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
  - eXtensible Markup Language (XML)
  - JavaScript Object Notation (JSON)
- Tightly related to the web
  - Easily readable by humans and machines
  - Data exchange formats for REST APIs

The term semi-structured describes data that have some structure but is neither regular nor known a priori
Called self describing data

The impedance mismatch, a challenge in RDBMS, is mitigated by NoSQL systems with their flexible data models and query languages

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*
  - Relational extensions for *Oracle*, *PostgreSQL*, etc.

No limit for nesting

Cannot be nested

That allows us to store XML

Attribute can also be written as separate element

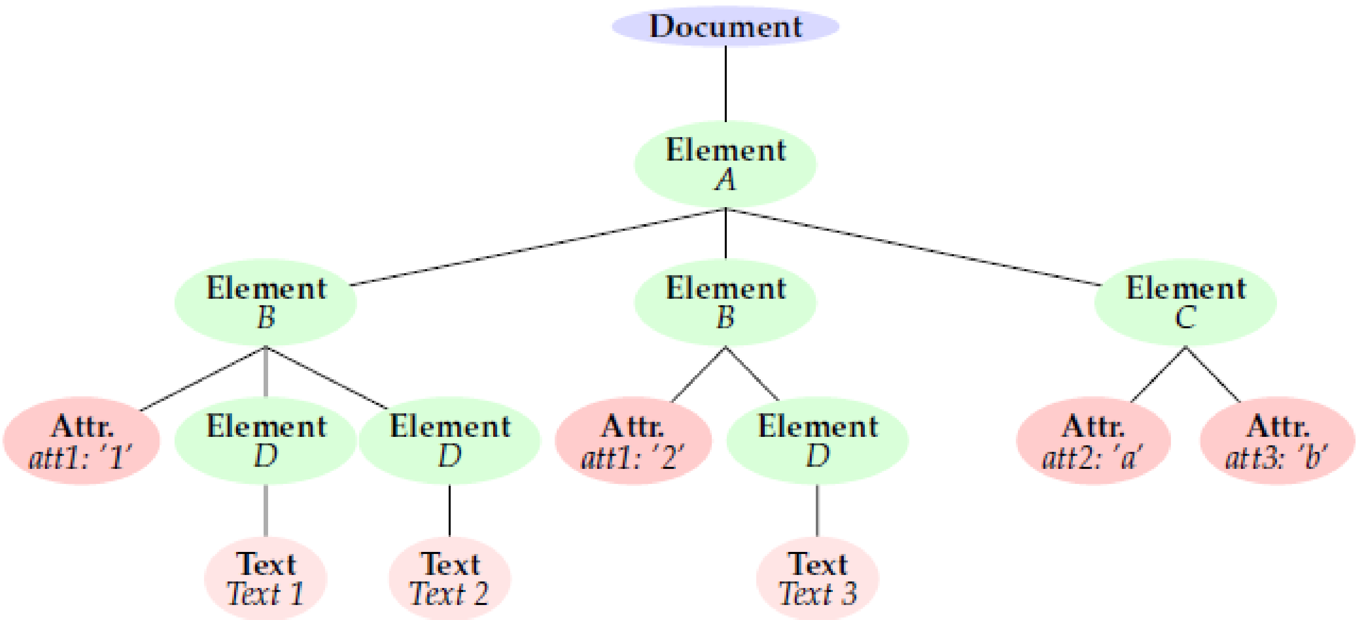
XML Document Example

While XML standard suggests that elements within a sequence must be ordered, in practical implementations, this order is often not enforced.

Header indicating file contains XML document

```
<?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>
```

Attribute



S. Abiteboul et al.

Different query languages for XML data

- XPath allows to access part of document
  - Language to address portions of an XML document (in a path form)
  - It is a subset of XQuery
- XQuery allows to access information from document
  - Language for extracting information from collections of XML documents
- XSLT allows to transform XML into HTML
  - 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 (a.k.a. finite maps)
    - 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)

```
{
  "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)

_id -> special key
Does not check constraints
They do not encourage to use this
Instead embed document one inside another

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

MongoDB

JSON Example (III)

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

Embedded sub-document

Embedded sub-document

MongoDB

Schema variability
1. Metadata embedding
2. Attribute optionality

Schema declaration
1. Structure and data types
2. Integrity constraints

Structure complexity
1. Nested structures
2. Multi-valued attributes

Data structure alternatives

Metadata Representation
Attribute optionality
Structure and data type
Integrity constraint
Structure Complexity - In relational database, nesting is not possible. Data needs to be flattened. But it is possible in JSON. But, takes high space

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

Index can be used to fetch required document

Consequences:

Duplicates

- Massive denormalization
- Independent documents

Data is stored in independent document

  - Avoid pointers (i.e., either FKs or references)

There are no foreign key. There might be reference
Reference is store identifier of one document to another
Foreign key - has constraint check

- Massive rearrangement of documents on changing the application layout

Document layouts may need to be frequently rearranged to accommodate changes in application requirements

Metadata representation

Attribute names are stored inside the document. For each document this attribute name is repeated. But in the case of relational databases schema is not repeated for each tuple.

One way is to shorten attribute names. But if this document needs to be shared with others, people might get confused with shortened attribute names as it might lack clarity.

JSON

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

Tuple

_id	A_1	...	A_n
123	"x"	...	"x"

Attribute optionality

Do not mention attributes which are not required

Not good practice

J-Abs

