# Document Stores

Big Data Management

# Knowledge objectives

1. Explain the main difference between key-value and document stores
2. Explain the main resemblances and differences between XML and JSON documents
3. Explain the design principle of documents
4. Name 3 consequences of the design principle of a document store
5. Explain the difference between relational foreign keys and document references
6. Exemplify 6 alternatives in deciding the structure of a document
7. Explain the difference between JSON and BJSON
8. Name the main functional components of MongoDB architecture
9. Explain the role of "mongos" in query processing
10. Explain what a replica set is in MongoDB
11. Name the three storage engines of MongoDB
12. Explain what shard and chunk are in MongoDB
13. Explain the two horizontal fragmentation mechanisms in MongoDB
14. Explain how the catalog works in MongoDB
15. Identify the characteristics of the replica synchronization management in MongoDB
16. Explain how primary copy failure is managed in MongoDB
17. Name the three query mechanisms of MongoDB
18. Explain the query optimization mechanism of MongoDB

# Understanding objectives

1. Given two alternative structures of a document, explain the performance impact of the choice in a given setting
2. Simulate splitting and migration of chunks in MongoDB
3. Configure the number of replicas needed for confirmation on both reading an writing in a given scenario

# Application objectives

1. Perform some queries on MongoDB through the shell and aggregation framework
2. Compare the access costs given different document design
3. Compare the access costs with different indexing strategies (i.e., hash and range based)
4. Compare the access costs with different sharding distributions (i.e., balanced and unbalanced)

# Semi-structured database model

XML and JSON

# Semi-structured data

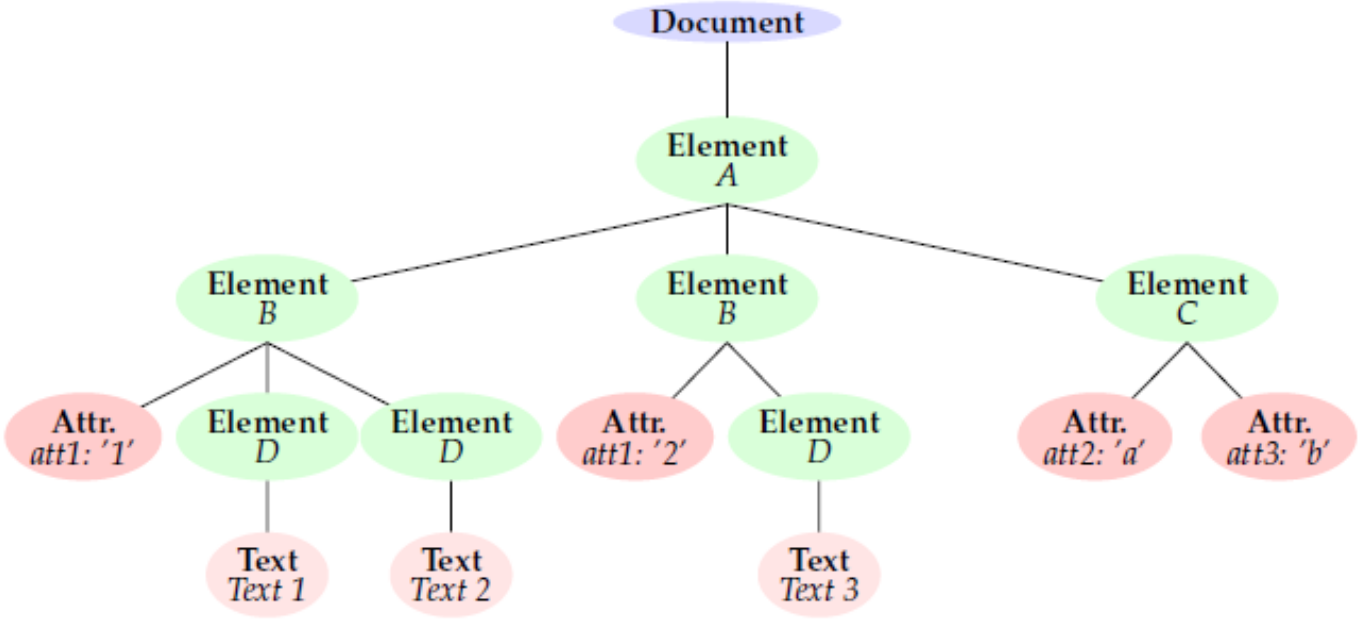
- Document stores are essentially key-value stores
  - The value is a document
    - Allow secondary indexes
- Different implementations
  - XML
  - JSON
- Tightly related to the web
  - Easily readable by humans and machines
  - Data exchange formats for REST APIs.

# XML Documents

- Tree data structure
  - Document: the root node of the XML document
  - Element: nodes that correspond to the tagged nodes in the document
  - Attribute: nodes attached to Element nodes
  - Text: text nodes, i.e., untagged leaves of the XML tree
- XML-oriented databases storage
  - *eXist-db*
  - *MarkLogic*

# XML Document Example

```
<?xml version="1.0"
      encoding="utf-8"?>
<A>
  <B att1='1'>
    <D>Text 1</D>
    <D>Text 2</D>
  </B>
  <B att1='2'>
    <D>Text 3</D>
  </B>
  <C att2="a"
     att3="b"/>
</A>
```




S. Abiteboul et al.

# Different query languages for XML data

- XPath
  - Language to address portions of an XML document (in a path form)
  - It is a subset of XQuery
- XQuery
  - Language for extracting information from collections of XML documents
- XSLT
  - Language to specify transformations (from XML to XML)
  - Mainly used to transform some XML document into XHTML, to be displayed as a Web page.

