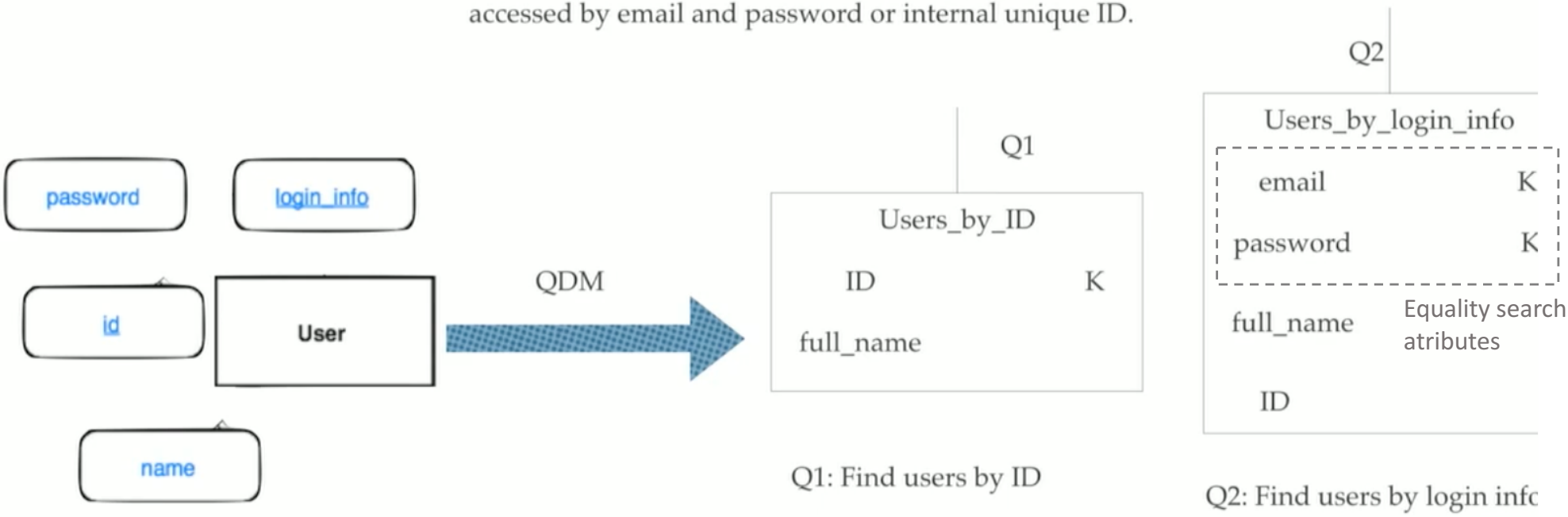
Requirement: 1

Books can be uniquely identified and accessed by ISBN, we also need a title, genre, author and publisher.



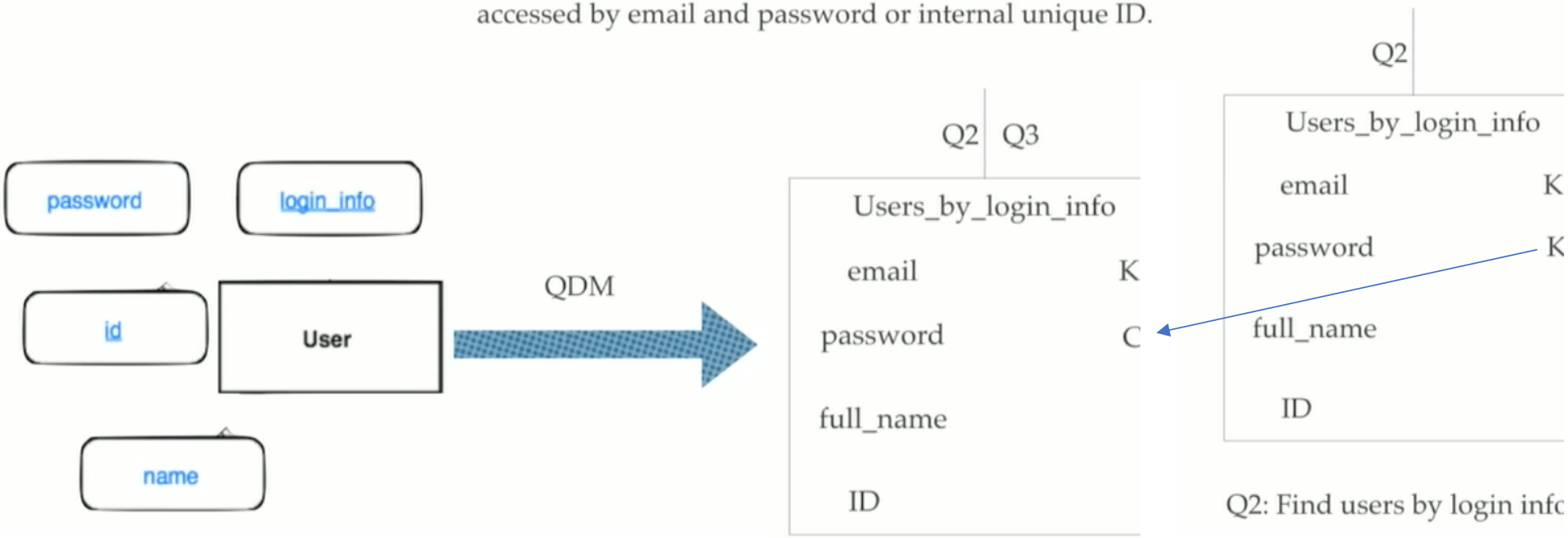
Requirement 2

Users register into the system uniquely identified by an email and a password. We also want their full name. They will be accessed by email and password or internal unique ID.



Requirement 2

Users register into the system uniquely identified by an email and a password. We also want their full name. They will be accessed by email and password or internal unique ID.



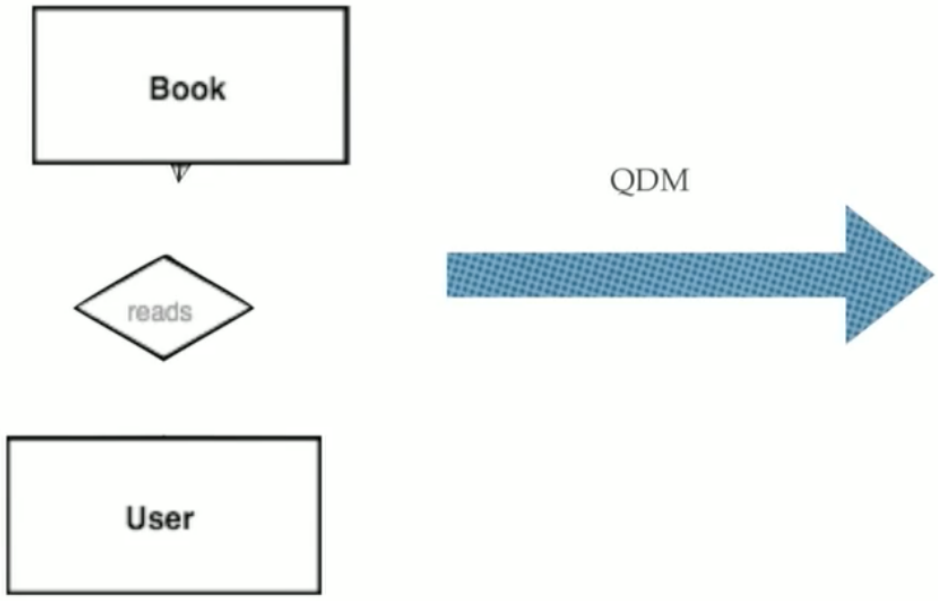
La misma tabla se puede utilizar para acomodar consultas de nuevos usuarios (Q3).

```
BEGIN BATCH
  INSERT INTO users_by_id (ID, full_name) VALUES (…) IF NOT EXISTS;
  INSERT INTO users_by_login_info (email, password, full_name, ID) VALUES (…);
APPLY BATCH;
```

Requirement 3

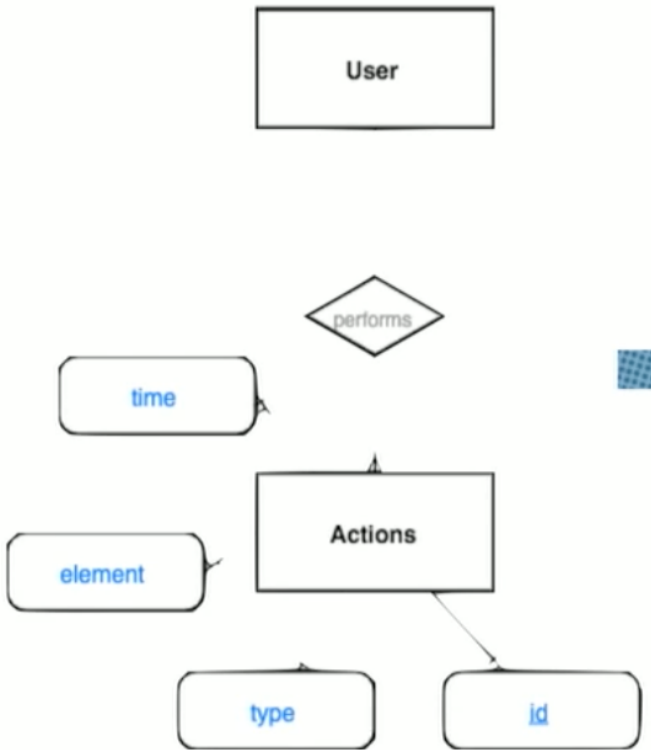
Users read books.
We want to know which books has a user read and show them sorted by title and author

Q1: Find all books a logged user has read



Requirement 4

In order to improve our site's usability we need to understand how our users use it by tracking every interaction they have with our site.