```
{ _id: 123
}
```

J-NULL

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

Not good practice

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

Not good practice

Structure and Data Types

To validate structure and datatypes, JSON Schema 2 uses the properties key.
required key - list of expected value

JSON Type

```
{ _id: 123,       { "type": "object",
  A1: k,            "properties": {
  ...                 "A1": {
  An: k                 "type": "number"
}                     },
                      ...
                      "An": {
                        "type": "number"
                      },
                    required: ["A1",...,"An"]
                    }
                  }
```

Tuple Type

_id	A_1	...	A_n
123	k	...	k

```
CREATE TABLE T (
_id INTEGER,
A1 INTEGER,
...
An INTEGER,
);
```

In relational database, it is mandatory to define data type prior to data insertion
In JSON, this can be done, but is not mandatory
If schema is defined, at insertion time this check is preformed
else nothing will be changed

Integrity Constraints

JSON-IC

```
{ _id: 123,      { "type": "object",
  A1: k,           "properties": {
  ...                "A1": {
  An: k                "type": "number",
}                      "minimum":-k',
                       "maximum: k'},
                     ...
                     "An": {
                       "type": "number",
                       "minimum":-k',
                       "maximum: k'}
                   }
                 }
```

Tuple-IC

_id	A_1	...	A_n
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');
```

In relational DB, we can add integrity check constraints
This can also be done in JSON, but it is optional

Structure complexity

JSON-Nest

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

JSON-Array

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

JSON-Attrib.

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

Tuple-Array

_id	A
123	[1,...,n]

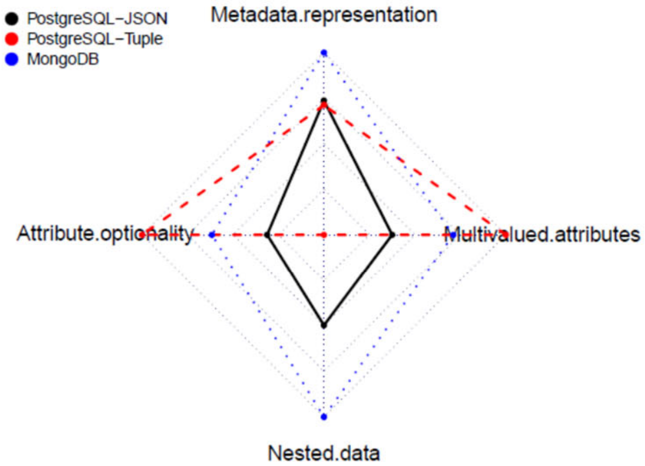
Tuple-Attrib.

_id	A_1	...	A_n
123	k	...	k

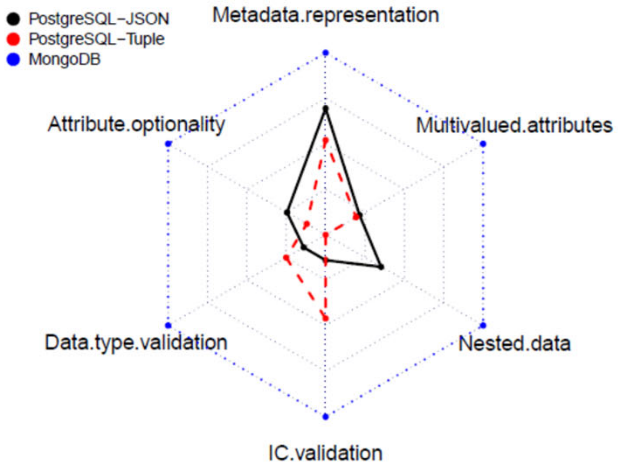
In relational database, nesting is not possible. Data needs to be flattened
But it is possible in JSON. But, takes high space