# XPath Example

```
doc('Spider-Man.xml')/movie/actor[last_name='Dunst']
```

```
<actor id='19'>
  <first_name>Kirsten</first_name>
  <last_name>Dunst</last_name>
  <birth_date>1982</birth_date>
  <role>Mary Jane Watson</role>
</actor>
```

S. Abiteboul et al.

# XQuery Example

```
for $m in collection('movies')/movie
where $m/year >= 2005
return
<film>
  {$m/title/text()},
  directed by {$m/director/last_name/text()}
</film>
```

```
<film>A History of Violence, directed by Cronenberg</film>
<film>Match Point, directed by Allen</film>
<film>Marie Antoinette, directed by Coppola</film>
```

S. Abiteboul et al.

# XSLT Example

```
<xsl:template  match="book">
   <h2>Description</h2>

  The book title is:
     <xsl:value-of select="title" />
</xsl:template>
```

```
 <h2>Description</h2>

The book title is:
   "Web Data Management and Distribution"
```

S. Abiteboul et al.

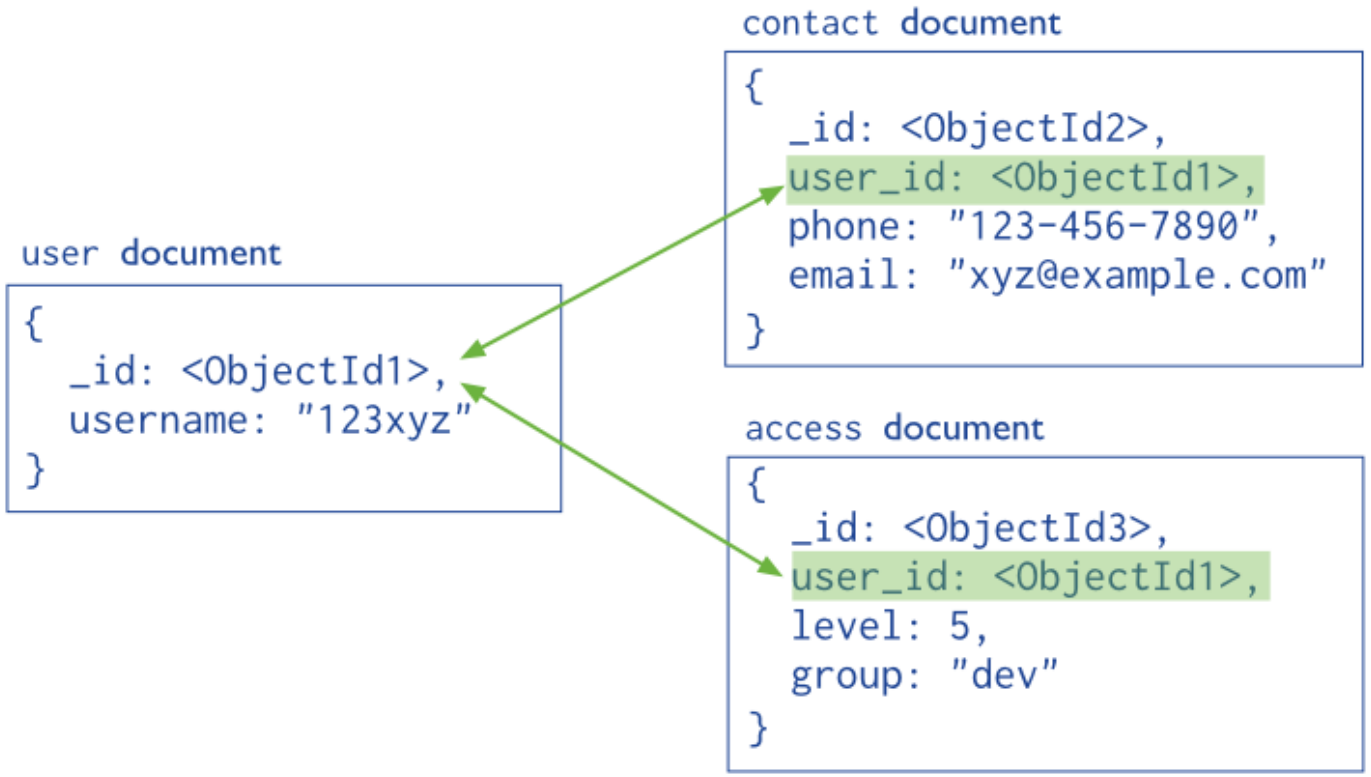
# JSON Documents

- Lightweight data interchange format
- Can contain unbounded nesting of arrays and objects
  - Brackets ([]) represent ordered lists
  - Curly braces ({}) represent key-value dictionaries
    - Keys must be strings, delimited by quotes (")
    - Values can be strings, numbers, booleans, lists, or key-value dictionaries
- Natively compatible with JavaScript
  - Web browsers are natural clients
- JSON-like storage
  - *MongoDB*
  - *CouchDB*
  - Relational extensions for *Oracle*, *PostgreSQL*, etc.

# JSON Example (I)

```json
{
  "title": "The Social network",
  "year": "2010",
  "genre": "drama",
  "country": "USA",
  "director": {
    "last_name": "Fincher",
    "first_name": "David",
    "birth_date": "1962"
  },
  "actors": [
    {
      "first_name": "Jesse",
      "last_name": "Eisenberg",
      "birth_date": "1983",
      "role": "Mark Zuckerberg"
    },
    {
      "first_name": "Rooney",
      "last_name": "Mara",
      "birth_date": "1985",
      "role": "Erica Albright"
    }
  ]
}
```

# JSON Example (II)




MongoDB

# JSON Example (III)


MongoDB

# Data structure alternatives

# Designing Document Stores

- Do not think relational-wise
  - Break 1NF to avoid joins
    - Get all data needed with one single fetch
    - Use indexes to identify finer data granularities
- Consequences:
  - Massive denormalization
  - Independent documents
    - Avoid pointers (i.e., FKs)
  - Massive rearrangement of documents on changing the application layout

# Metadata representation

*JSON*

```
{ _id: 123,
 A1: "x",
 ...
 An: "x"
}
```

*Tuple*

| _id | $A_1$ | ... | $A_n$ |
|---|---|---|---|
| 123 | "x" | ... | "x" |

# Attribute optionality

*J-Abs*

```
{ _id: 123
}
```

*J-NULL*

```
{ _id: 123,
  A1: null,
  ...
  An: null
}
```

*J-666*

```
{ _id: 123,
  A1: 666,
  ...
  An: 666
}
```

*T-NULL*

| _id | $A_1$ | ... | $A_n$ |
|---|---|---|---|
| 123 | null | ... | null |

*T-666*

| _id | $A_1$ | ... | $A_n$ |
|---|---|---|---|
| 123 | 666 | ... | 666 |

# Structure and data Types

*J-Typ*

```
{ _id: 123,
 A1: k,
 ...
 A64: k
}
```

```
{ "type": "object",
 "properties": {
  "A1": {
   "type": "number"
  },
  ...
  "An": {
   "type": "number"
   },
  required: ["A1",...,"An"]
  }
}
```