QDM

Q1

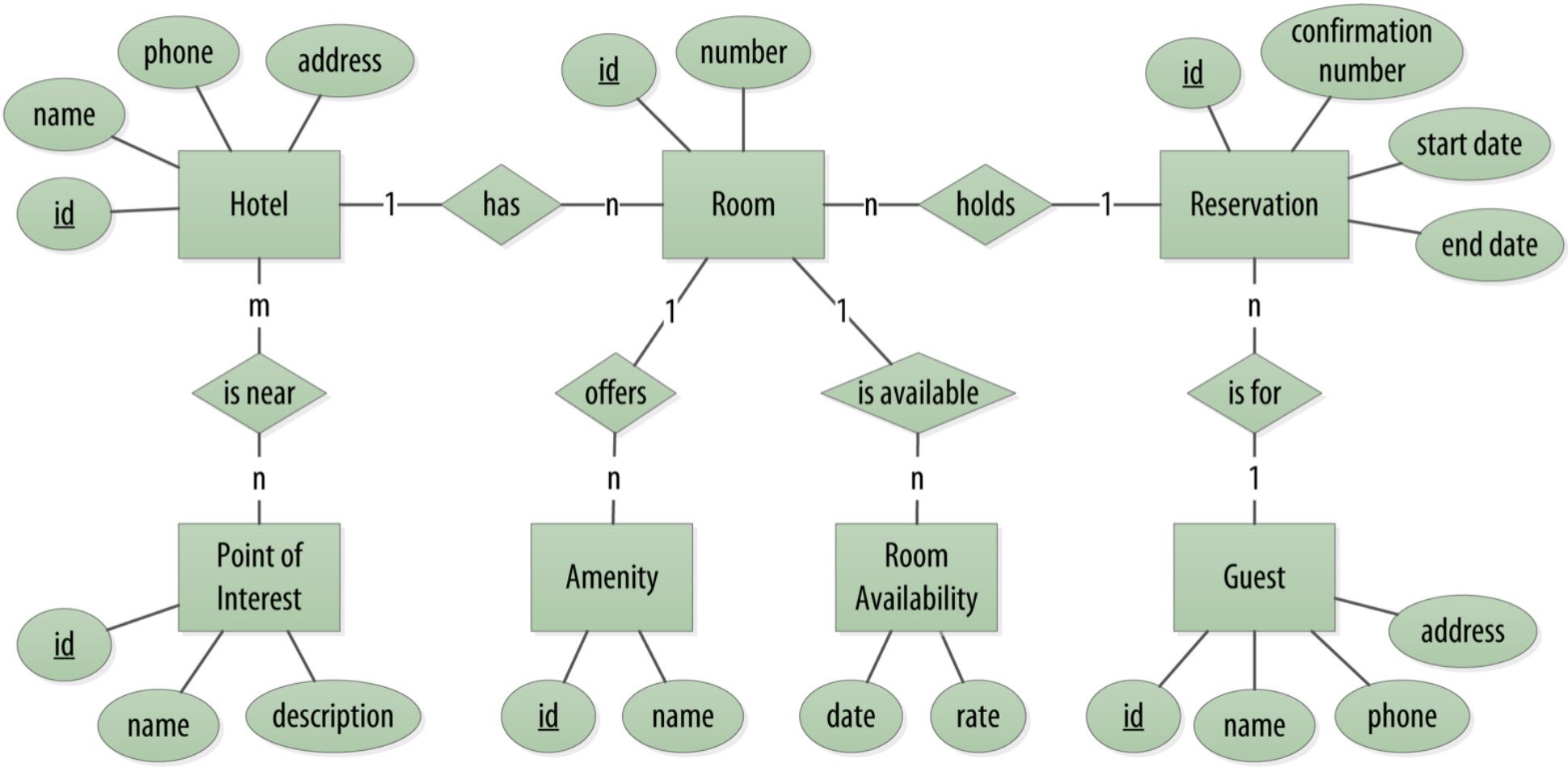
Actions_by_user

user_ID	K
time	C
element	
type	

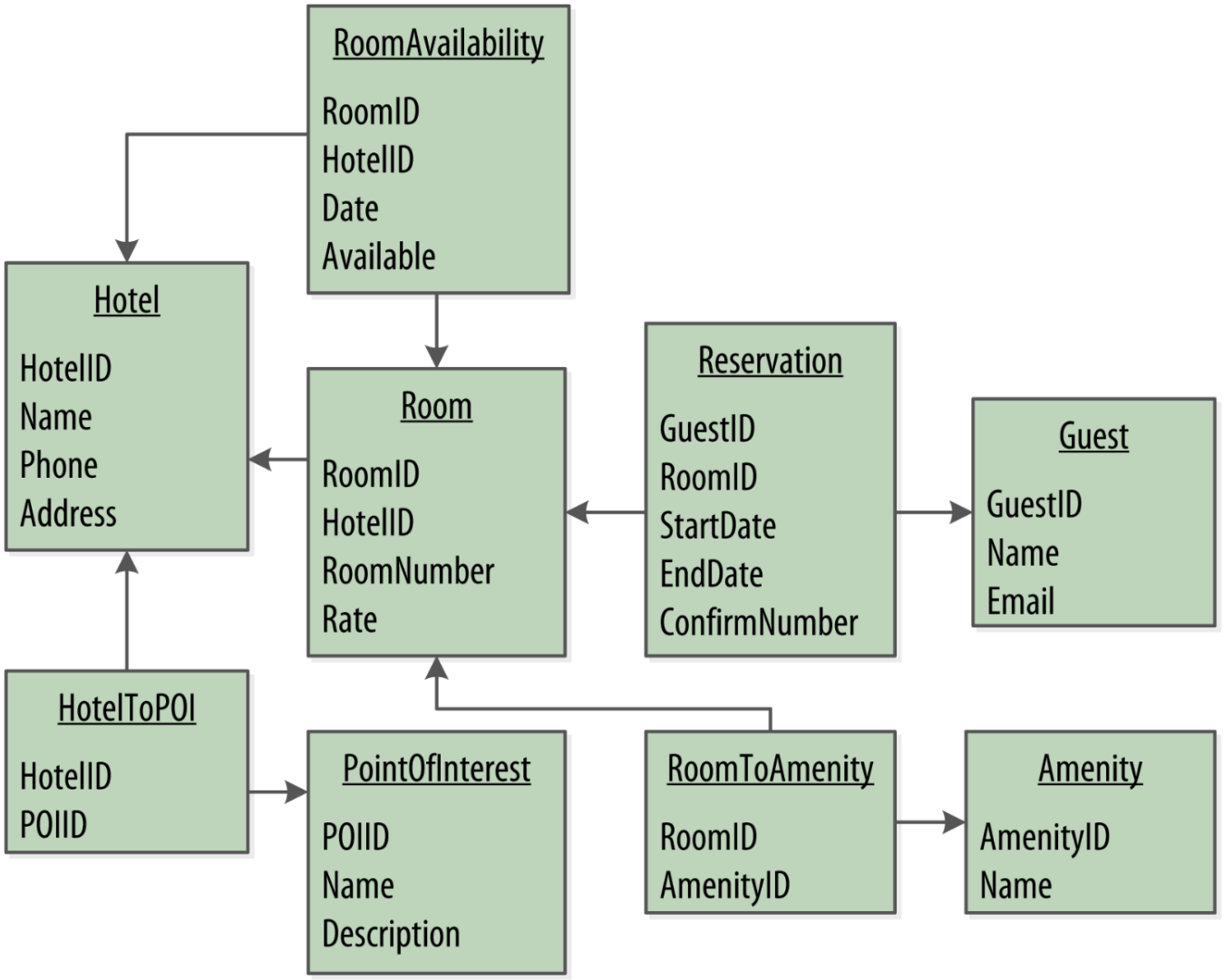
Actions_by_user

user_ID	K
month	K
time	C
element	
type	

Q1: Find all actions a user does in a time range

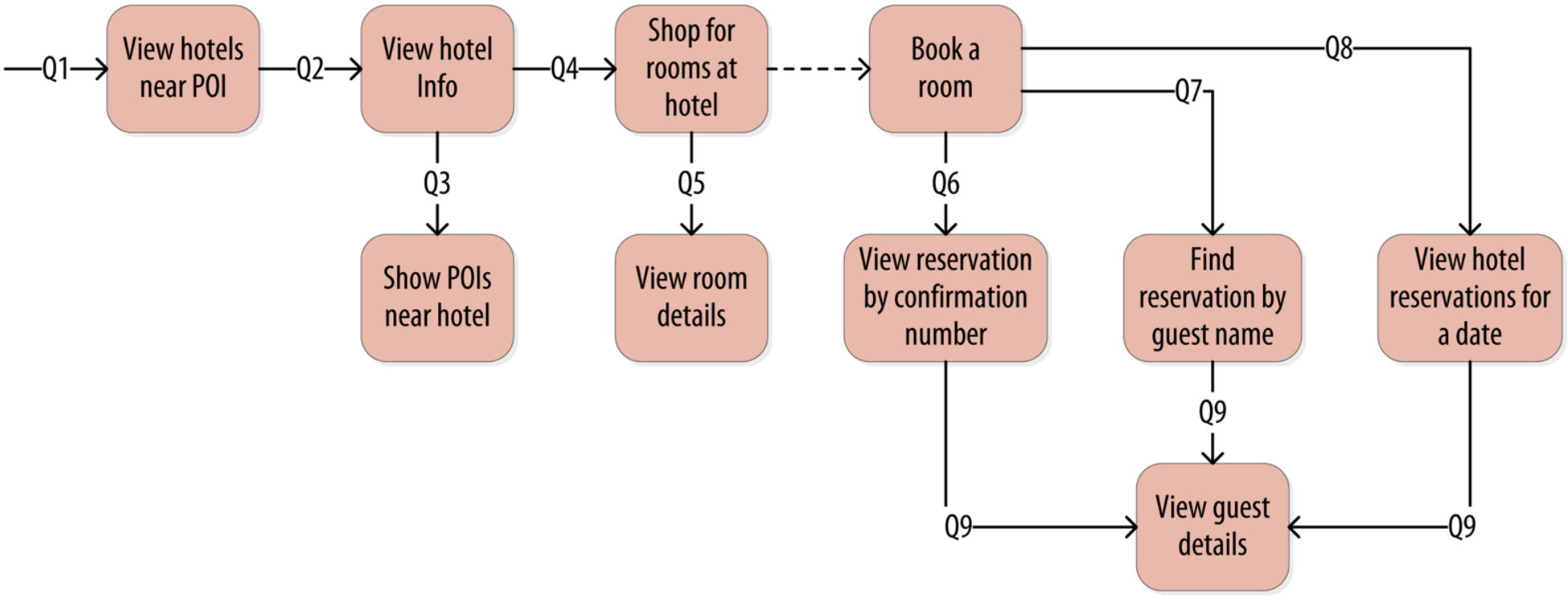


https://www.oreilly.com/library/view/data-modeling-with/9781491979723/ch01.html#idm140227301581216