Performance comparison

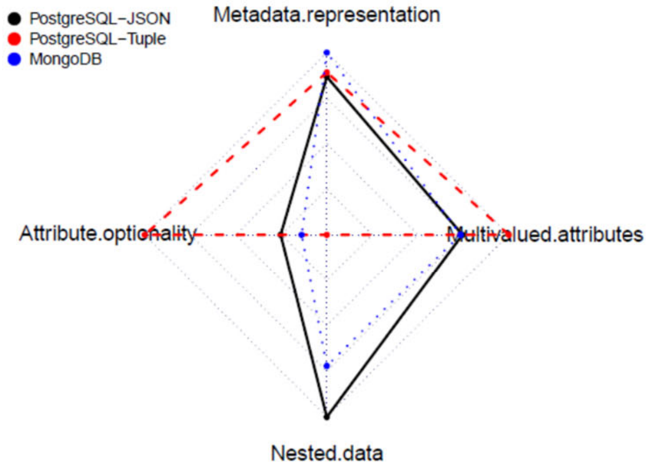
Larger the area, better



(a) Storage



(b) Insertion time



(c) Query time

Better -
PostgreSQL (Tuple) Attribute not repeated

If datatype, constraint present, need to check it
Better
MongoDB (no datatype, no constraint check)

Best
PostgreSQL - Tuple (Related to storage as well - Store less data, No nesting, Query Optimizer)

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 Do not require to define schema although it can be done
  - MongoDB Namespace: *database.collection*

JSON vs. BSON (Binary JSON)

There is some serialization processing going before storing document

```
{
  "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
  }
}
```

What we write

```
{
  "_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
  }
}
```

Stored in 32 or 64 bits

What is written in the disk

A. Hogan

Shell commands

- show dbs show databases
- show collections show collections
- show users
- use <*database*>
- coll = db.<*collection*> define variable, this collection in this db is called x
- find([<*criteria*>], [<*projection*>]) Query by example
- insert(<*document*>)
- update(<*query*>, <*update*>, <*options* [e.g., upsert]>) query = pattern, where condition update = what to update
- remove(<*query*>, [*justOne*]) Delete one or all
- drop() drop collection
- createIndex(<*keys*>, <*options*>) create secondary index

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

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

MongoDB syntax

Global variable

Query-by-example (Depending on the method: document, array of documents, etc.)

```
db.[collection-name].[method]([query],[options])
```

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

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

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

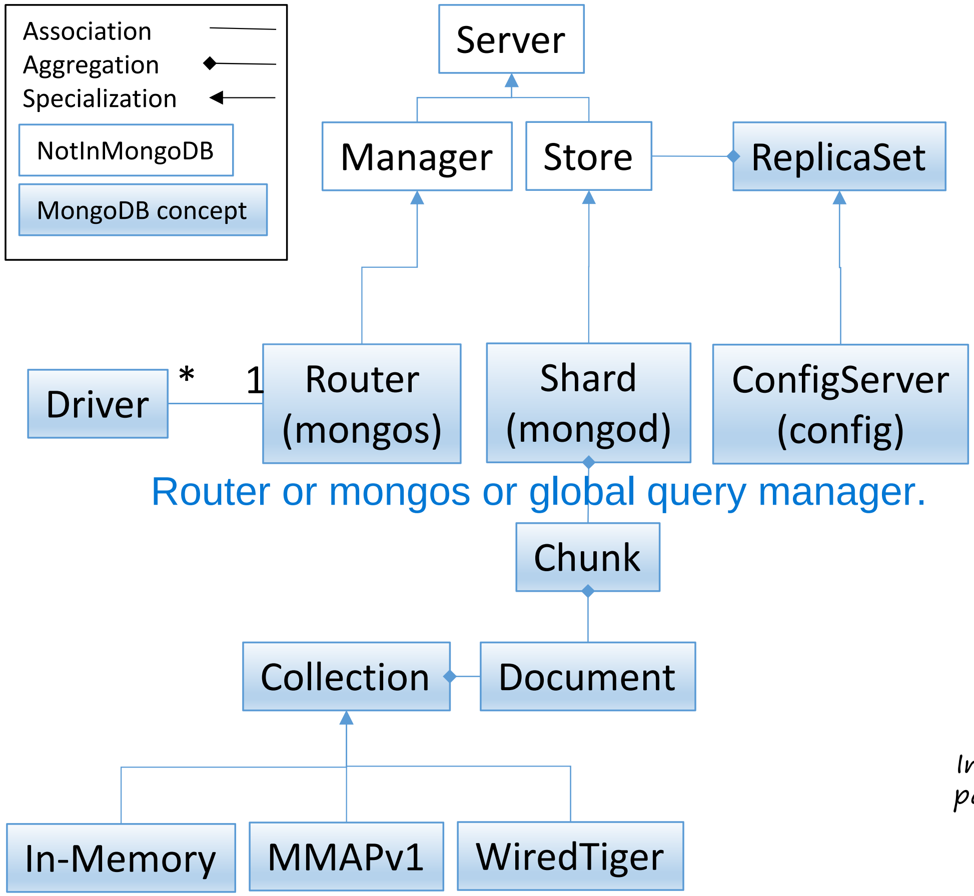
```
db.restaurants.find({"name": "x"}).count()
```

