

NDBI040

Big Data Management and NoSQL Databases

Lecture 8. Document stores

Doc. RNDr. Irena Holubova, Ph.D.

holubova@ksi.mff.cuni.cz

<http://www.ksi.mff.cuni.cz/~holubova/NDBI040/>



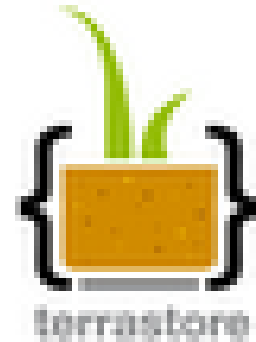
Document Databases

Basic Characteristics

- Documents are the main concept
 - Stored and retrieved
 - XML, JSON, ...
- Documents are
 - Self-describing
 - Hierarchical tree data structures
 - Can consist of maps, collections, scalar values, nested documents, ...
- Documents in a collection are expected to be **similar**
 - Their schema can differ
- Document databases store documents in the value part of the key-value store
 - Key-value stores where the value is **examinable**

Document Databases

Representatives



Lotus Notes
Storage Facility



Document Databases

Suitable Use Cases

Event Logging

- Many different applications want to log events
 - Type of data being captured keeps changing
- Events can be sharded by the name of the application or type of event

Content Management Systems, Blogging Platforms

- Managing user comments, user registrations, profiles, web-facing documents, ...

Web Analytics or Real-Time Analytics

- Parts of the document can be updated
- New metrics can be easily added without schema changes

E-Commerce Applications

- Flexible schema for products and orders
- Evolving data models without expensive data migration



Document Databases

When Not to Use

Complex Transactions Spanning Different Operations

- Atomic cross-document operations
 - Some document databases do support (e.g., RavenDB)

Queries against Varying Aggregate Structure

- Design of aggregate is constantly changing → we need to save the aggregates at the lowest level of granularity
 - i.e., to normalize the data

MongoDB



- Initial release: 2009
- Written in C++
 - Open-source
- Cross-platform
- JSON documents
 - Dynamic schemas
- Features:
 - High performance – indexes
 - High availability – replication + eventual consistency + automatic failover
 - Automatic scaling – automatic sharding across the cluster
 - MapReduce support

MongoDB

Terminology

```
{
  na
  ag
  st
  gr
}

{
  na
  ag
  st
  gr
}

{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

Collection

Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

Terminology in Oracle and MongoDB

- Each MongoDB instance has multiple **databases**
- Each database can have multiple **collections**
- When we store a document, we have to choose database and collection

MongoDB

Documents

- Use JSON
- Stored as BSON
 - Binary representation of JSON
- Have maximum size: 16MB (in BSON)
 - Not to use too much RAM
 - GridFS tool divides larger files into fragments
- Restrictions on field names:
 - `_id` is reserved for use as a primary key
 - Unique in the collection
 - Immutable
 - Any type other than an array
 - The field names cannot start with the `$` character
 - Reserved for operators
 - The field names cannot contain the `.` character
 - Reserved for accessing fields



MongoDB

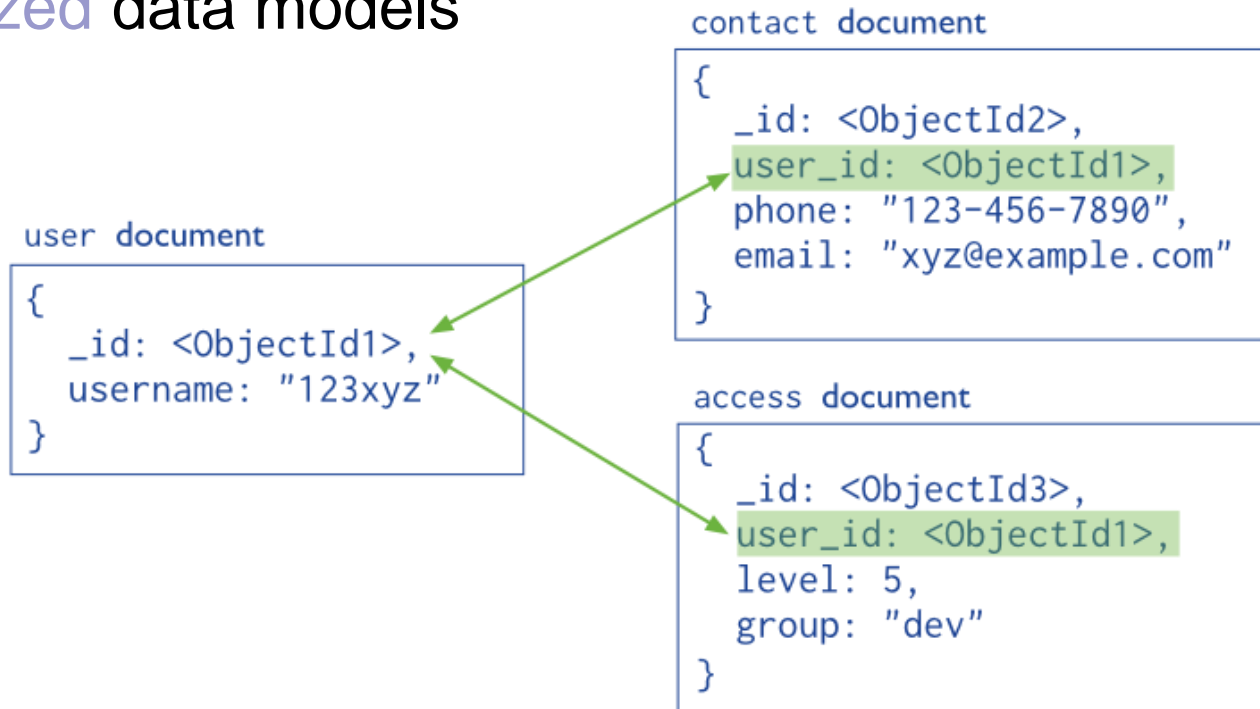
Data Model

- Documents have flexible schema
 - Collections do not enforce structure of data
 - In practice the documents are similar
- Challenge: Balancing
 - the needs of the application
 - the performance characteristics of database engine
 - the data retrieval patterns
- Key decision: references vs. embedded documents
 - Structure of data
 - Relationships between data

MongoDB

Data Model – References

- Including links / references from one document to another
- Normalized data models





MongoDB

Data Model – References

- References provides more flexibility than embedding
- Use normalized data models:
 - When embedding would result in duplication of data not outweighed by read performance
 - To represent more complex many-to-many relationships
 - To model large hierarchical data sets
- Disadvantages:
 - Can require more roundtrips to the server (follow up queries)

MongoDB

Data Model – Embedded Data

- Related data in a single document structure
 - Documents can have subdocuments (in a field of array)
 - Applications may need to issue less queries
- Denormalized data models
- Allow applications to retrieve and manipulate related data in a single database operation

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

Embedded sub-document

Embedded sub-document

MongoDB

Data Model – Embedded Data

- Use embedded data models when:
 - When we have “contains” relationships between entities
 - One-to-one relationships
 - In one-to-many relationships, where child documents always appear with one parent document
- Provides:
 - Better performance for read operations
 - Ability to retrieve/update related data in a single database operation
- Disadvantages:
 - Documents may significantly grow after creation
 - Impacts write performance
 - The document must be **relocated** on disk if the size exceeds allocated space
 - May lead to data **fragmentation**



MongoDB

Data Modification

- Operations: create, update, delete
 - Modify the data of a single collection of documents
- For update / delete: criteria to select the documents to update / remove

Collection
↓
db.users.insert(
Document
↓
{
 name: "sue",
 age: 26,
 status: "A",
 groups: ["news", "sports"]
}

Document