*T-Typ*

| _id | $A_1$ | ... | $A_{64}$ |
|---|---|---|---|
| 123 | k | ... | k |

# Integrity Constraints

*J-IC*

```
{ _id:    { "type": "object",
   123,     "properties": {
 A1: k,       "A1": {
 ...            "type": "number",
 A64: k         "minimum":−k′,
}               "maximum: k′},
              ...
              "An": {
                "type": "number",
                "minimum":−k′,
                "maximum: k′}
            }
          }
```

*T-IC*

| _id | $A_1$ | ... | $A_{64}$ |
|---|---|---|---|
| 123 | k | ... | k |

```
ALTER TABLE T ADD CONSTRAINT
val_A1 CHECK
(A1 BETWEEN −k′ AND k′);
...
ALTER TABLE T ADD CONSTRAINT
val_An CHECK
(An BETWEEN −k′ AND k′);
```

# Structure complexity

*Nest-one*

```
{ _id: 123,
  L1: {
    ...
    Ln: {
      An+1: k }
      ...
      }
}
```

*Nest-all*

```
{ _id: 123,
  L1: {
  ...
    Ln: {
      An+1: k,
      ...
      A64: k }
      ...
      }
}
```

*J-Arr*

```
{ _id: 123,
  A: [1,...,n]
}
```

*J-Att*

```
{  _id: 123,
  A1: k,
  ...
  An: k
}
```

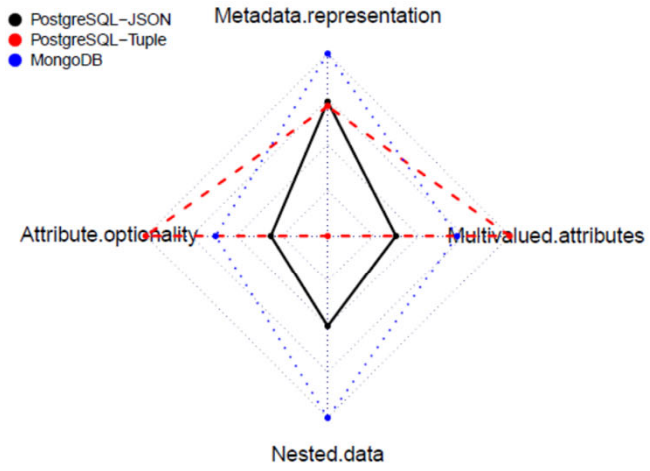
*T-Arr*

| _id | *A* |
|---|---|
| 123 | [1,...,n] |

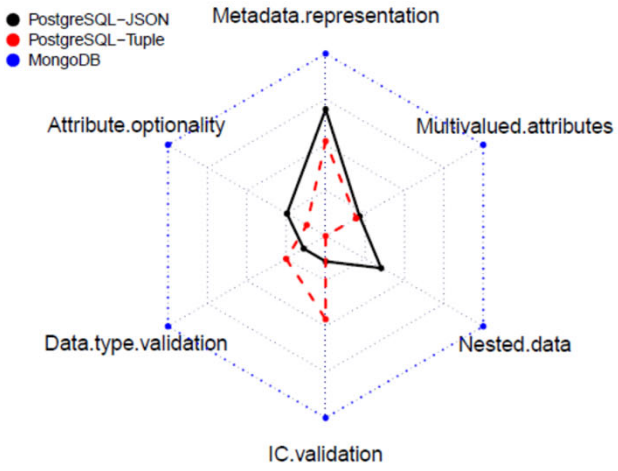
*T-Att*

| _id | $A_1$ | ... | $A_n$ |
|---|---|---|---|
| 123 | k | ... | k |

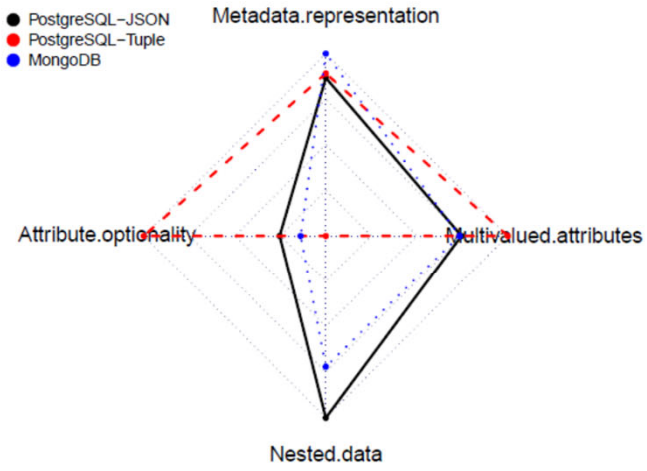
# Performance comparison




(a) Storage




(b) Insertion time




(c) Query time

# MongoDB architecture

# Abstraction

- Documents
  - *Definition*: JSON documents (serialized as BSON)
    - Basic atom
    - Identified by "*_id*" (user or system generated)
    - May contain
      - References (not FKs!)
      - Embedded documents
- Collections
  - *Definition*: A grouping of MongoDB documents
    - A collection exists within a single database
    - Collections do not enforce a schema
  - MongoDB Namespace: *database.collection*

# JSON vs. BSON (Binary JSON)

```
{
  "id": 179,
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": "2002-06-02",
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```

```
{
  "_id": ObjectId(99a88b77c66d),
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": ISODate("2002-06-02"),
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```

A. Hogan

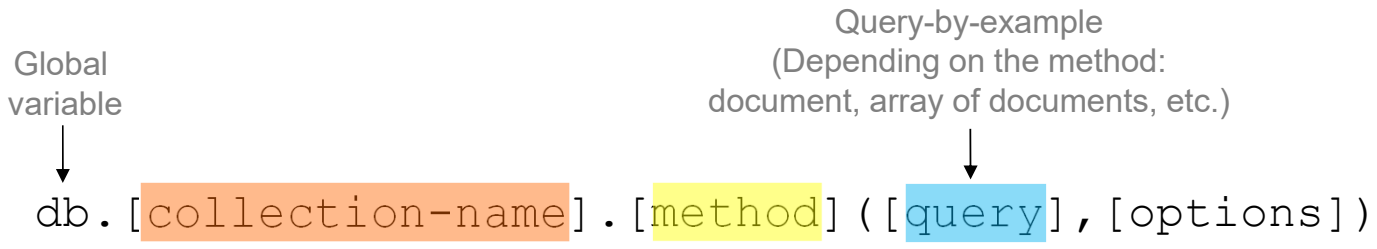
# Shell commands

- show dbs
- show collections
- show users
- use <*database*>
- coll = db.<*collection*>
- find([*criteria*], [*projection*])
- insert(*document*)
- update(*query, update, options* [e.g., upsert])
- remove(*query,* [*justOne*])
- drop()
- createIndex(*keys, options*)