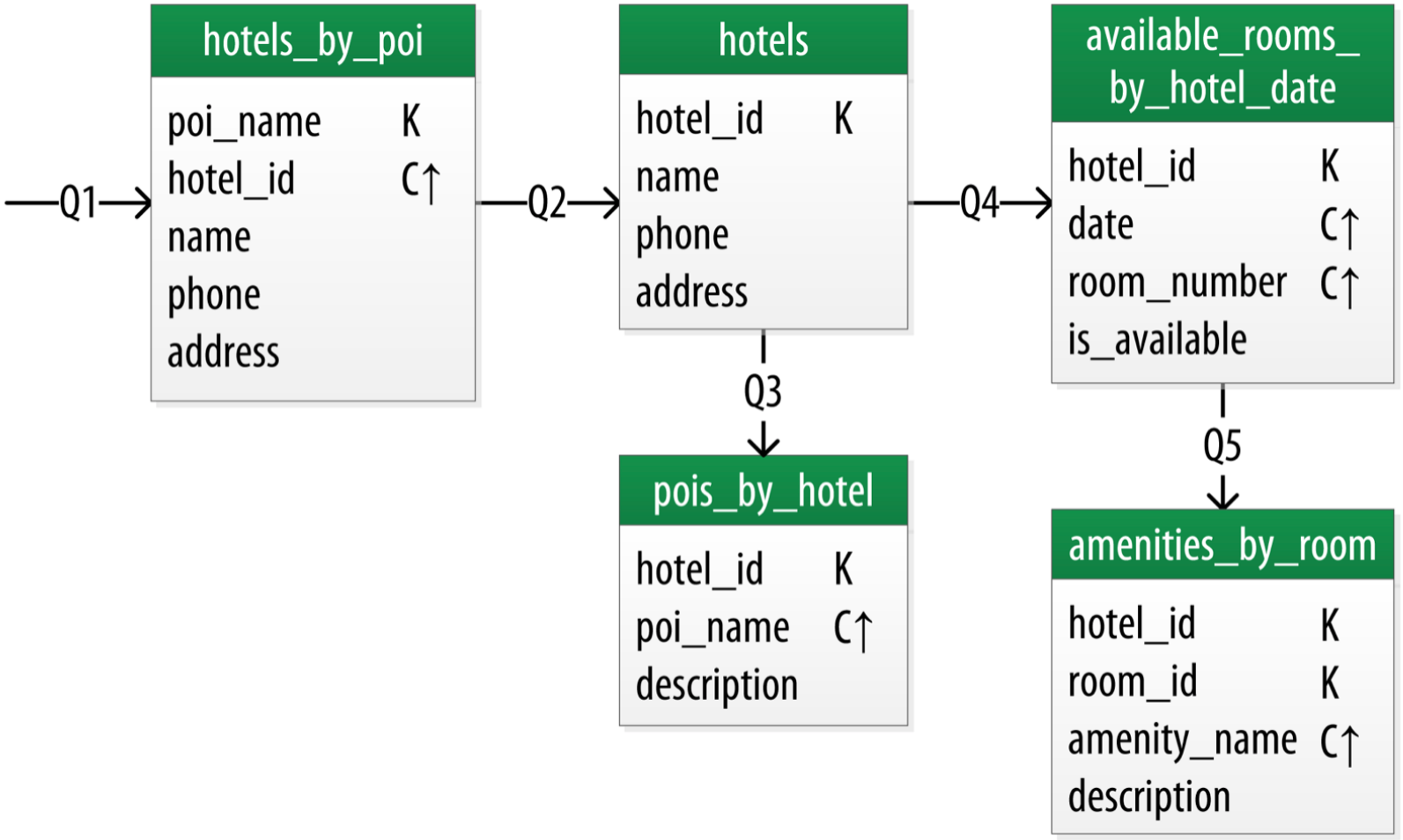


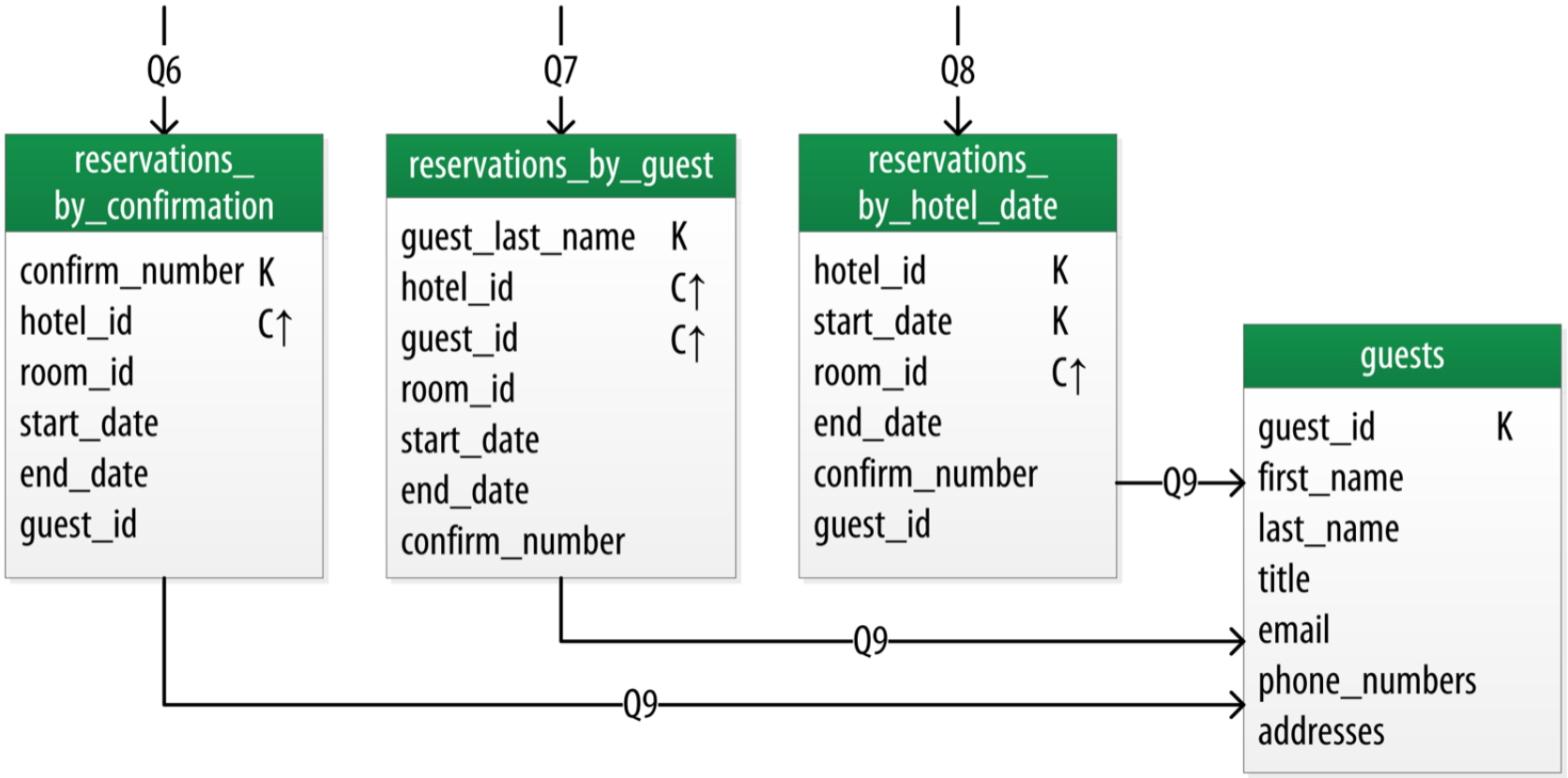
Modelo relacional de datos para el sistema de reserva de hoteles

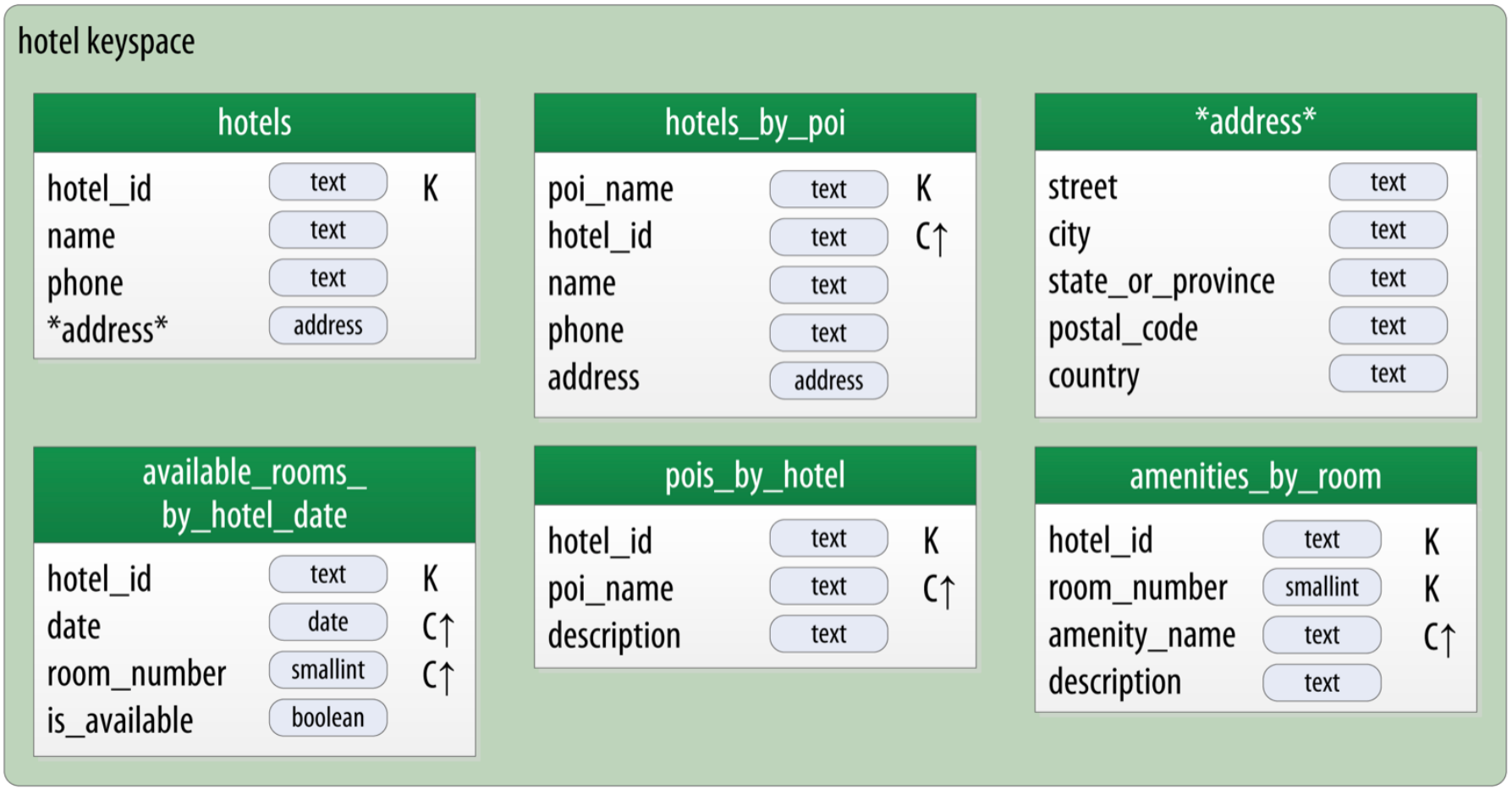
- Q1. Find hotels near a given point of interest.
- Q2. Find information about a given hotel, such as its name and location.
- Q3. Find points of interest near a given hotel.
- Q4. Find an available room in a given date range.
- Q5. Find the rate and amenities for a room.
- Q6. Lookup a reservation by confirmation number.
- Q7. Lookup a reservation by hotel, date, and guest name.
- Q8. Lookup all reservations by guest name.
- Q9. View guest details.