```
{  
  name: "sue",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```

insert →

Collection

{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }

users

MongoDB

Data Insertion

```
db.inventory.insert( { _id: 10, type: "misc", item:
    "card", qty: 15 } )
```

- Inserts a document with three fields into collection `inventory`
 - User-specified `_id` field

```
db.inventory.update (
    { type: "book", item : "journal" },
    { $set : { qty: 10 } },
    { upsert : true }
)
```

- Creates a new document if no document in the `inventory` collection contains `{ type: "books", item : "journal" }`
 - MongoDB adds the `_id` field and assigns as its value a unique `ObjectId`
 - The result contains fields `type`, `item`, `qty` with the specified values

MongoDB

Data Insertion and Removal

```
db.inventory.save( { type: "book", item:  
  "notebook", qty: 40 } )
```

- Creates a new document in collection `inventory` if `_id` is not specified or does not exist in the collection

```
db.inventory.remove( { type : "food" } )
```

- Removes all documents that have `type` equal to `food` from the `inventory` collection

```
db.inventory.remove( { type : "food" }, 1 )
```

- Removes one document that have `type` equal to `food` from the `inventory` collection

MongoDB

Data Updates

```
db.inventory.update (
    { type : "book" },
    { $inc : { qty : -1 } },
    { multi: true }
)
```

- Finds all documents with `type` equal to `book` and modifies their `qty` field by -1

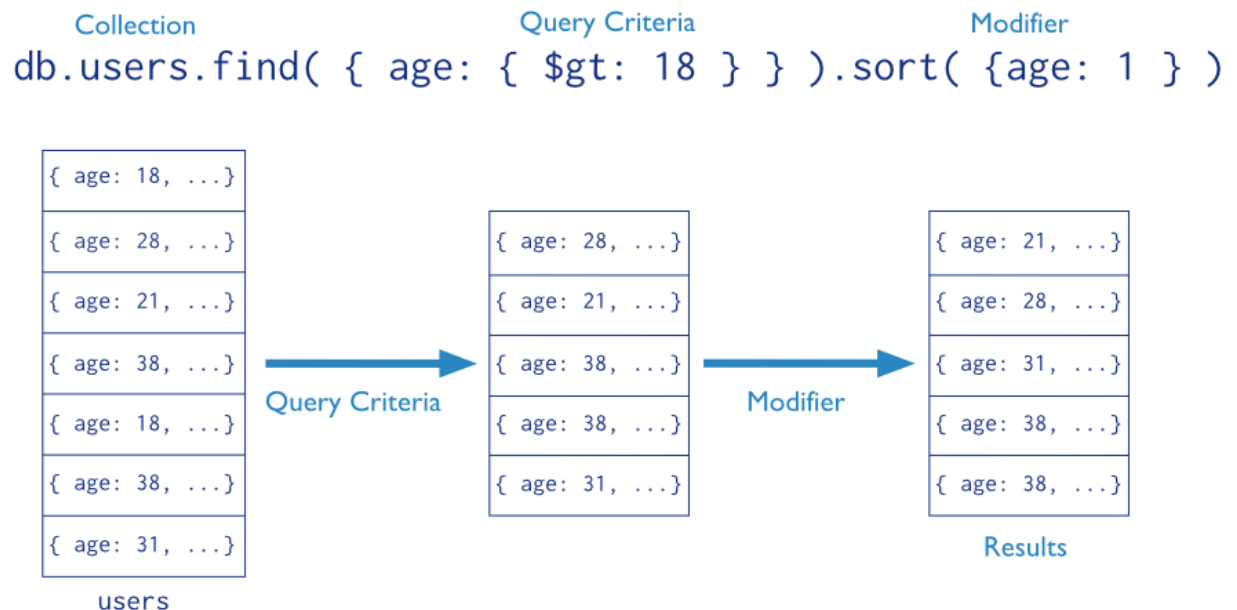
```
db.inventory.save (
    {
        _id: 10,
        type: "misc",
        item: "placard"
    } )
```