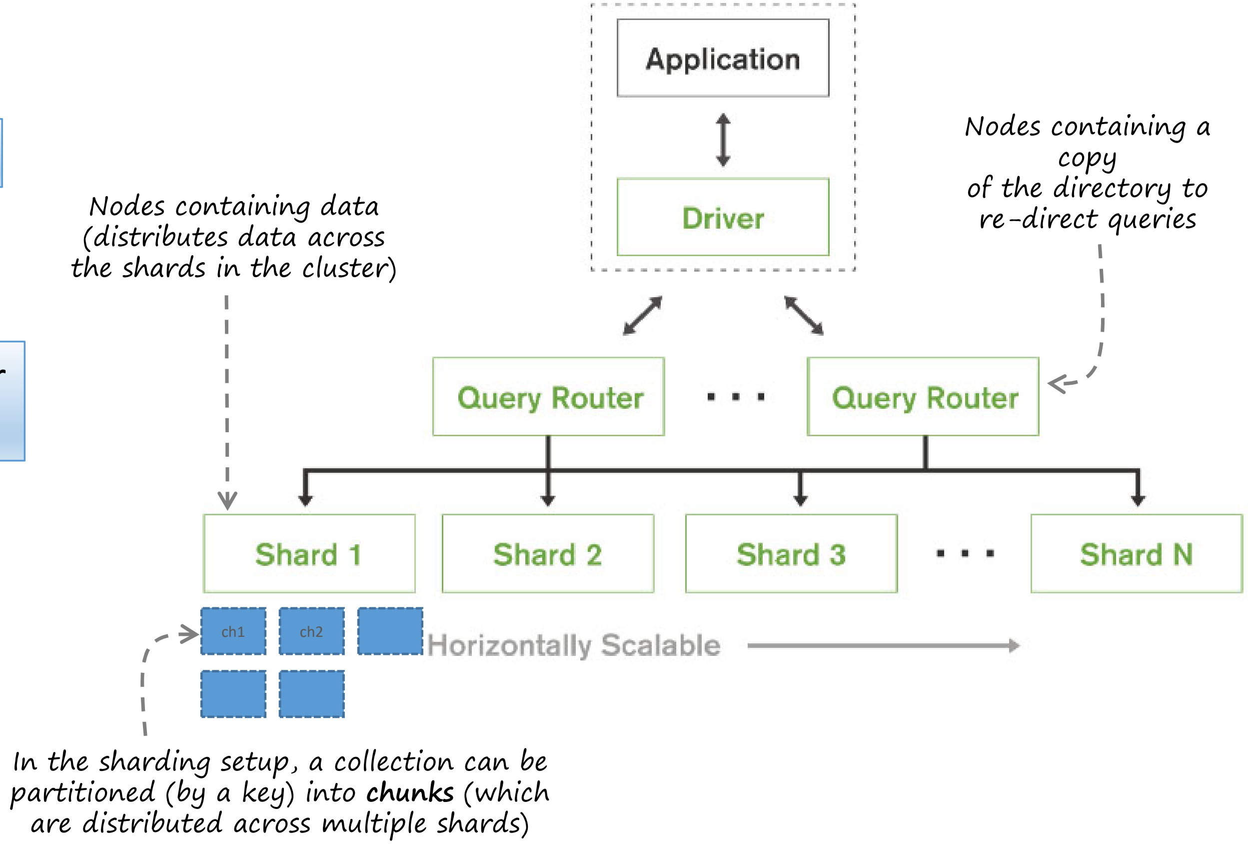
- Database methods: createCollection, copyDatabase, ...

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

- …

MongoDB functional components





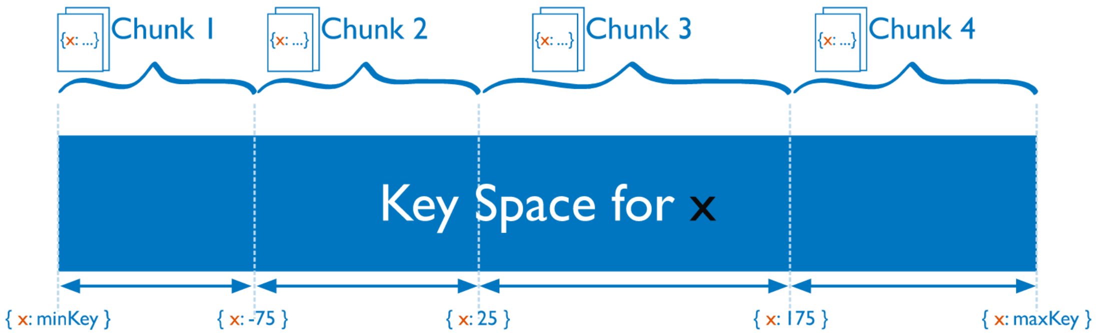
https://docs.mongodb.com/manual/core/sharded-cluster-components