- Notes:
  - *db* refers to the current database
  - *query* is a document (query-by-example)

http://docs.mongodb.org/manual/reference/mongo-shell

# MongoDB syntax




- Collection methods: insert, update, remove, find, …

```
db.restaurants.find({"name": "x"})
```

- Cursor methods: forEach, hasNext, count, sort, skip, size, ...

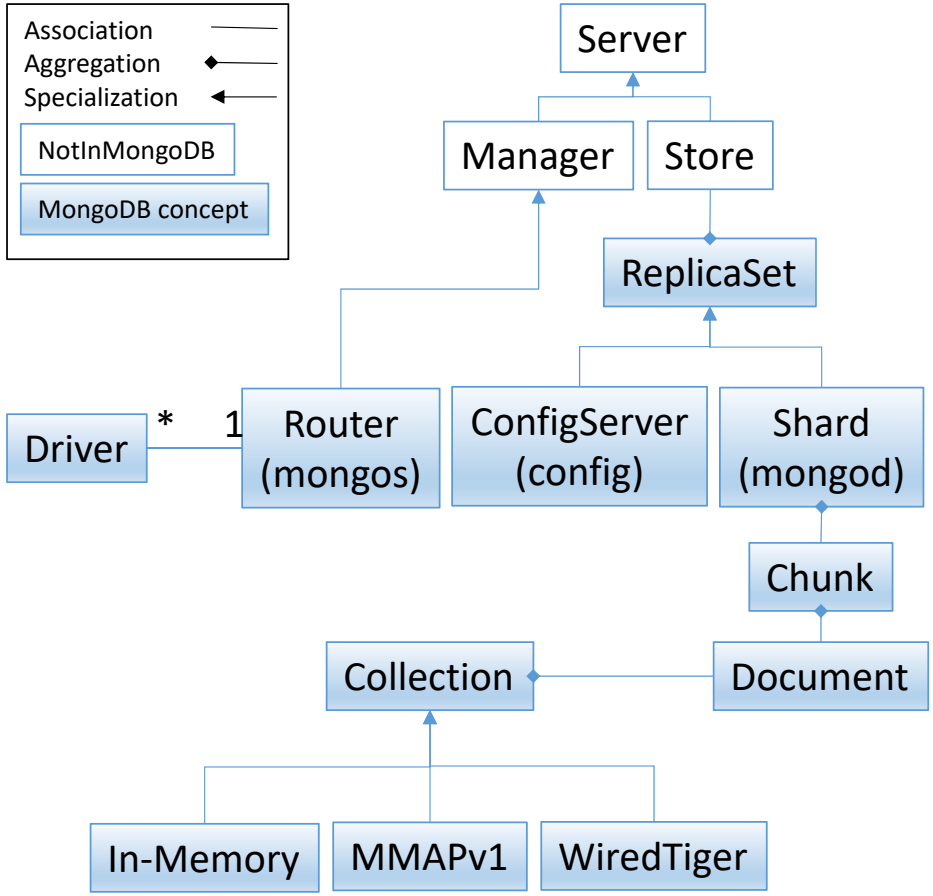
```
db.restaurants.find({"name": "x"}).count()
```

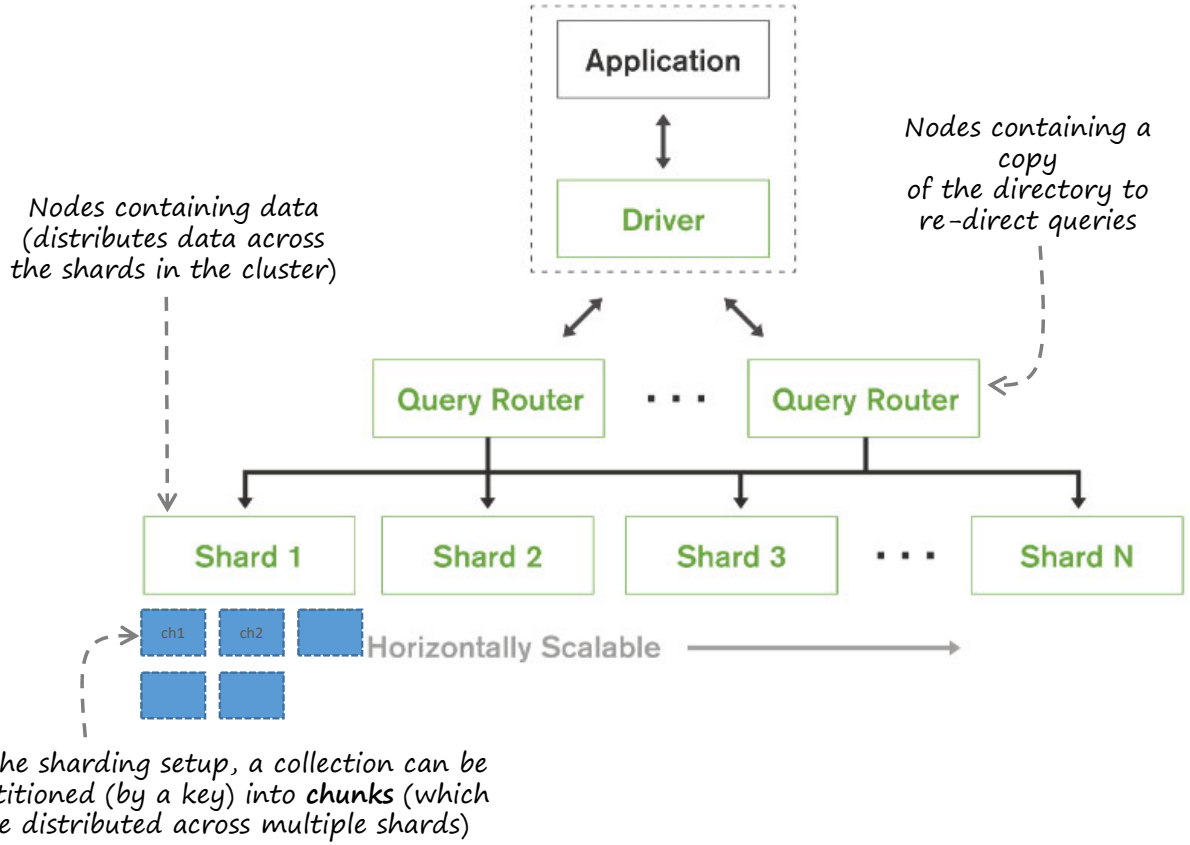
- Database methods: createCollection, copyDatabase, ...

```
db.createCollection("collection-name")
```