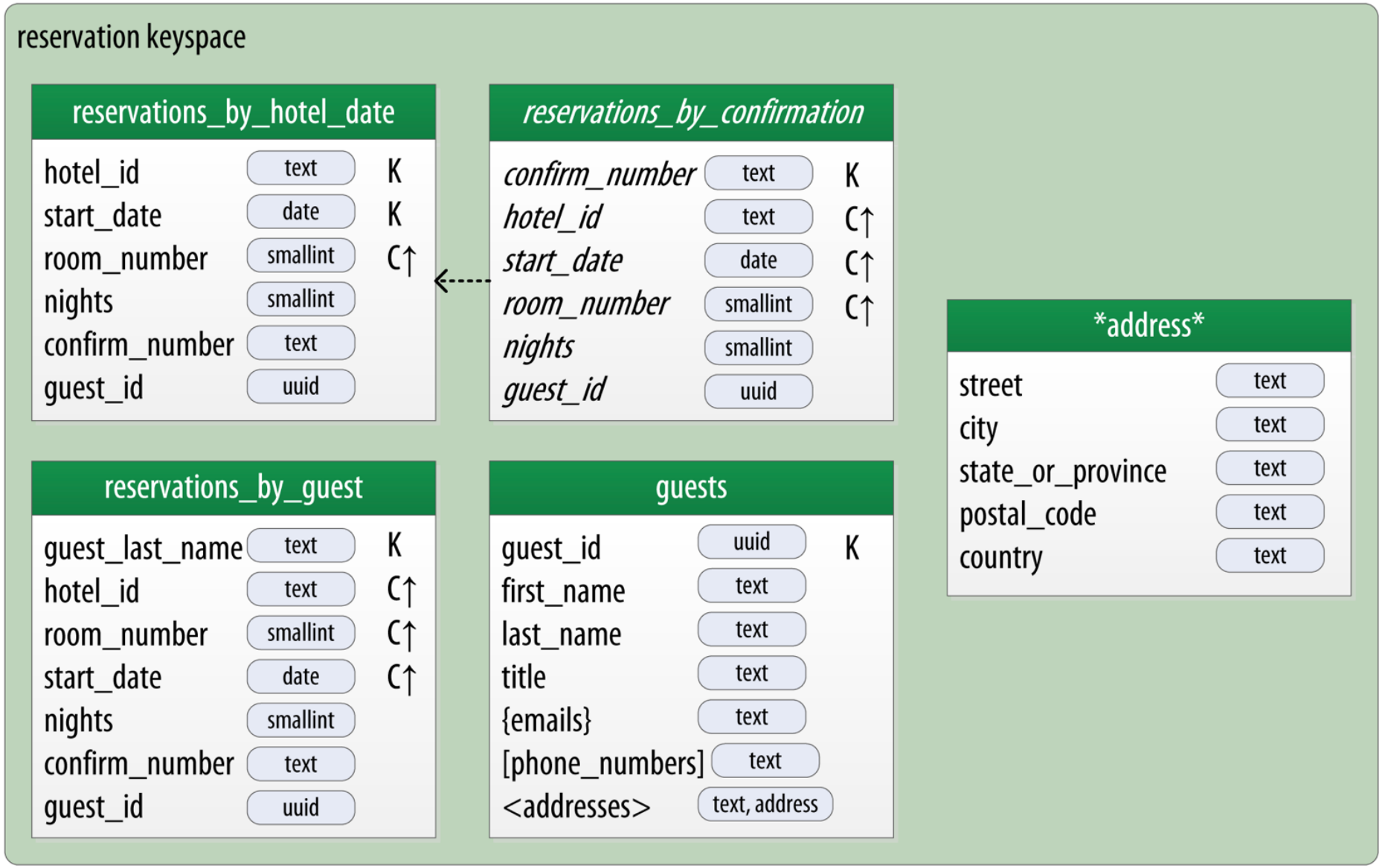
Organización lógica de las consultas que debe proporcionar el sistema de reserva de hoteles.


hotels_by_poi
poi_name K
hotel_id C↑
name
phone
address
Q1
Q2
hotels
hotel_id K
name
phone
address
Q3
pois_by_hotel
hotel_id K
poi_name C↑
description
Q4
available_rooms_by_hotel_date
hotel_id K
date C↑
room_number C↑
is_available
Q5
amenities_by_room
hotel_id K
room_id K
amenity_name C↑
description

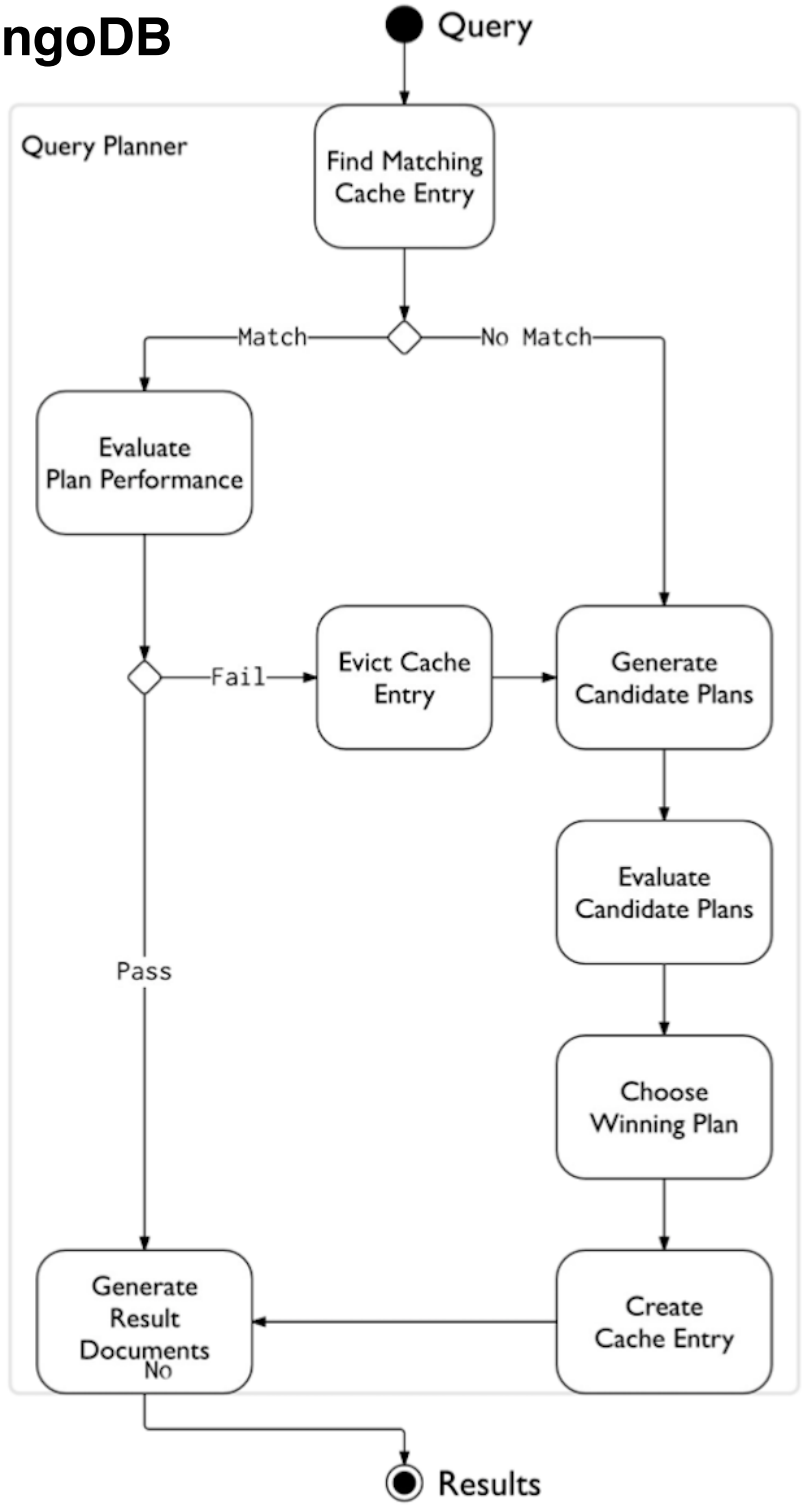

Q6
Q7
Q8
reservations_
by_confirmation
confirm_number K
hotel_id C↑
room_id
start_date
end_date
guest_id
reservations_by_guest
guest_last_name K
hotel_id C↑
guest_id C↑
room_id
start_date
end_date
confirm_number
reservations_
by_hotel_date
hotel_id K
start_date K
room_id C↑
end_date
confirm_number
guest_id
guests
guest_id K
first_name
last_name
title
email
phone_numbers
addresses
Q9
Q9
Q9


hotel keyspace
hotels
hotel_id text K
name text
phone text
address address
hotels_by_poi
poi_name text K
hotel_id text C↑
name text
phone text
address address
address
street text
city text
state_or_province text
postal_code text
country text
available_rooms_by_hotel_date
hotel_id text K
date date C↑
room_number smallint C↑
is_available boolean
pois_by_hotel
hotel_id text K
poi_name text C↑
description text
amenities_by_room
hotel_id text K
room_number smallint K
amenity_name text C↑
description text

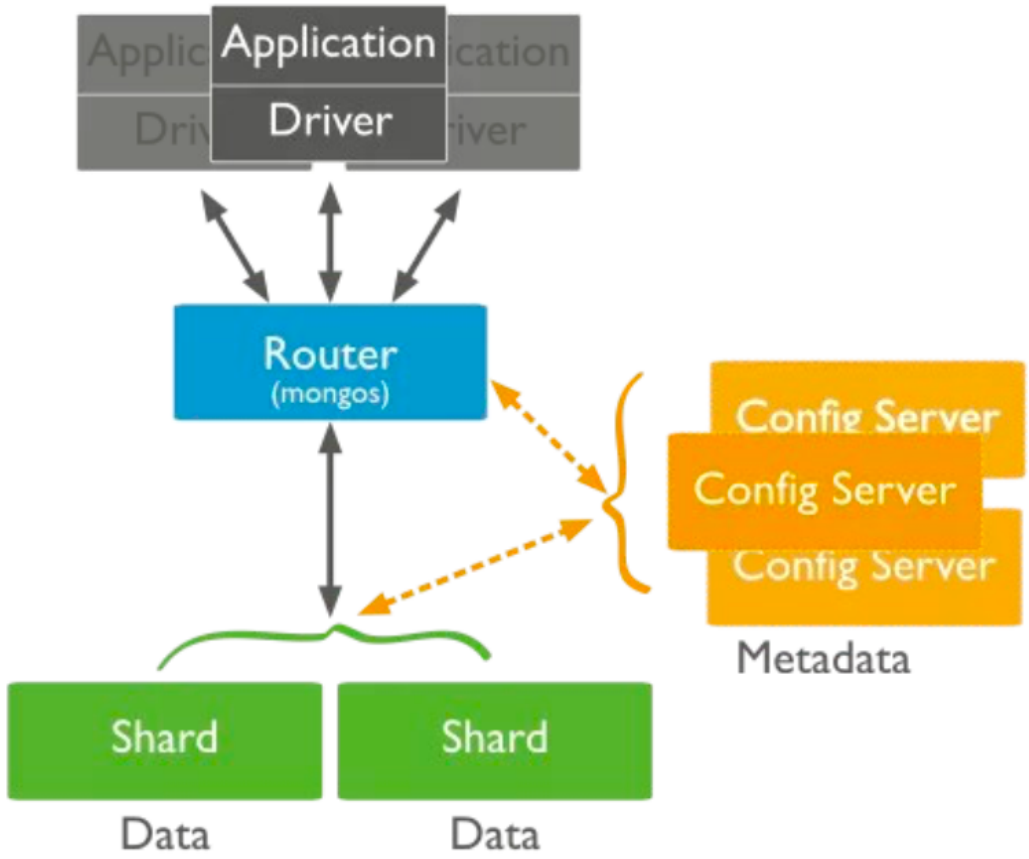
```
CREATE KEYSPACE hotel
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE hotel.address (
    street text,
    city text,
    state_or_province text,
    postal_code text,
    country text
);

CREATE TABLE hotel.hotels_by_poi (
    poi_name text,
    hotel_id text,
    name text,
    phone text,
    address frozen<address>,
    PRIMARY KEY ((poi_name), hotel_id)
) WITH comment = 'Q1. Find hotels near given poi'
AND CLUSTERING ORDER BY (hotel_id ASC) ;
```


reservation keyspace
reservations_by_hotel_date
hotel_id text K
start_date date K
room_number smallint C↑
nights smallint
confirm_number text
guest_id uuid
reservations_by_confirmation
confirm_number text K
hotel_id text C↑
start_date date C↑
room_number smallint C↑
nights smallint
guest_id uuid
address
street text
city text
state_or_province text
postal_code text
country text
reservations_by_guest
guest_last_name text K
hotel_id text C↑
room_number smallint C↑
start_date date C↑
nights smallint
confirm_number text
guest_id uuid
guests
guest_id uuid K
first_name text
last_name text
title text
{emails} text
[phone_numbers] text
<addresses> text, address