- Replaces document with `_id` equal to 10

MongoDB

Query

- Targets a specific collection of documents
- Specifies criteria that identify the returned documents
- May include a **projection** that specifies the fields from the matching documents to return
- May impose limits, sort orders, ...



MongoDB

Query – Basic Queries, Logical Operators

```
db.inventory.find( {} )
```

```
db.inventory.find()
```

- All documents in the collection

```
db.inventory.find( { type: "snacks" } )
```

- All documents where the `type` field has the value `snacks`

```
db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
```

- All documents where value of the `type` field is either `food` or `snacks`

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

- All documents where the `type` field has the value `food` **and** the value of the `price` field is less than `9.95`

MongoDB

Query – Logical Operators

```
db.inventory.find(  
    { $or: [  
        { qty: { $gt: 100 } },  
        { price: { $lt: 9.95 } }  
    ] } )
```

- All documents where the field `qty` has a value greater than (`$gt`) 100 **or** the value of the `price` field is less than (`$lt`) 9.95

```
db.inventory.find( { type: 'food', $or: [  
    { qty: { $gt: 100 } },  
    { price: { $lt: 9.95 } } ]  
} )
```

- All documents where the value of the `type` field is `food` **and** either the `qty` has a value greater than (`$gt`) 100 **or** the value of the `price` field is less than (`$lt`) 9.95

MongoDB

Query – Subdocuments

```
db.inventory.find( {  
    producer: {  
        company: 'ABC123',  
        address: '123 Street'  
    }  
} )
```

- All documents where the value of the field `producer` is a subdocument that contains only the field `company` with the value `ABC123` and the field `address` with the value `123 Street`, in the exact order

```
db.inventory.find( { 'producer.company': 'ABC123' } )
```

- All documents where the value of the field `producer` is a subdocument that contains a field `company` with the value `ABC123` and may contain other fields



dot notation

The diagram consists of a rectangular box at the bottom containing the text 'dot notation'. From the top-left corner of this box, a line extends upwards and to the left, ending with an arrowhead pointing to the 'producer' field in the second query. From the top-right corner of the box, a line extends upwards and to the right, ending with an arrowhead pointing to the 'company' field in the same query. A third line extends vertically upwards from the top-center of the box, ending with an arrowhead pointing to the 'may contain other fields' text in the list item below.

MongoDB

Query – Arrays



exact match

```
db.inventory.find( { tags: [ 'fruit', 'food',  
    'citrus' ] } )
```

- All documents where the value of the field `tags` is an array that holds exactly three elements, `fruit`, `food`, and `citrus`, in this order

```
db.inventory.find( { tags: 'fruit' } )
```

- All documents where value of the field `tags` is an array that contains `fruit` as one of its elements

```
db.inventory.find( { 'tags.0' : 'fruit' } )
```

- All documents where the value of the `tags` field is an array whose first element equals `fruit`

MongoDB

Query – Arrays of Subdocuments

```
db.inventory.find( { 'memos.0.by': 'shipping' } )
```

- All documents where the `memos` field contains an array whose first element is a subdocument with the field `by` with the value `shipping`

```
db.inventory.find( { 'memos.by': 'shipping' } )
```

- All documents where the `memos` field contains an array that contains at least one subdocument with the field `by` with the value `shipping`

```
db.inventory.find({  
    'memos.memo': 'on time',  
    'memos.by': 'shipping'  
})
```

- All documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to `on time` and the field `by` equal to `shipping`

MongoDB

Query – Limit Fields of the Result

or true

```
db.inventory.find( { type: 'food' }, { item: 1, qty: 1 } )
```

- Only the `item` and `qty` fields (and by default the `_id` field) return in the matching documents

```
db.inventory.find( { type: 'food' }, { item: 1, qty: 1, _id: 0 } )
```

- Only the `item` and `qty` fields return in the matching documents

```
db.inventory.find( { type: 'food' }, { type : 0 } )
```

- The `type` field does not return in the matching documents

or false

- Note: With the exception of the `_id` field we cannot combine inclusion and exclusion statements in projection documents.