- …

# MongoDB functional components



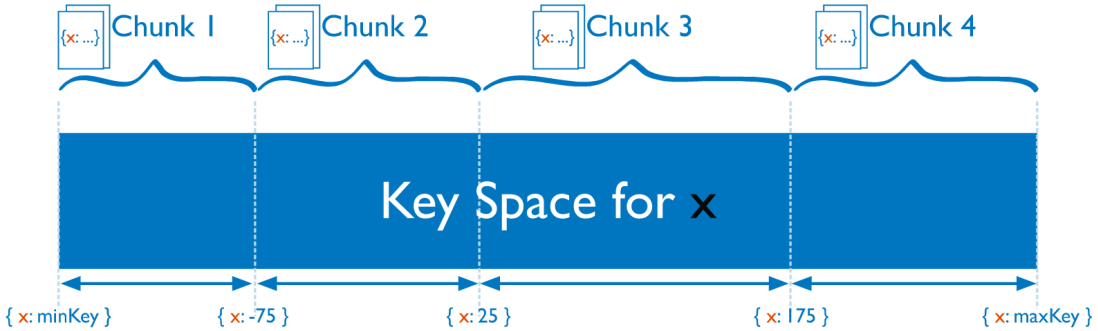

https://docs.mongodb.com/manual/core/sharded-cluster-components

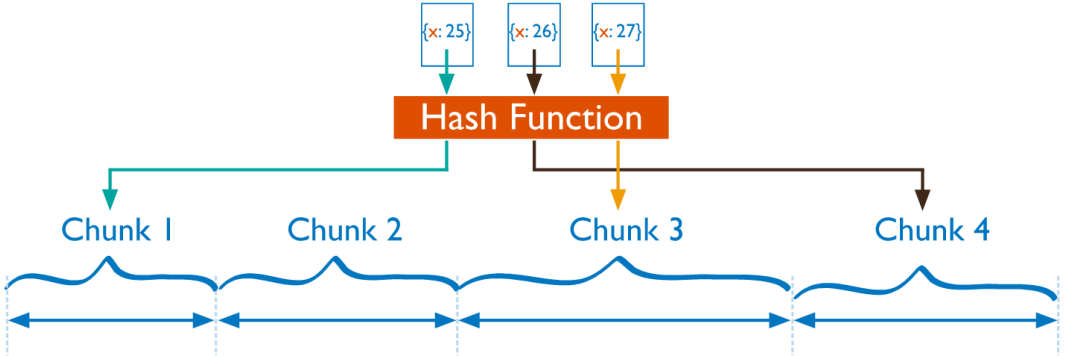
# Distributed Data Design

Challenge I

# Sharding (horizontal fragmentation)

- Shard key
  - Must be indexed (*sh.shardCollection( namespace, key)*)
  - If not existing in a document, treated as null
- Chunk (64MB)
  - Horizontal fragment according to the shard key
    - Range-based: Range of values determines the chunks
      - Adequate for range queries
    - Hash-based: Hash function determines the chunks
      - Consistent hashing

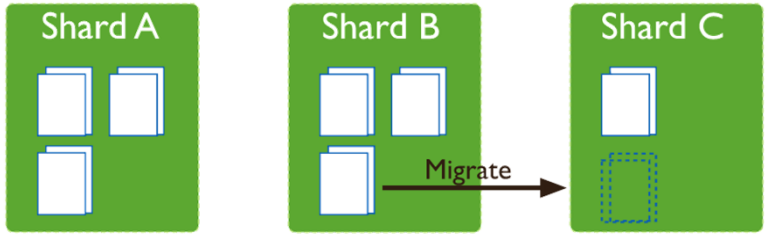



https://docs.mongodb.com/manual/core/ranged-sharding/#sharding-ranged
https://docs.mongodb.com/manual/core/hashed-sharding/#sharding-hashed

# Splitting and migrating chunks

- Inserts and updates above a threshold trigger splits
  - Not in single-key chunks
- Uneven distributions in the number of chunks per shard trigger migrations
  1. A new chunk is created in an underused shard
  2. Per document requests are sent to the origin shard
  3. Origin keeps working as usual
     - Changes made during the migration are applied *a posteriori* in the destination shard
  4. Changes are annotated made in the config servers, which enables the new chunk
  5. Chunk at origin is dropped
  6. Query routers are eventually synchronized




https://docs.mongodb.com/manual/core/sharding-balancer-administration/#sharding-balancing

# Distributed Catalog Management

Challenge II

# Catalog structure

- Content
  - List of chunks in every shard
- Implemented in a replica set (as any other data)
- Client cache in the routers
  - Lazy/Primary-copy replication maintenance