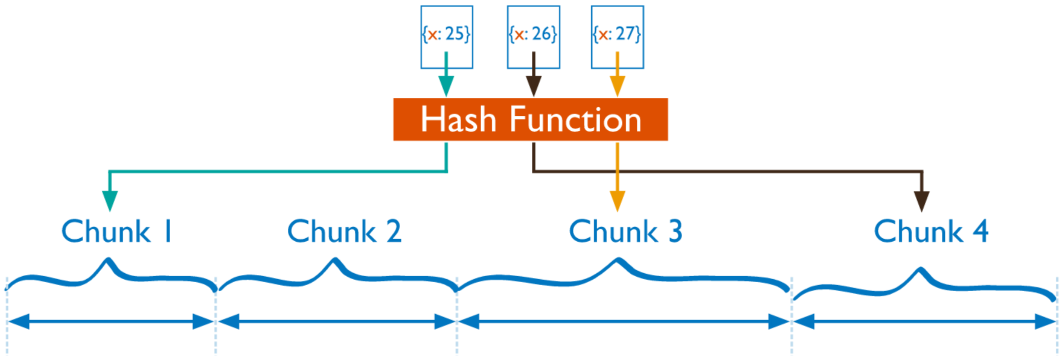
Data Design

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

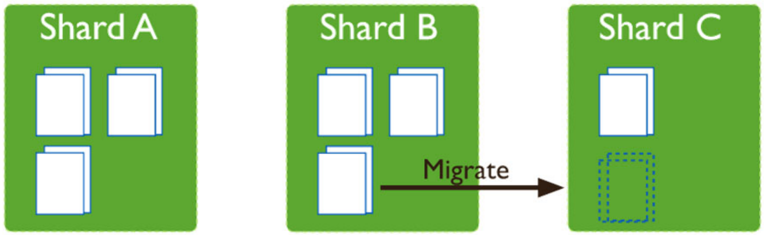
Splitting and migrating chunks

- Inserts and updates above a threshold trigger splits
  - Not in single-key chunks

Splitting is not possible when all documents have same key. They should have different keys.

- 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 in the config servers, which enables the new chunk
  5. Chunk at origin is dropped
  6. Client cache in query routers is inconsistent
     - Eventually synchronized

Change config in config server



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

the key is a mandatory attribute in all the documents of the collection, and must be indexed. It can be chosen by calling sh.shardCollection(<namespace>, <key>)

Catalog Management

Challenge II

Catalog structure

- Content Content of catalog is list of chunks in every shard
  - List of chunks in every shard
- Implemented in a replica set (as any other data) This information of correspondence between chunk and shard is stored like any other data
- Client cache in the query routers To avoid bottleneck, client cache is maintained.
  - Lazy/Primary-copy replication maintenance

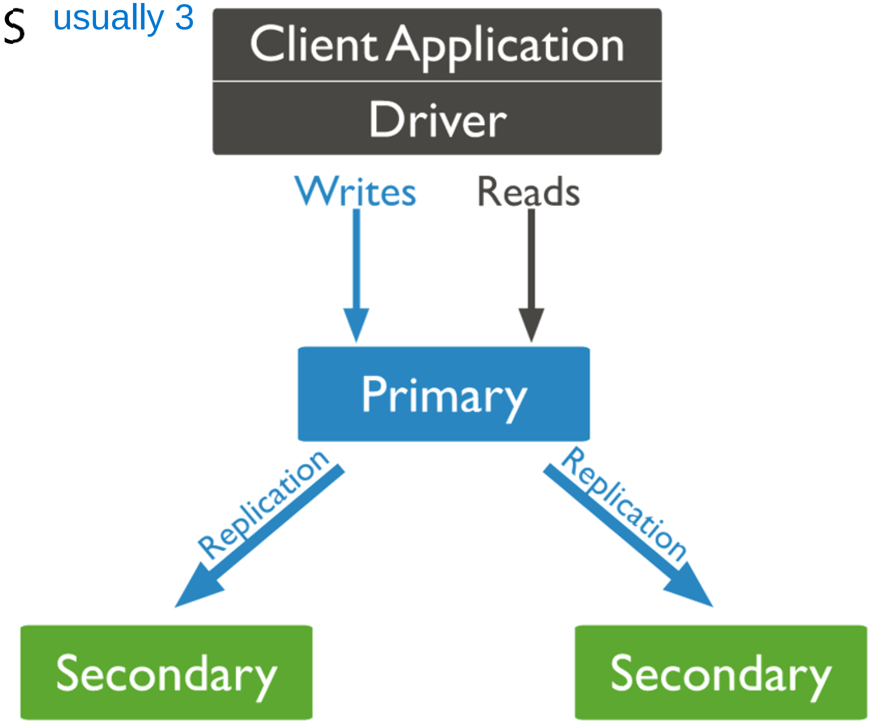
https://docs.mongodb.com/manual/core/sharded-cluster-config-servers

