

# Exercise Session 10 : Document Stores (MongoDB)

Agenda:

- ① RDD queries are mandatory
- ② Document stores vs. RDBMS
- ③ Indexing in MongoDB
- ④ MongoDB system design
- ⑤ MongoDB references
- ⑥ Key-value stores vs. Document stores

# Why use a document store (MongoDB) over a RDMS?

- ① Simplicity : ~ No restrictions on the kinds of data operations.
- ② Schema flexibility : Update the data model as apps. evolve. Bad for maintenance
- ③ Performance (write) : Optimized for large volumes of updates Fax numbers ~  
(No quite for MongoDB due to B-tree : Read > Write)
  - BSON format  $\Rightarrow$  efficient I/O on documents
  - No schema locking  $\Rightarrow$  Data integrity X
  - MongoDB : eventual consistency (default)
  - Data locality : Store related fields together
    - One-to-many : Nesting / embedding (RDMS : joins)

Cross machines  
over networks  
 $\Rightarrow$  perf  $\downarrow$

④ Allows for multiple (types of) secondary indices

- Single-field:

  - \* B-tree sorting

  - \* Hash

  - \* Multikey (array)

  - \* Geospatial (multidimensional)

  - \* Text

  - ...

- Compound index

⇒ Tailor the DB  
to specific workloads/  
queries

# Indexing in MongoDB

## ① Multikey index in compound index

⇒ Max. one multikey

## ② Indexing embedd fields

```
1 db.restaurants.createIndex({
2   "address.coord": 1,
3   "grades": -1
4 })
```

arrays

```
db.restaurants.insertOne(
{
  "address" : {
    "street" : "2 Avenue",
    "zipcode" : "10075",
    "building" : "1480",
    "coord" : [ -73.9557413, 40.7720266 ]
  },
  "borough" : "Manhattan",
  "cuisine" : "Italian",
  "grades" : [
    {
      "date" : ISODate("2014-10-01T00:00:00Z"),
      "grade" : "A",
      "score" : 11
    },
    {
      "date" : ISODate("2014-01-16T00:00:00Z"),
      "grade" : "A",
      "score" : 17
    }
  ],
  "name" : "Vella",
  "restaurant_id" : "41704620"
})
```

```
1 db.restaurants.createIndex({ "grades.score" : -1 })
2
3 db.restaurants.find({"grades.score" : { $eq : 11}})
4 db.restaurants.find({"grades.score" : { $gt : 11}})
```

⇒ Only effective on equality matching



### ③ Matching prefixes for compound indices.

```
1 db.restaurants.createIndex({  
2   "borough": 1,  
3   "cuisine": -1,  
4   "name" : -1  
5 })
```