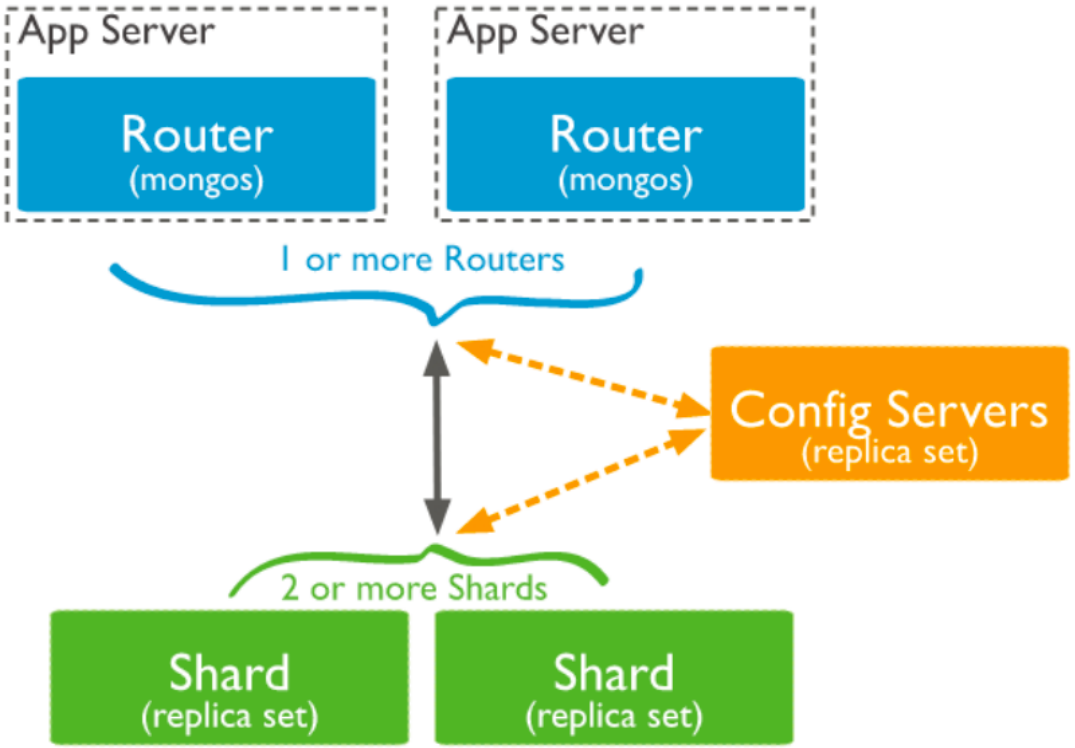
Escalabilidad de MongoDB



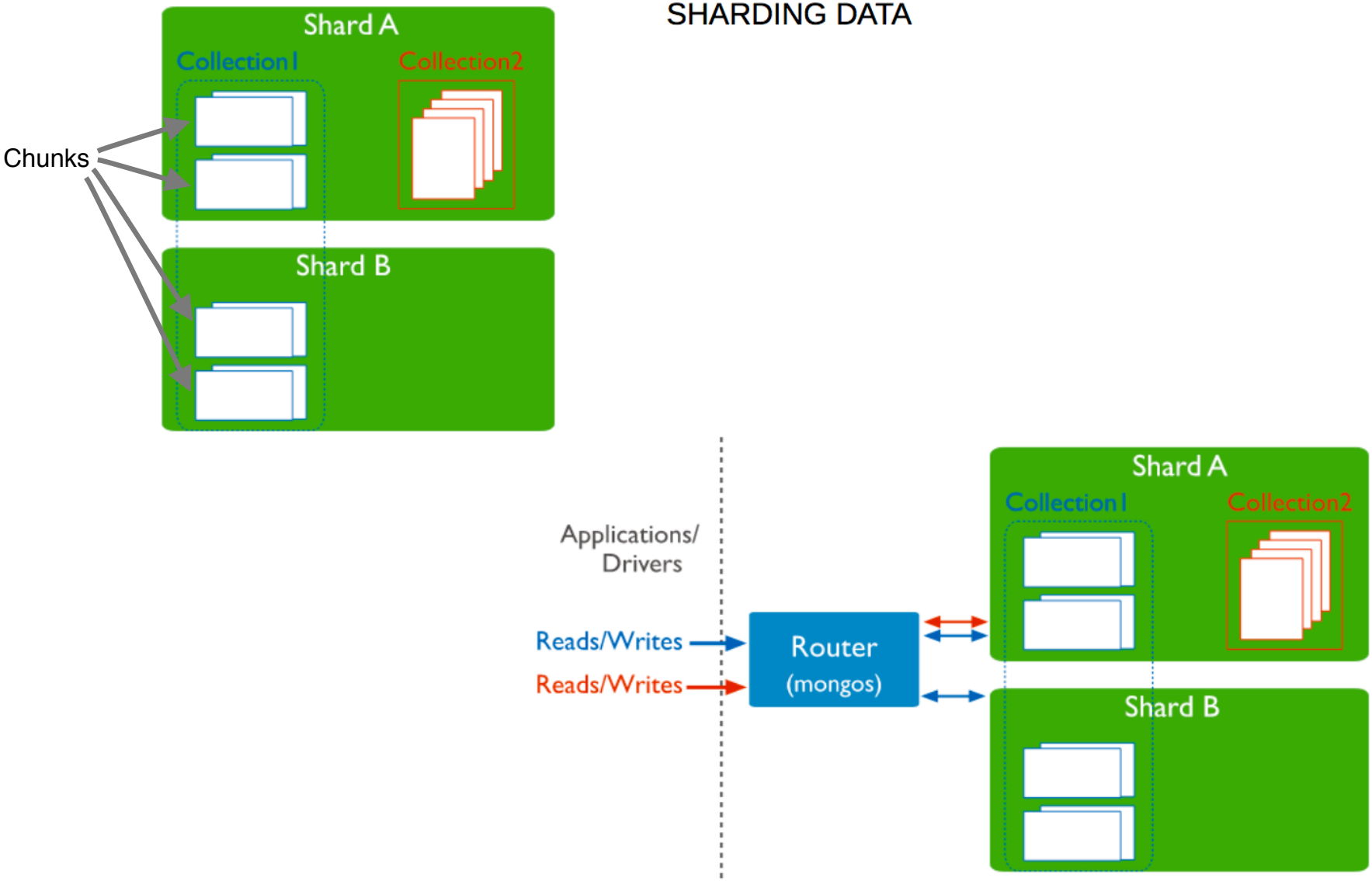
ARCHITECTURE MONGODB



SHARD CLUSTER (SHARDING DATA)