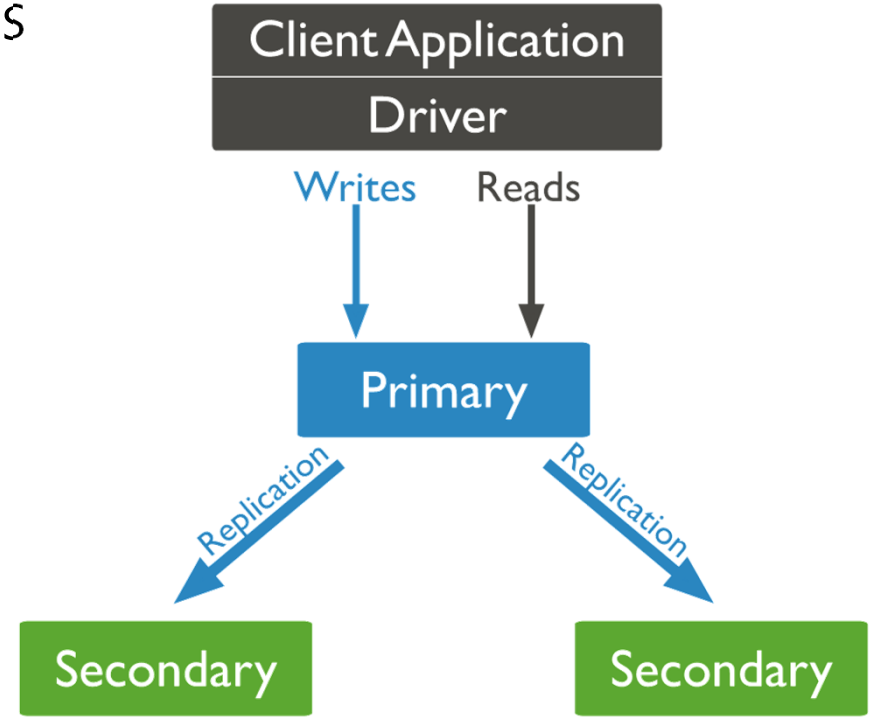
https://docs.mongodb.com/manual/core/sharded-cluster-config-servers

# Distributed Transaction Management

Challenge III

# Replica sets

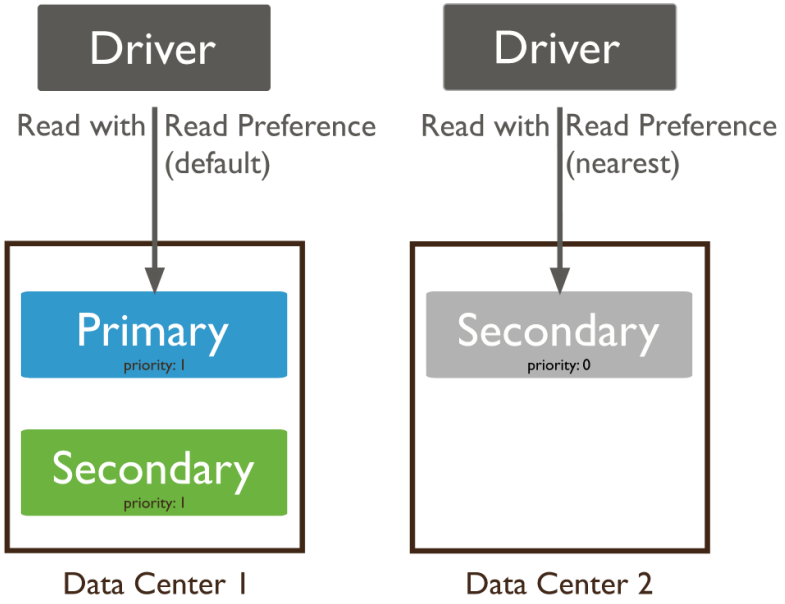
- A replica set is a set of *mongod* instances
- Primary copy with lazy replication
  - One primary copy
    - Inserts, writes, updates
    - Reads
  - Secondary copies
    - Reads




MongoDB

# Read preference

- By default, applications will try to read the primary replica
- It can also specify a read preference
  - primary
  - primaryPreferred
  - secondary
  - secondaryPreferred
  - nearest
    - Least network latency



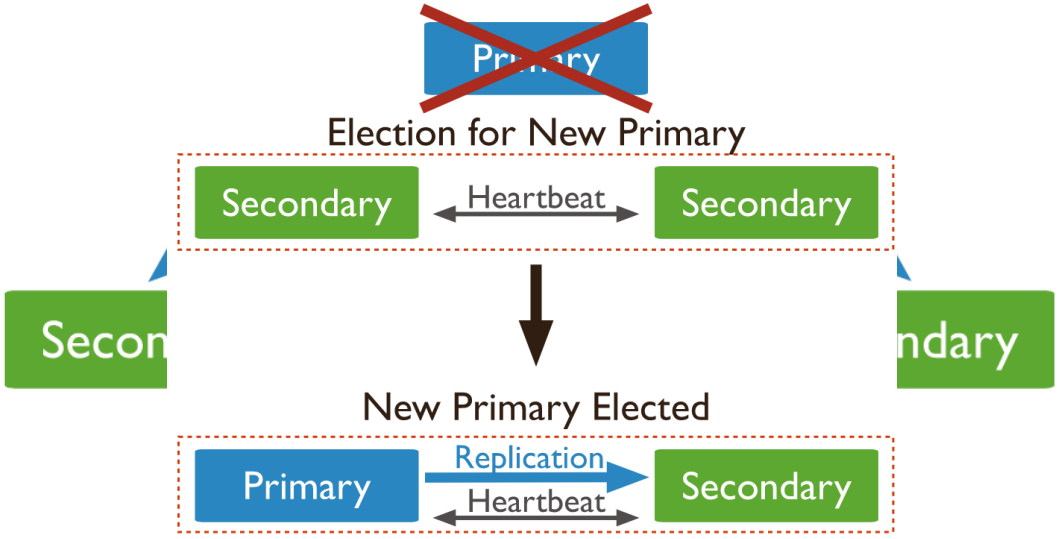

MongoDB

# Required read and writes

- ReadConcern
  - Specifies how many copies need to be read before confirmation
    - They should coincide
- WriteConcern
  - Specifies how many copies need to be writen before confirmation
    - Might be zero

# Handling failures

- Heartbeat system
  - Failure → Primary does not communicate with the other members for 10sec
- New primary is decided based on consensus protocols
  - PAXOS