Transaction Management

Challenge III

Replica sets

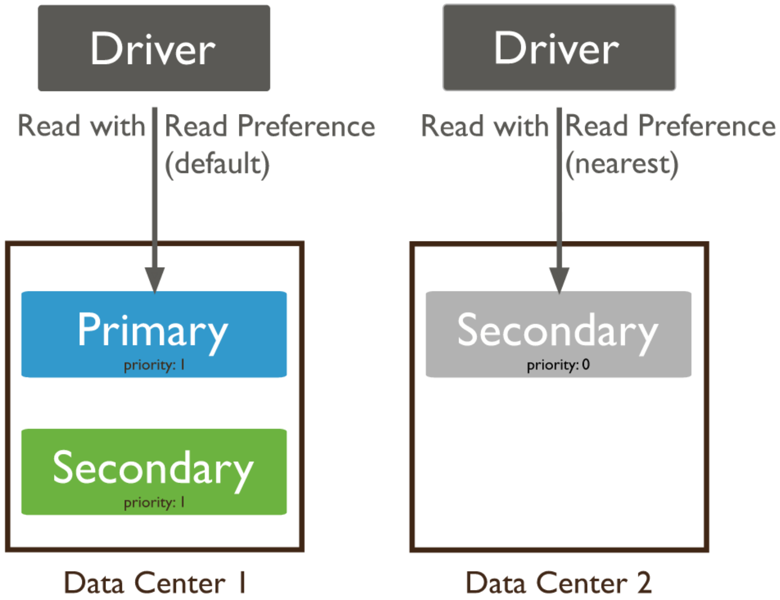
- A replica set is a set of *mongod* instances usually 3
- Primary copy with lazy replication
  - One primary copy
    - Inserts, writes, updates
    - Reads
  - Secondary copies
    - Reads



MongoDB

Read preference

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



MongoDB

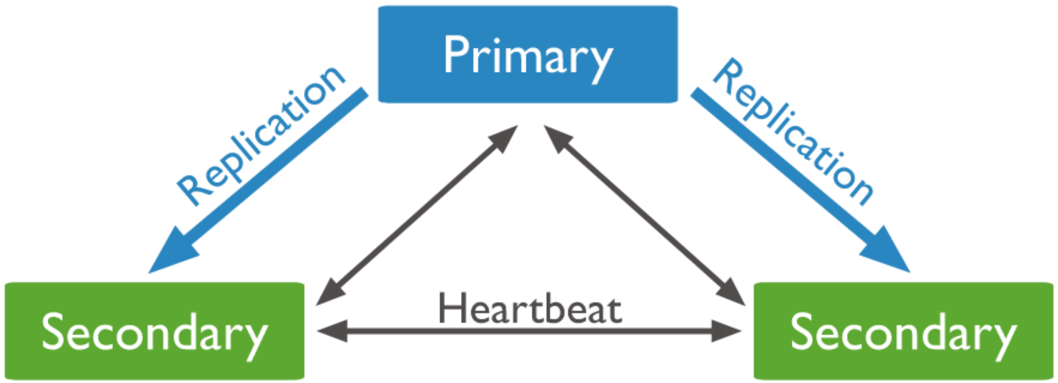
Required read and writes

- 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 Data could be lost

Smaller the value, faster will be reads and writes
Larger the value, more safer will it be

Handling failures

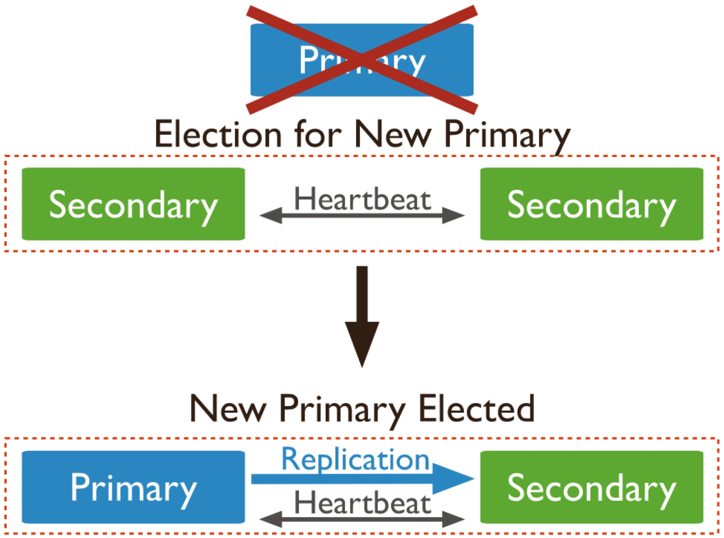
- Heartbeat system
  - Primary does not communicate with the other members for 10sec → Failure