MongoDB

Query – Sorting

```
db.collection.find().sort( { age: -1 } )
```

- Returns all documents in `collection` sorted by the `age` field in descending order

```
db.bios.find().sort( { 'name.last': 1,  
  'name.first': 1 } )
```

- Specifies the sort order using the fields from a sub-document `name`
- Sorts first by the `last` field and then by the `first` field in ascending order

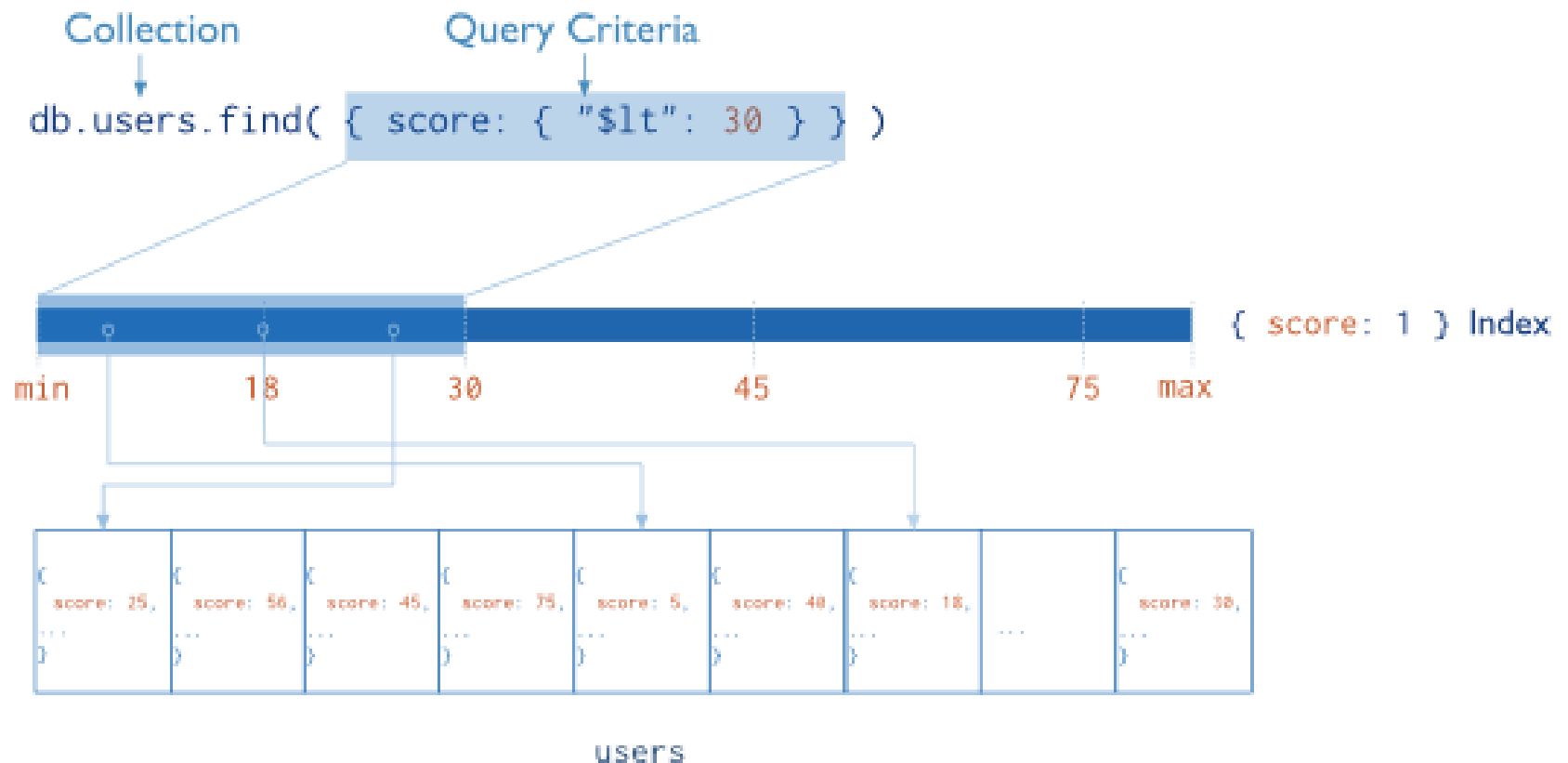
MongoDB

Indexes

- Without indexes:
 - MongoDB must scan every document in a collection to select those documents that match the query statement
- Indexes store a portion of the collection's data set in an easy to traverse form
 - Stores the value of a specific field or set of fields ordered by the value of the field
 - B-tree like structures
- Defined at collection level
- Purpose:
 - To speed up common queries
 - To optimize the performance of other operations in specific situations