⇒ Ordering of the fields matters

① Matching prefix  
② Or the inverse of

A. db.restaurants.find({"borough" : "Brooklyn"}) ✓

B. db.restaurants.find({"cuisine" : "Hamburgers"}) X

C. db.restaurants.find({"borough" : "Brooklyn", "cuisine" : "Hamburgers" }) ✓

D. db.restaurants.find().sort({"borough" : -1}) ✓

E. db.restaurants.find().sort({"borough" : 1, "cuisine" : 1}) X

F. db.restaurants.find().sort({"borough" : -1, "cuisine" : 1 }) ✓

G. db.restaurants.find().sort({"cuisine" : 1, "borough" : -1 }) X

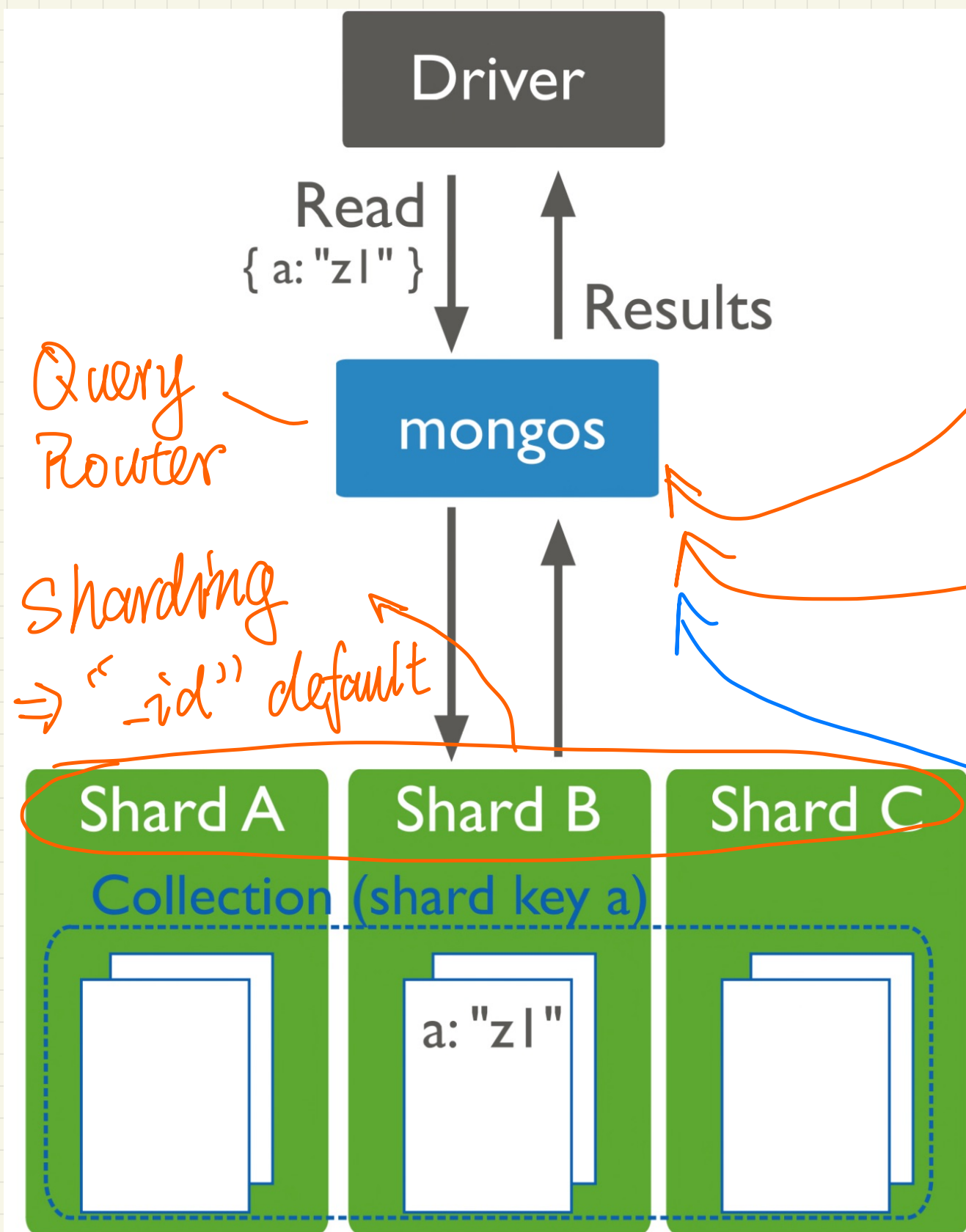
Prefix indices X

redundant.