MongoDB

Handling failures

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



MongoDB

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 stage operations

c) MapReduce   not recommended

Aggregation Framework Syntax

Pipeline stages: ($match, $group, $addfields, $sort, $unwind …)

The name of the computed field

Name collection and aggregate is the method

```
db.orders.aggregate(
  {$match: {status:"A"}},
  {$group: {_id: "$cust_id", total:{$sum: "$amount"}}}
)
```

Different stages in pipe

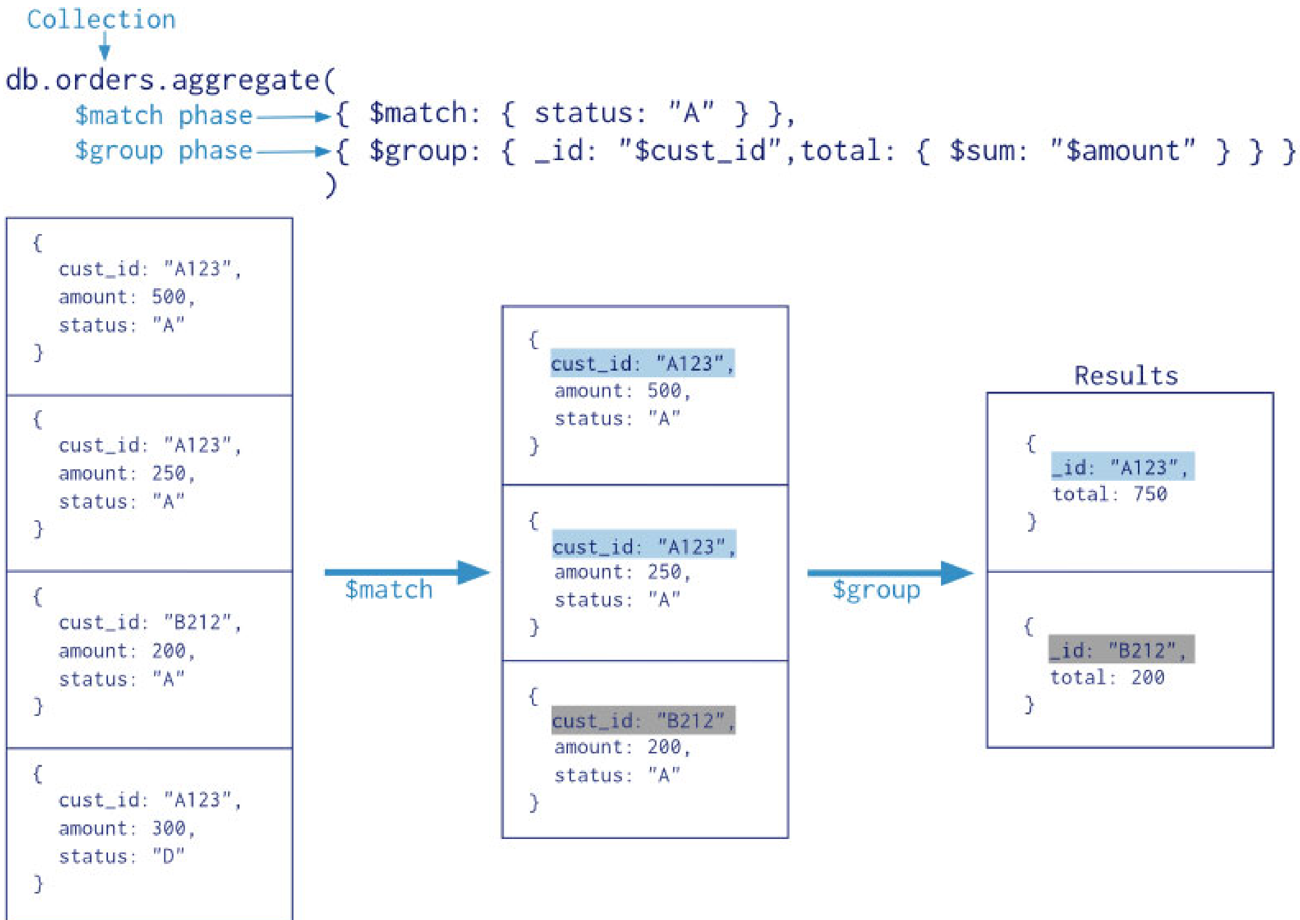
Required field: to identify the field for the group by