MongoDB

# Distributed Query Processing

Challenge IV

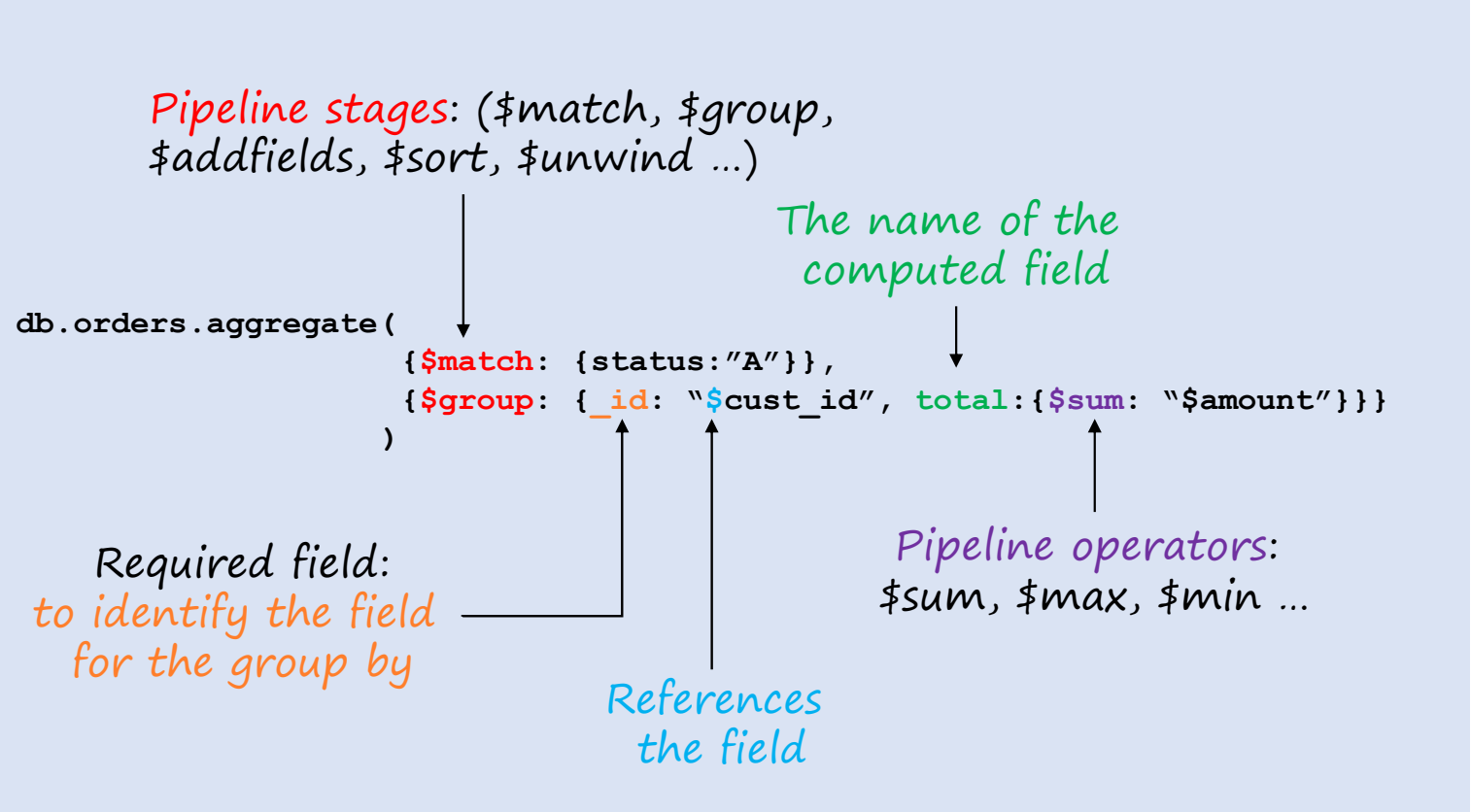
# Query mechanisms

a) JavaScript API
   - *find* and *findOne* methods (Query By Example)
     - db.collection.find()
     - db.collection.find( *{ qty: { $gt: 25 } }* )
     - db.collection.find( *{ field: { $gt: value1, $lt: value2 } }* )

b) Aggregation Framework
   - Documents enter a multi-stage pipeline that transforms them
     - Filters that operate like queries
     - Transformations that reshape the output document
     - Grouping
     - Sorting
     - Other operations