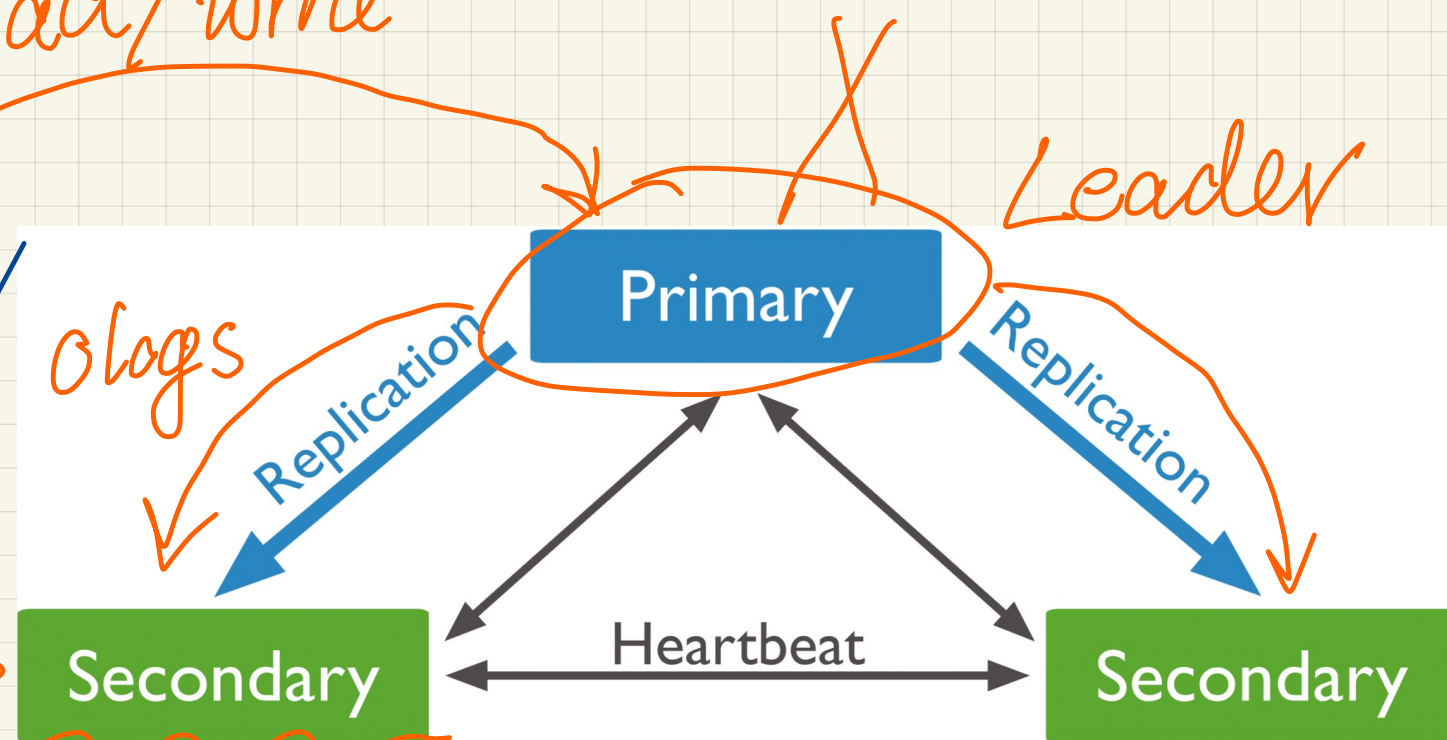
# MongoDB System Design

Single - leader replication

APP



read/write



New Leader (Primary)  
=> Downtime!