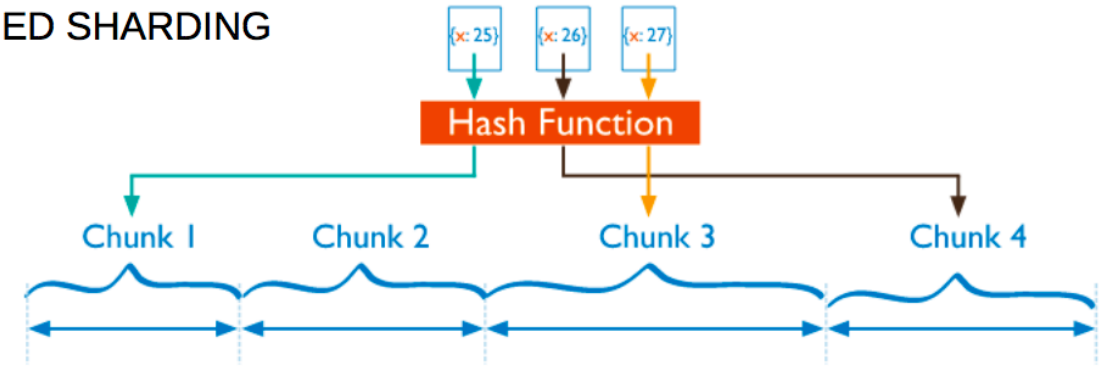


SHARDING DATA

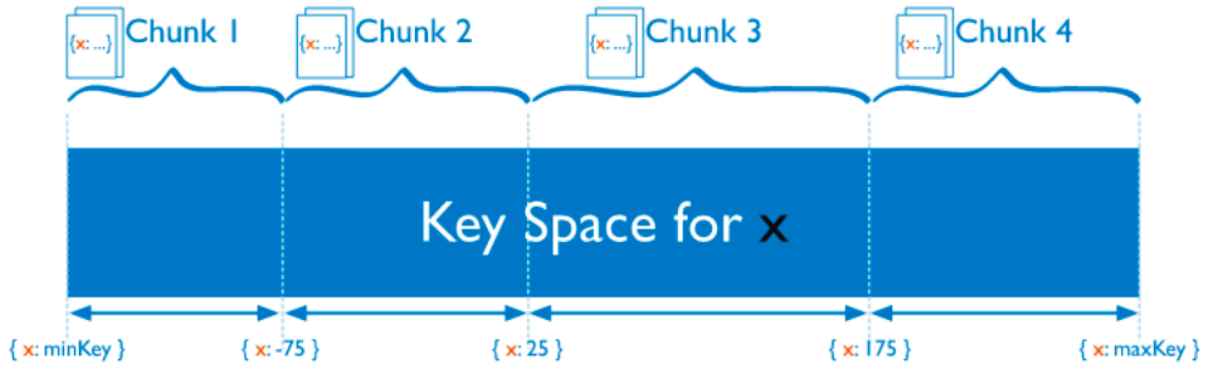


DISTRIBUTED SHARDING

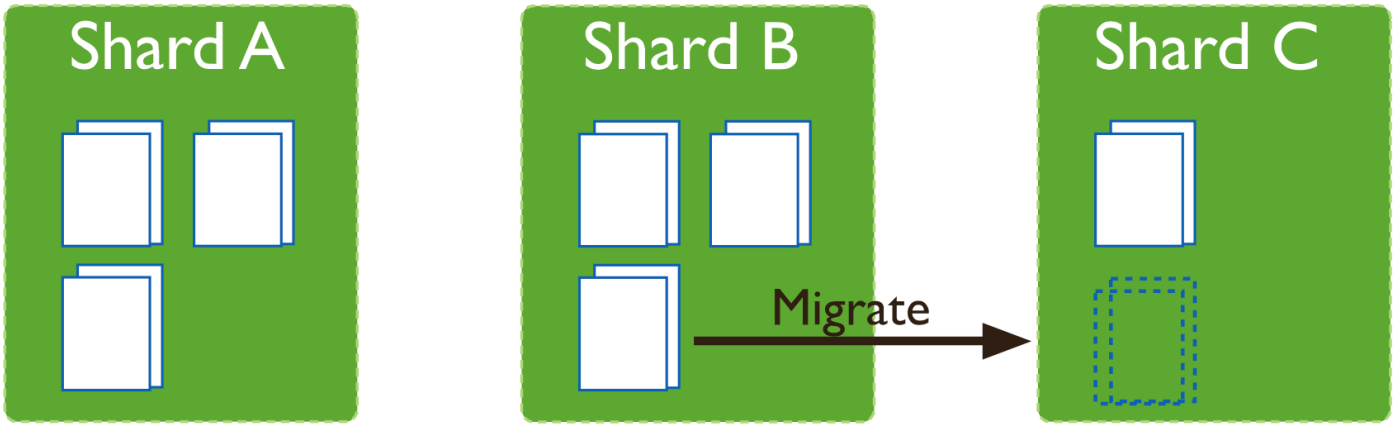
HASHED SHARDING

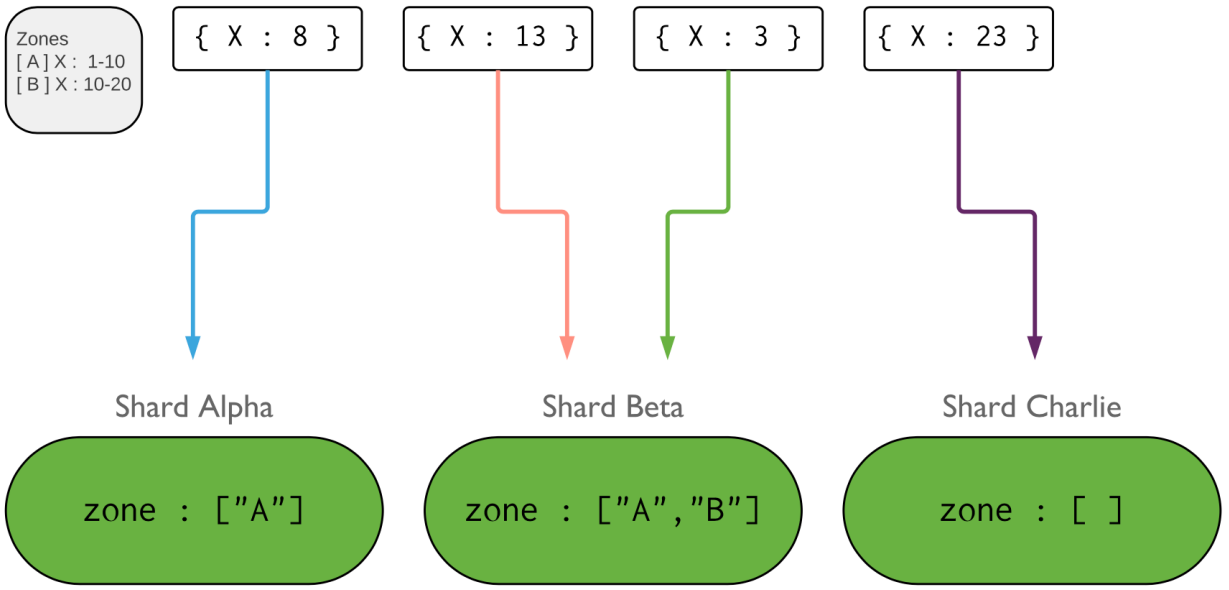


RANGED SHARDING

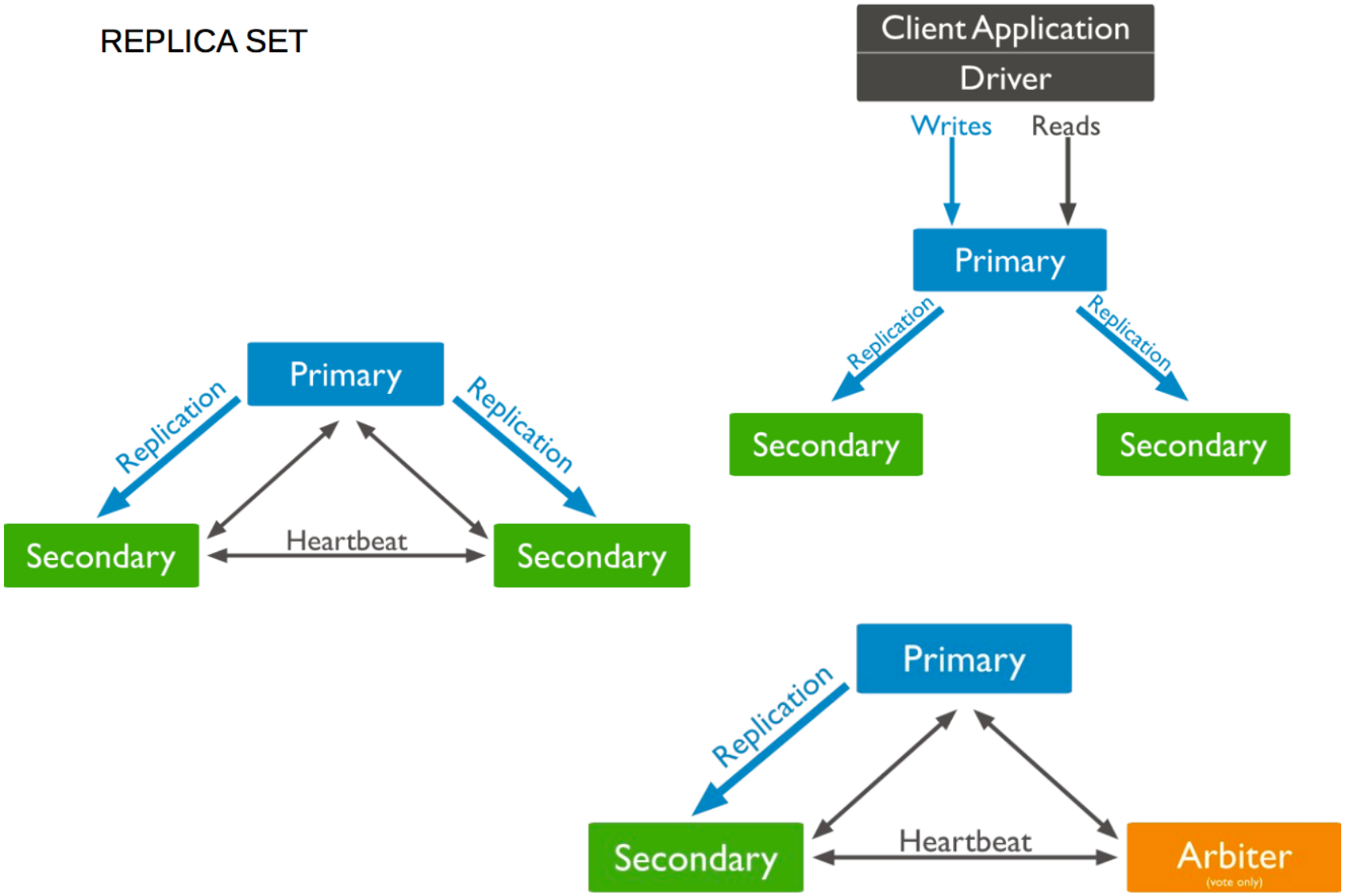


El balanceador de carga opera re-distribuyendo los ”chunks”





REPLICA SET


Client Application
Driver
Writes
Reads
Primary
Replication
Replication
Secondary
Secondary
Primary
Replication
Replication
Secondary
Heartbeat
Secondary
Primary
Replication
Secondary
Heartbeat
Arbiter
(vote only)

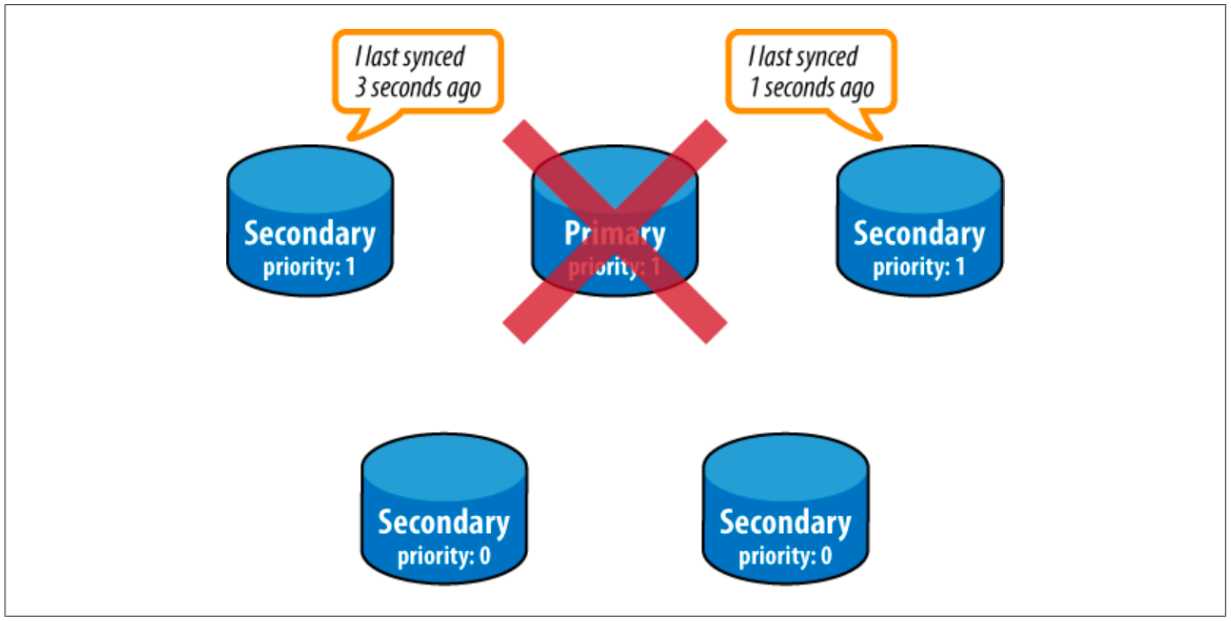


Figure 9-7. If the primary goes down, the highest-priority servers will compare how up-to-date they are