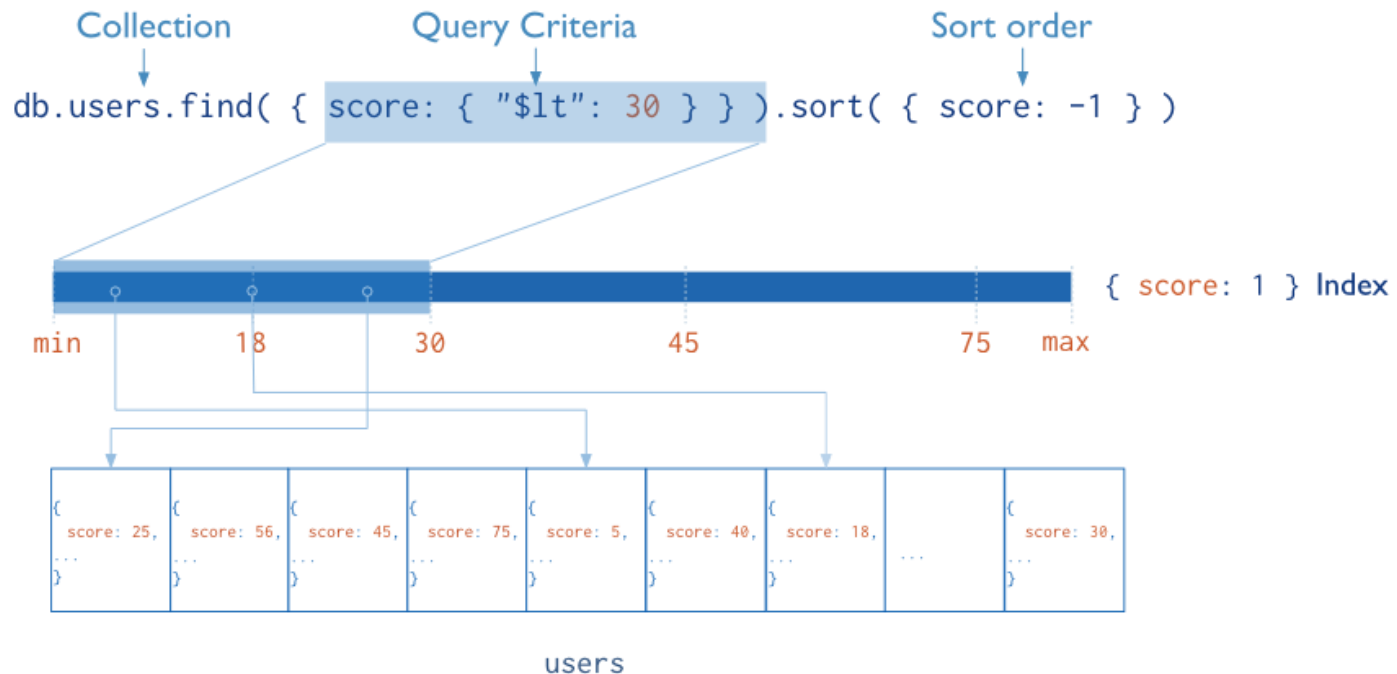
MongoDB

Indexes – Example



MongoDB

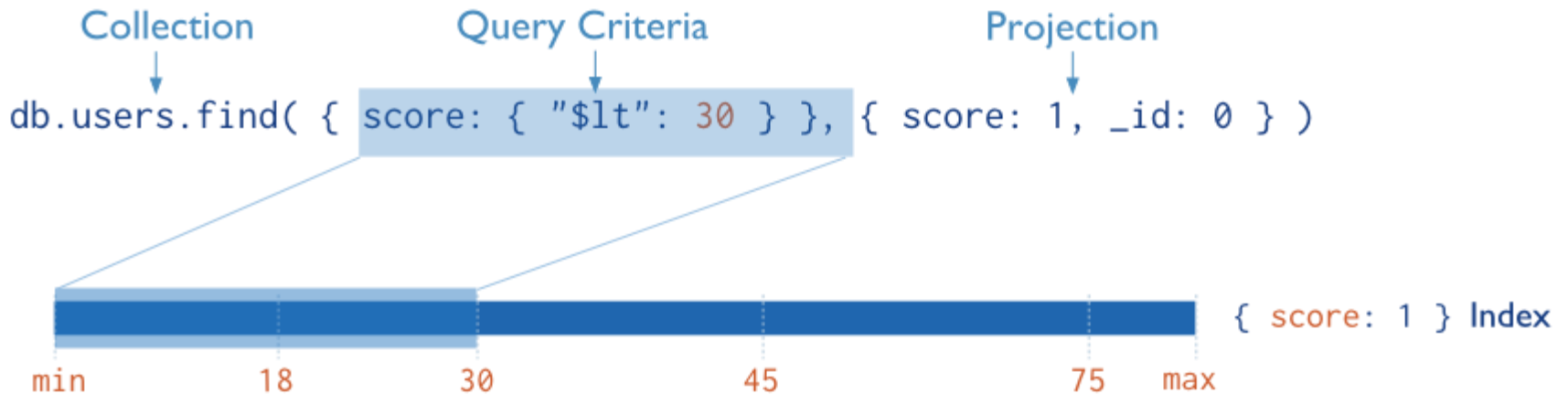
Indexes – Usage for Sorted Results



- The index stores `score` values in ascending order
- MongoDB can traverse the index in either ascending or descending order to return sorted results (without sorting)

MongoDB

Indexes – Usage for Covered Results



- MongoDB does not need to inspect data outside of the index to fulfil the query

MongoDB

Index Types

■ **Default `_id`**

- Exists by default
 - If applications do not specify `_id`, it is created automatically
- Unique by default

■ **Single Field**

- User-defined indexes on a single field of a document

■ **Compound**

- User-defined indexes on multiple fields

■ **Multikey index**

- To index the content stored in arrays
- Creates separate index entry for every element of the array

collection



Single field index on the `score` field (ascending).

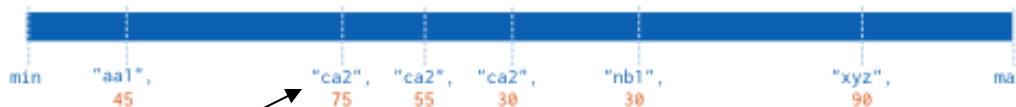


`{ score: 1 } Index`

collection



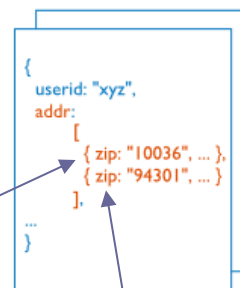
Compound index on the `userid` field (ascending) and the `score` field (descending).