References the field

Pipeline operators: $sum, $max, $min …

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

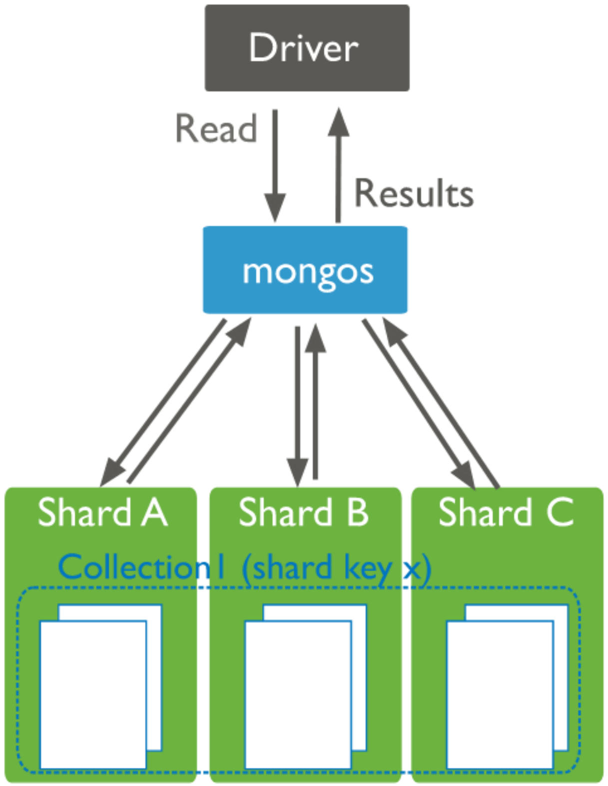
Aggregation Framework Steps



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

Query routing

Router or mongos or global query manager. The router's role is to divide the query among the different shards (data partitions) and then merge the results.



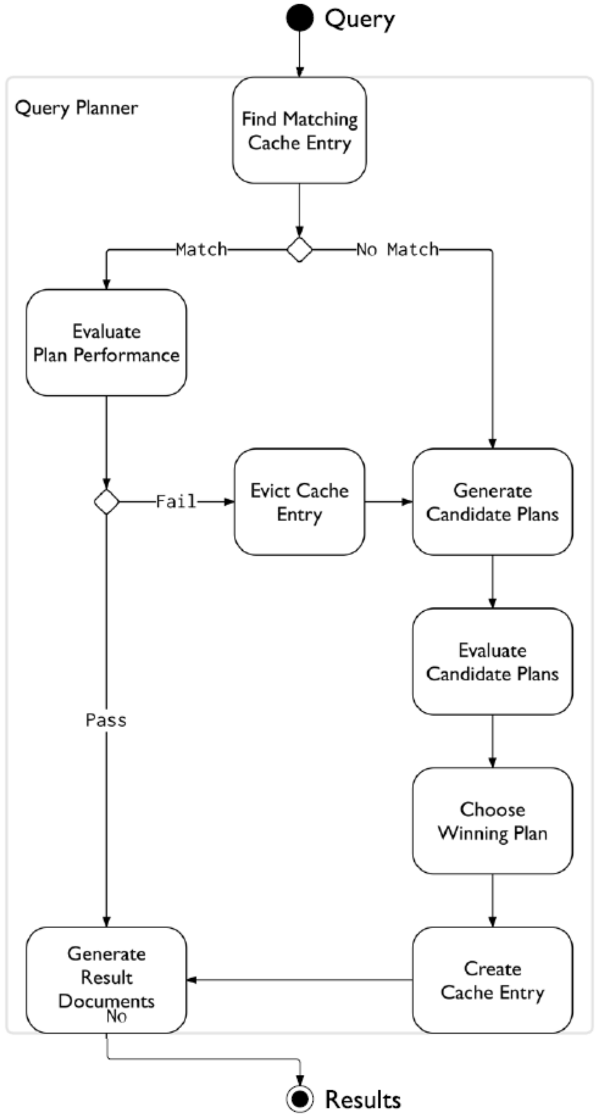
https://docs.mongodb.com/manual/core/sharded-cluster-query-router

Indexing

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

Optimizer is not cost based

To choose best execution plan, it checks which query produces more number of documents in allocated amount of time



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
- M. Hewasinghage et al. On the Performance Impact of Using JSON, Beyond Impedance Mismatch. ADBIS 2020
- A. Hogan: *Procesado de Datos Masivos*. U. de Chile.

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