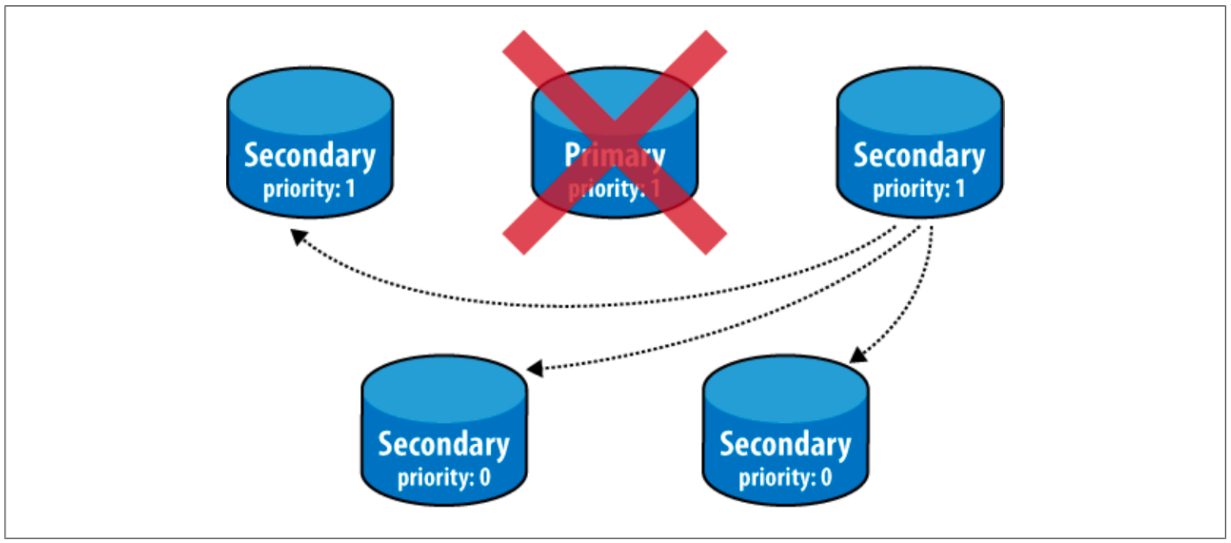
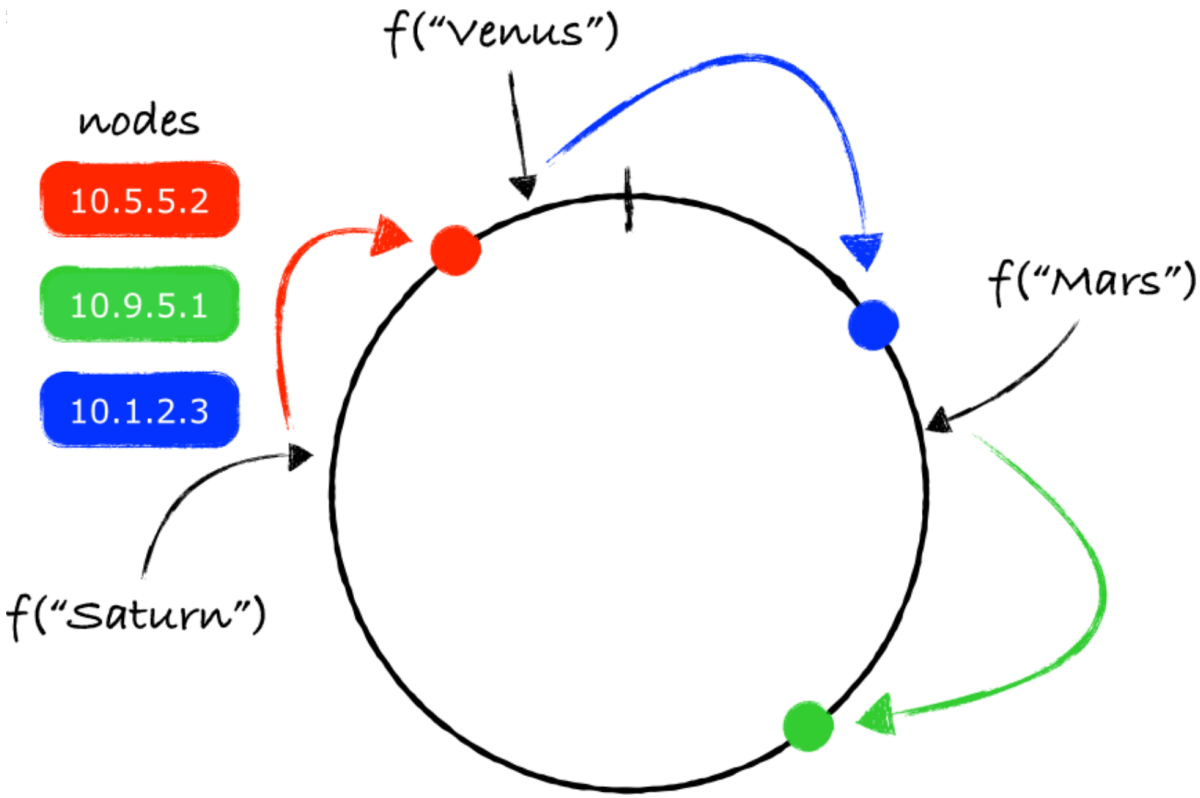


Figure 9-8. The highest-priority most-up-to-date server will become the new primary

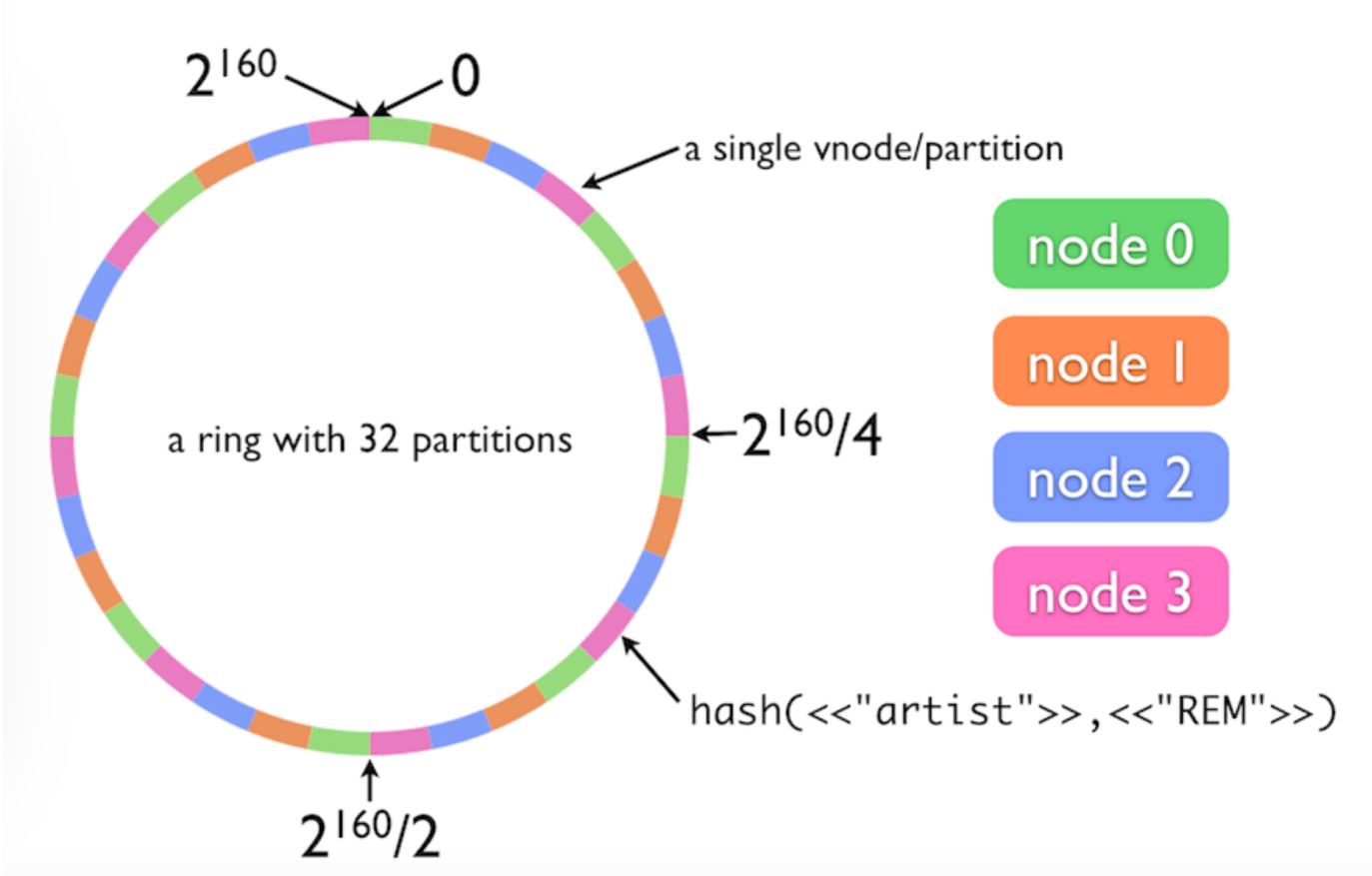
Figuras tomadas desde el libro → MongoDB: The definitive guide

Consistent Hashing



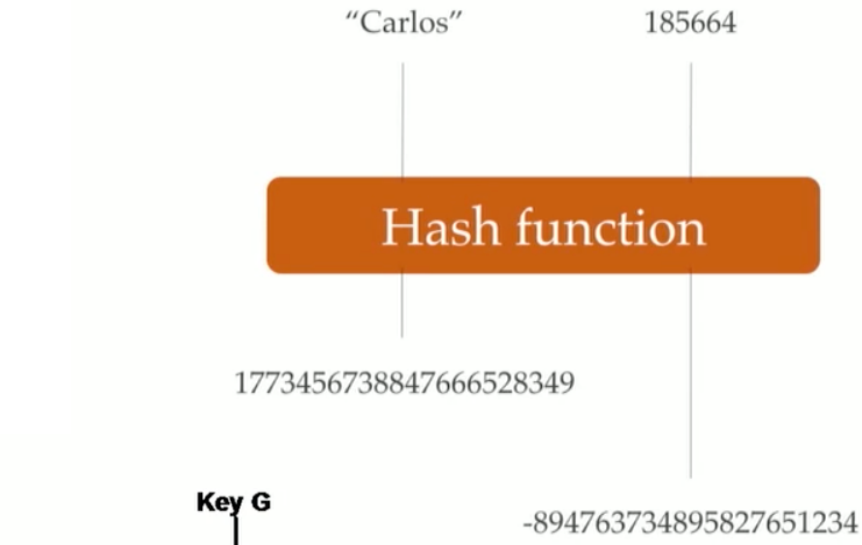
https://akshatm.svbtle.com/consistent-hash-rings-theory-and-implementation