`{ userid: 1, score: -1 } Index`

sorts first by `userid` and then, within each `userid` value, sort by `score`

collection



Multikey index on the `addr.zip` field



`{ "addr.zip": 1 } Index`

MongoDB

Index Types

■ Geospatial Field

- 2d indexes = use planar geometry when returning results
 - For data representing points on a two-dimensional plane
- 2sphere indexes = use spherical (Earth-like) geometry to return results
 - For data representing longitude, latitude

■ Text Indexes

- Searching for string content in a collection

■ Hash Indexes

- Indexes the hash of the value of a field
- Only support equality matches (not range queries)

MongoDB

Indexes

```
db.people.ensureIndex( { "phone-number": 1 } )
```

- Creates a single-field index on the phone-number field of the people collection

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

- Creates a compound index on the item, category, and price fields

```
db.accounts.ensureIndex( { "tax-id": 1 }, { unique: true } )
```

- Creates a unique index
 - Prevents applications from inserting documents that have duplicate values for the inserted fields

```
db.collection.ensureIndex( { _id: "hashed" } )
```

- Creates a hashed index on _id

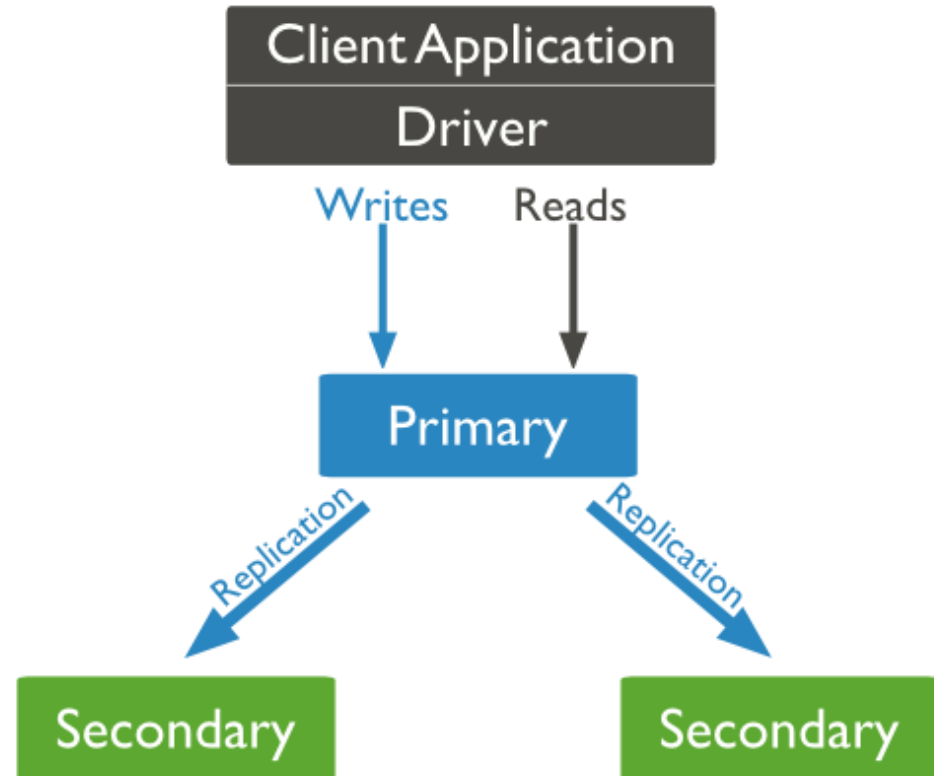


More on
Internals

MongoDB

Replication

- Master/slave replication
- **Replica set** = group of instances that host the same data set
 - **primary** (master) – receives all write operations
 - **secondaries** (slaves) – apply operations from the primary so that they have the same data set



MongoDB

Replication Steps

■ Write:

1. MongoDB applies write operations on the primary
2. MongoDB records the operations to the primary's **oplog**
3. Secondary members replicate oplog + apply the operations to their data sets