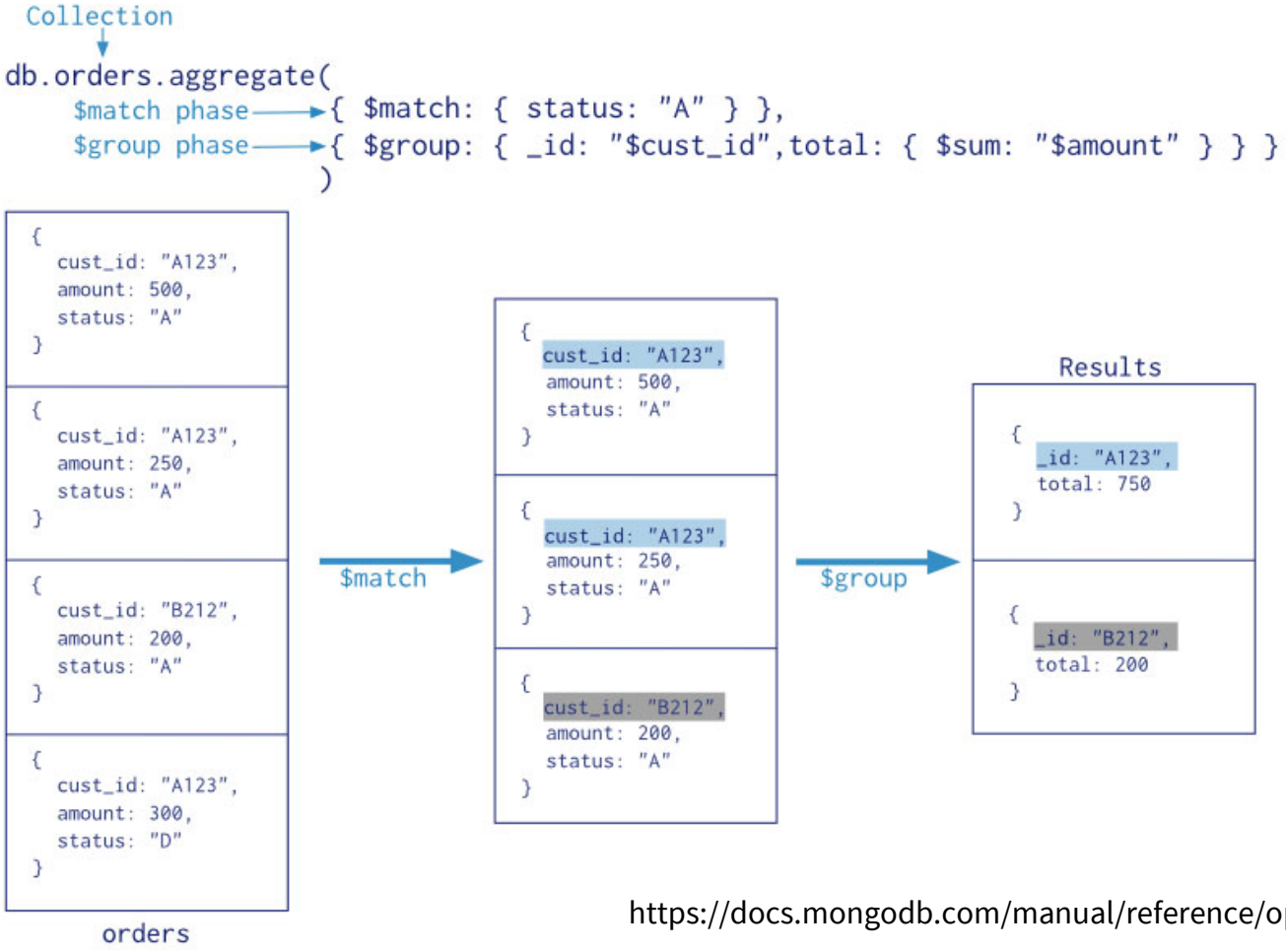
c) MapReduce

# Aggregation Framework Syntax



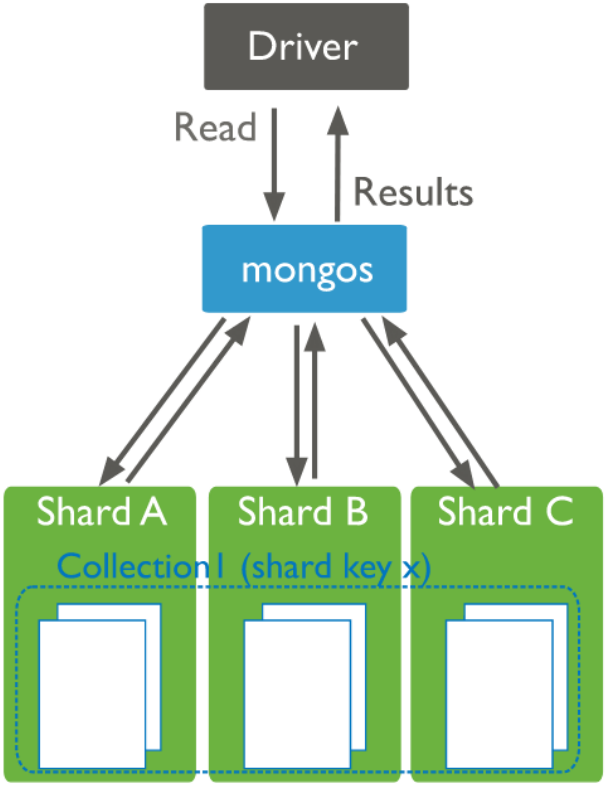

https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline

# Aggregation Framework Steps




https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline

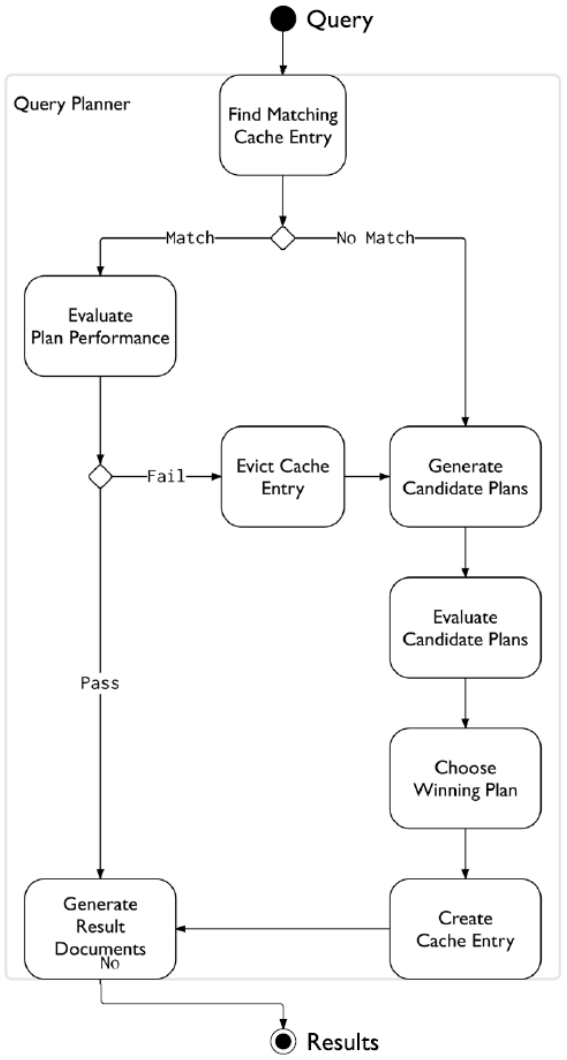
# Query routing




https://docs.mongodb.com/manual/core/sharded-cluster-query-router

# Indexing

- Kinds
  - B+
  - Hash
  - Geospatial
  - Text
- Usage
  - Best plan is cached
  - Performance is evaluated on execution
    - New candidate plans are evaluated for some time
- Allow
  - Multi-attribute indexes
  - Multi-valued indexes
    - On arrays
  - Index-only query answering




https://www.docs4dev.com/docs/en/mongodb/v3.6/reference/core-query-plans.html

# Closing

# Summary

- Document-stores
  - Semi-structured database model
  - Indexing
- MongoDB
  - Architecture
  - Interfaces

# References


- E. Brewer. *Towards Robust Distributed Systems*. PODC'00
- L. Liu and M.T. Özsu (Eds.). *Encyclopedia of Database Systems*. Springer, 2009
- S. Abiteboul et al. *Web Data Management*. Cambridge University Press, 2012
- A. Hogan: *Procesado de Datos Masivos*. U. de Chile.

  http://aidanhogan.com/teaching/cc5212-1-2020