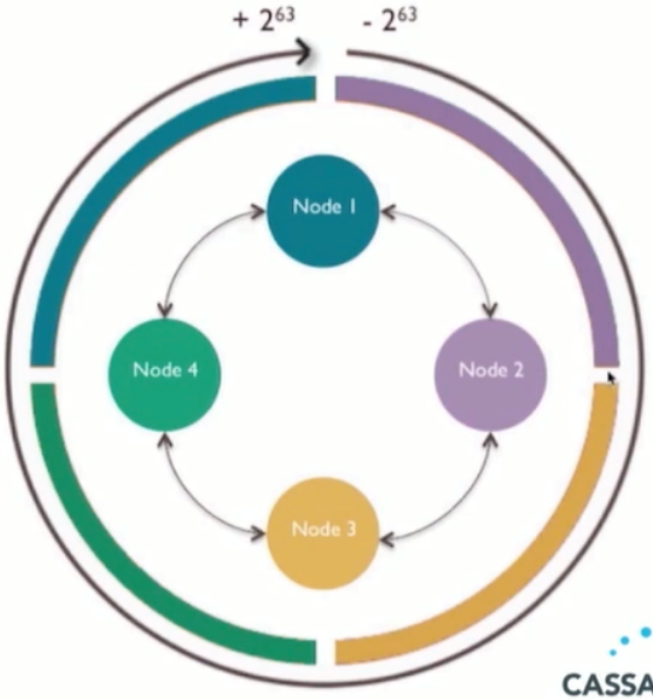
Balance de carga

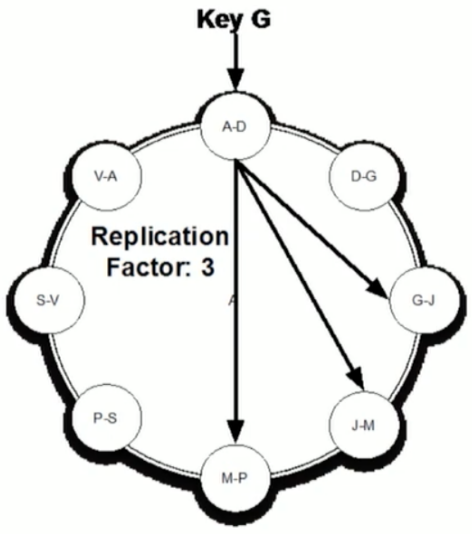


https://docs.riak.com/riak/kv/2.2.3/learn/concepts/clusters/

Consistent Hashing

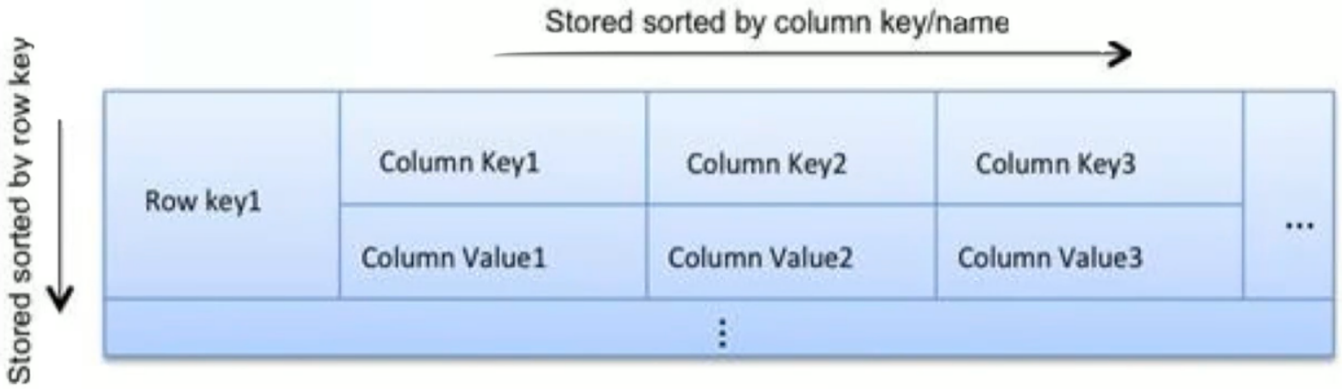






CASSANDRA SUMMIT 2016

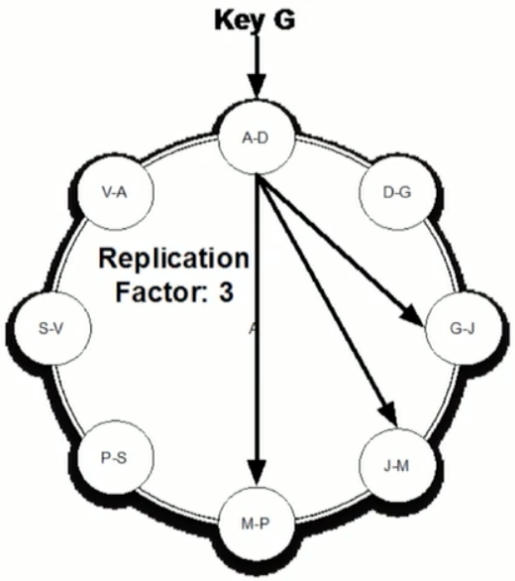
Physical Data Layout

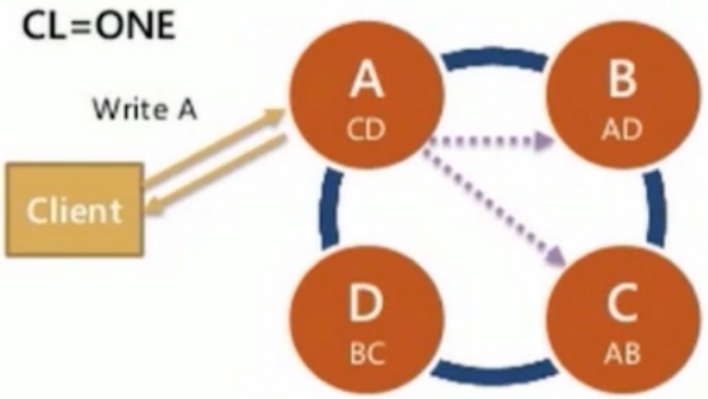


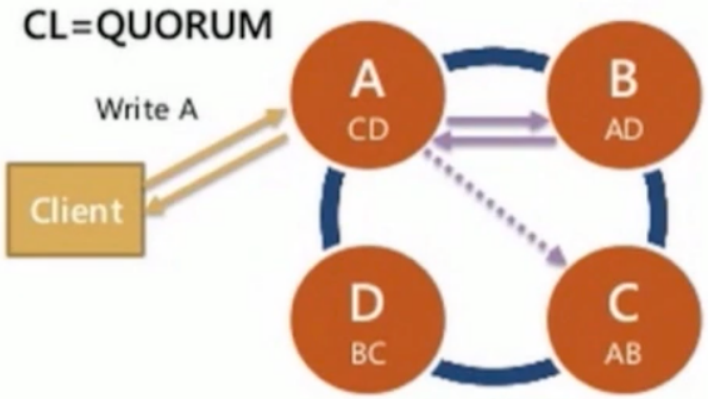
Data organisation

Consistency Level

How many replicas of your data must acknowledge?

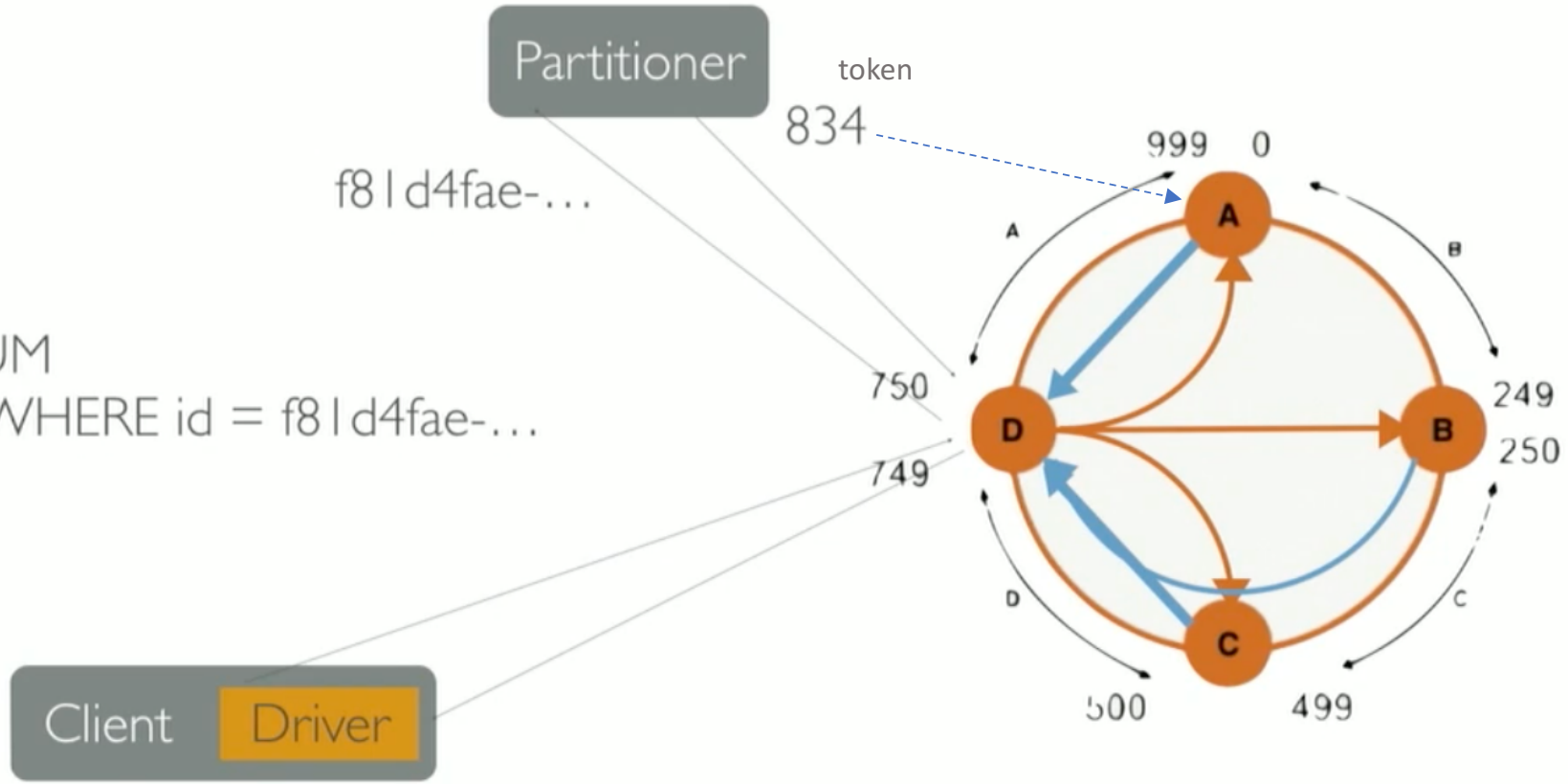






A complete read/write example

- RF = 3
- CL = QUORUM
- SELECT * … WHERE id = f81d4fae-…



Client Driver