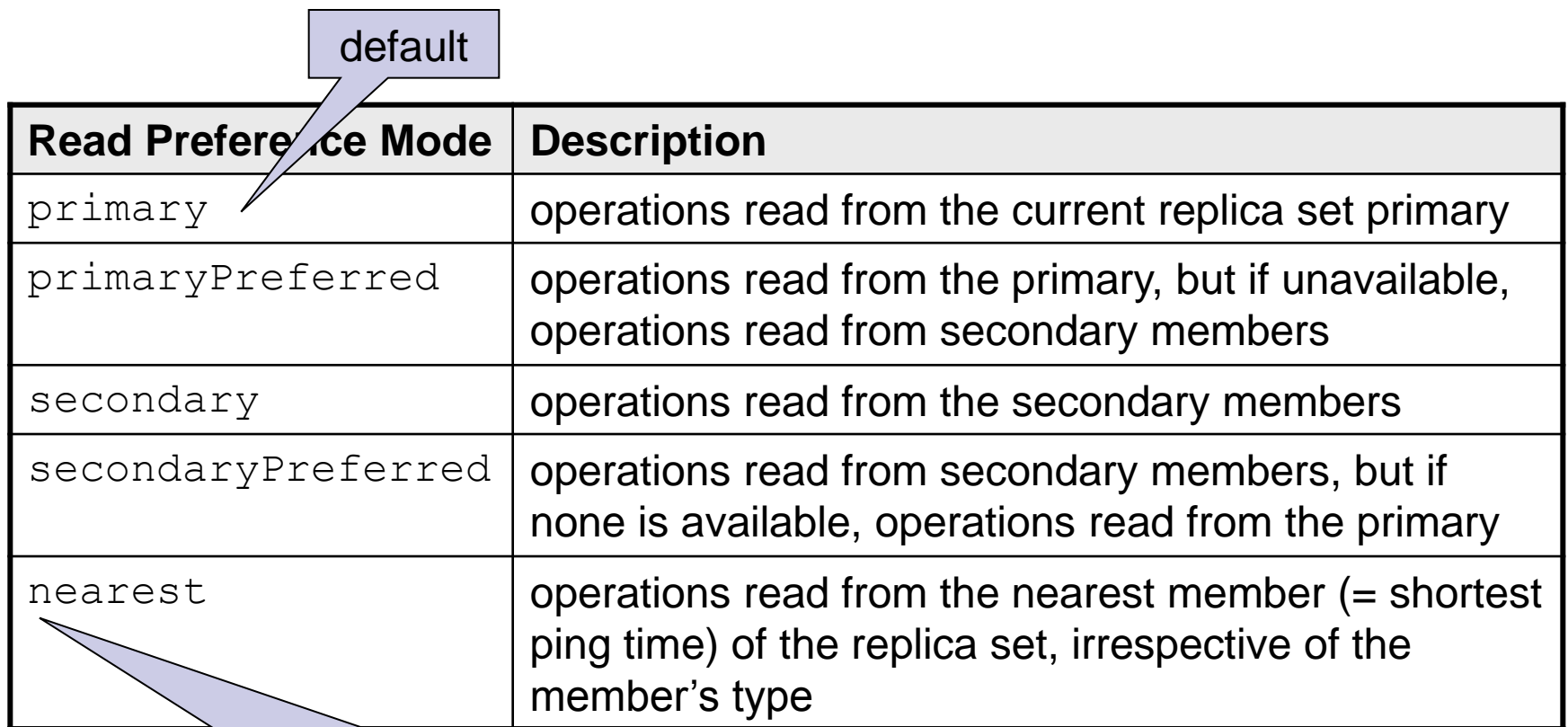
operation log

■ Read: All members of the replica set can accept read operations

- By default, an application directs its read operations to the primary member
 - Guaranties the latest version of a document
 - Decreases read throughput
- Read preference mode can be set

MongoDB

Replication – Read Preference Mode



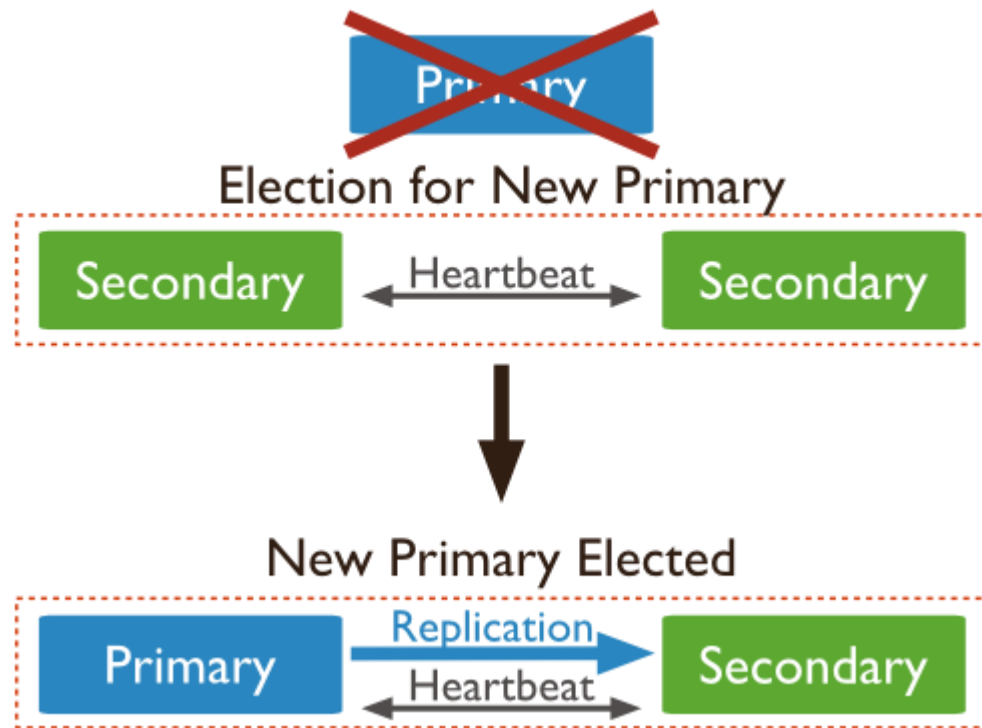
Read Preference Mode	Description
primary	operations read from the current replica set primary
primaryPreferred	operations read from the primary, but if unavailable, operations read from secondary members
secondary	operations read from the secondary members
secondaryPreferred	operations read from secondary members, but if none is available, operations read from the primary
nearest	operations read from the nearest member (= shortest ping time) of the replica set, irrespective of the member's type

minimize the effect of network latency

MongoDB

Replica Set Elections

- Replica set can have at most one primary
- If the current primary becomes unavailable, an **election** determines a new primary
- Note:
 - Elections need some time
 - No primary \Rightarrow no writes



MongoDB

Replica Set Elections – Influencing Factors

■ Heartbeat (ping)

- Every 2s sent to each other
- No response for 10s \Rightarrow node is inaccessible

■ Priority comparisons

- Higher priority = preferred to be voted
- Members with priority = 0
 - Cannot become primary (not eligible)
 - Cannot trigger election, but can vote
- The current primary has the highest priority and is within 10s of the latest oplog entry \Rightarrow OK
- A higher-priority member catches up to within 10s of the latest oplog entry of the current primary \Rightarrow elections
 - The higher-priority node has a chance to become primary

■ Connections

- A node cannot become primary unless it can connect to a majority of the members