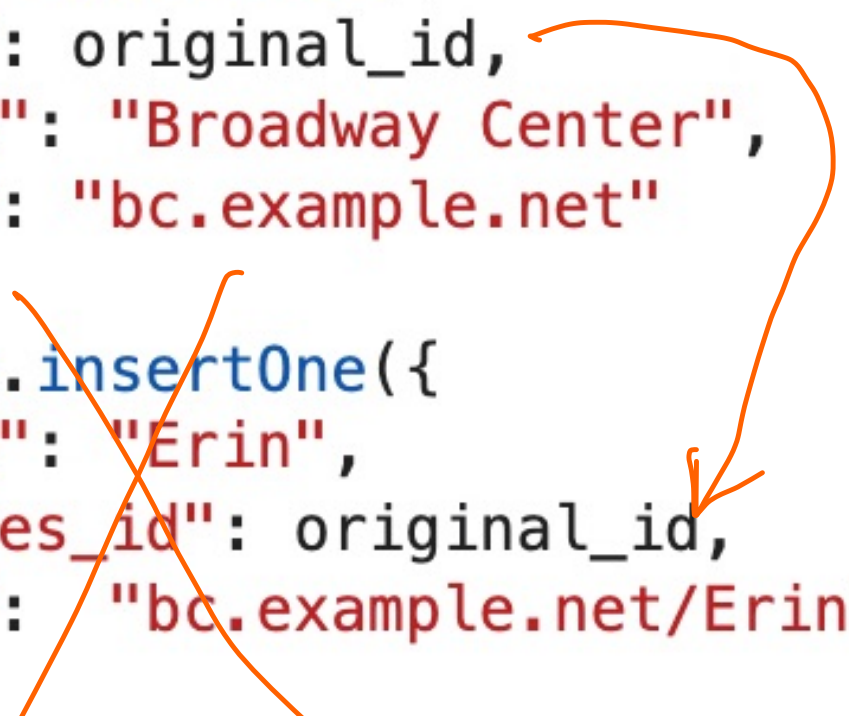
Stale info  
(Eventual consistency)

# References in MongoDB

- Why?
  - ① Normalization (~ Foreign keys in RDBMS)
  - ② Max. doc. size (16MB)
- Two types: Referential consistency NOT enforced!

## Manual Reference

```
original_id = ObjectId()
db.places.insertOne({
  "_id": original_id,
  "name": "Broadway Center",
  "url": "bc.example.net"
})
db.people.insertOne({
  "name": "Erin",
  "places_id": original_id,
  "url": "bc.example.net/Erin"
})
```



## DBRefs

⇒ cross-DB/collection referencing

```
{
  "_id" : ObjectId("123"),
  "referenced_feild" : {
    "$ref" : "collection_x",
    "$id" : ObjectId("456"),
    "$db" : "database_y",
    "extraField" : "anything"
  }
}
```




# Key-Value Stores vs. Document Stores

① Values are opaque objects in KV stores

Black boxes  $\Rightarrow$  index  $\times$  search  $\times$

② Querying options

 KV stores: Simple key-based lookups

Document stores: More complex queries.



## 1. Data Structure:

- **Document Stores:** In a document store, data is stored in flexible, semi-structured documents, often using formats like JSON or BSON (binary JSON). These documents contain fields and values, allowing nested structures and complex data types within a single document.
- **Key-Value Stores:** Key-value stores, on the other hand, store data as simple pairs of keys and values. Each value is associated with a unique key, and the store itself is agnostic to the structure of the data stored.

## 2. Schema Flexibility:

- **Document Stores:** Document stores offer more flexibility in terms of schema. Documents within the same collection can have varying structures, allowing for schema evolution and accommodating diverse data types within a single collection.
- **Key-Value Stores:** Key-value stores have little to no inherent structure or schema. Values associated with keys are opaque to the database system, offering no native way to query or manipulate the data within the values.

## 3. Querying and Indexing:

- **Document Stores:** Typically, document stores allow for more advanced querying capabilities. They often support querying based on the fields within the documents and can create indexes on specific fields to optimize queries.
- **Key-Value Stores:** Key-value stores offer basic operations centered around fetching, storing, and deleting data based on keys. They lack advanced querying capabilities and indexes beyond key-based lookups.

## 4. Data Relationships:

- **Document Stores:** Document stores often support more complex relationships between documents within the same or different collections. They can embed related data within documents or use references to establish connections between documents.
- **Key-Value Stores:** Key-value stores do not inherently support relationships between values. They store independent key-value pairs without explicit support for relationships.

## 5. Use Cases:

- **Document Stores:** Suited for use cases where flexibility in data models, complex structures within documents, and more advanced querying capabilities are required. Commonly used for content management systems, e-commerce platforms, and applications with evolving data schemas.
- **Key-Value Stores:** Ideal for scenarios where fast, simple key-based data retrieval is the primary requirement. They are used in caching systems, session stores, and applications needing high-throughput, low-latency data access.