MongoDB

Replica Set Elections – Mechanism

- Replica sets hold an election any time there is no primary:
 - Initiation of a new replica set
 - A secondary loses contact with a primary
 - A primary **steps down**
- A primary will step down:
 - After receiving the `replSetStepDown` command
 - Forces a primary to become a secondary
 - If one of the current secondaries is eligible for election and has a higher priority
 - If it cannot contact a majority of the members of the replica set

MongoDB

Replica Set Elections – Mechanism

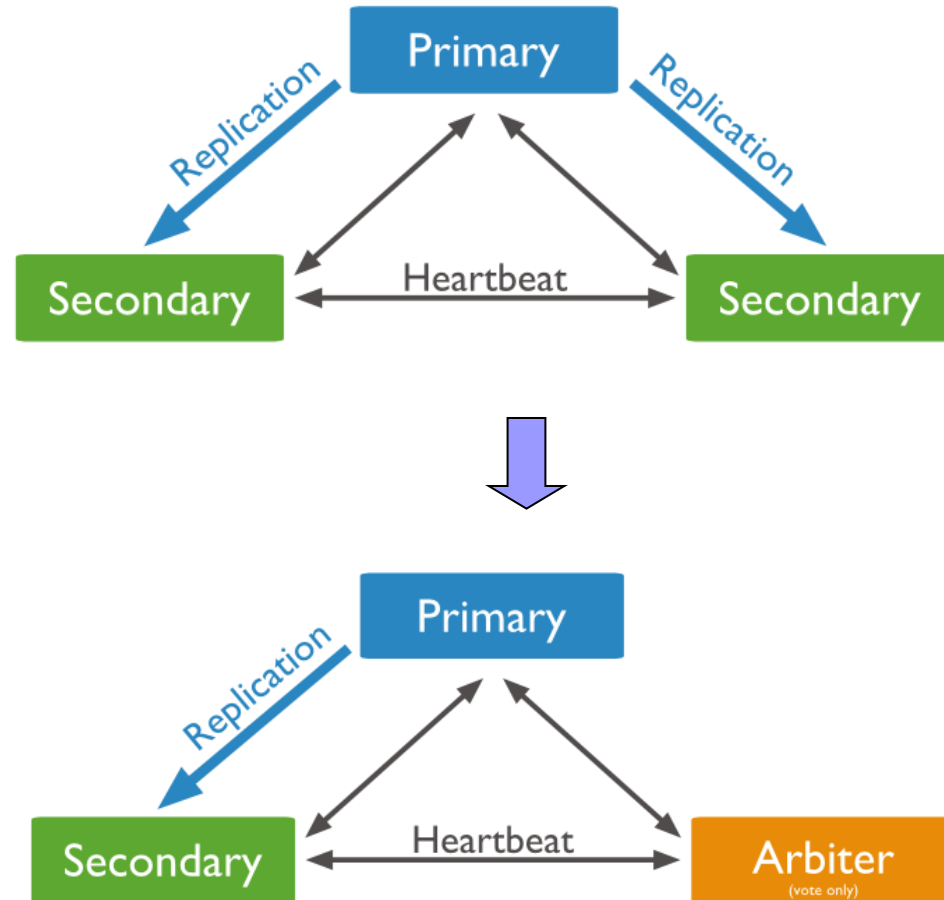
- The replica set elects an eligible member with the highest priority value as primary
 - By default, all members have a priority of 1
 - Can be adjusted
- The first member to receive the majority of votes becomes primary
 - By default, all members have 1 vote
 - Can be disabled = **non-voting members**
 - Hold copies of data
 - Can become primary
 - Not recommended to set more than 1 (better use priority)
- All members of a replica set can **veto** an election, e.g.,
 - If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set.
 - If the member seeking an election has a lower priority than another member in the set that is also eligible for election.
 - ...

MongoDB

Replication – Arbiters

■ Arbiter

- A special node
- Does not maintain a data set
 - Does not require dedicated hardware
- Cannot be a primary
- Exists to vote in elections
 - For replicas with even number of members



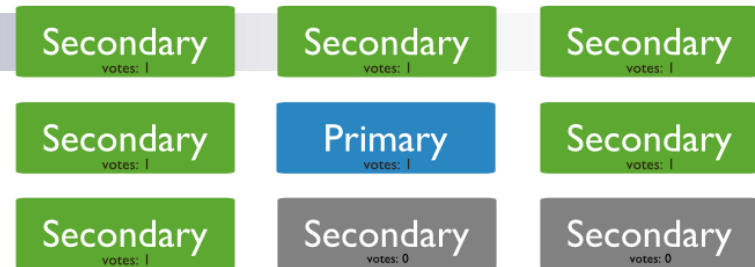
MongoDB

Replication – Secondaries

- A secondary can be configured as:
 - **Priority 0** – to prevent it from becoming a primary in an election
 - e.g., a standby
 - **Hidden** – to prevent applications from reading from it
 - Just replicates the data for special usage
 - Can vote in elections
 - **Delayed** – to keep a running “historical” snapshot
 - For recovery from errors like unintentionally deleted databases

MongoDB

Replication – a few more notes



- A replica set selects a new primary within cca 1 minute
 - No primary = no writes
- Fault tolerance = number of members that can become unavailable and still leave enough members in the set to elect a primary
 - Primary needs majority
 - Otherwise the replica set cannot have a primary = no writes
- In current version of MongoDB: only 12 members in total
 - Only 7 members can vote at a time
 - If > 12 nodes are necessary, use older master-slave technique (without automatic failover)

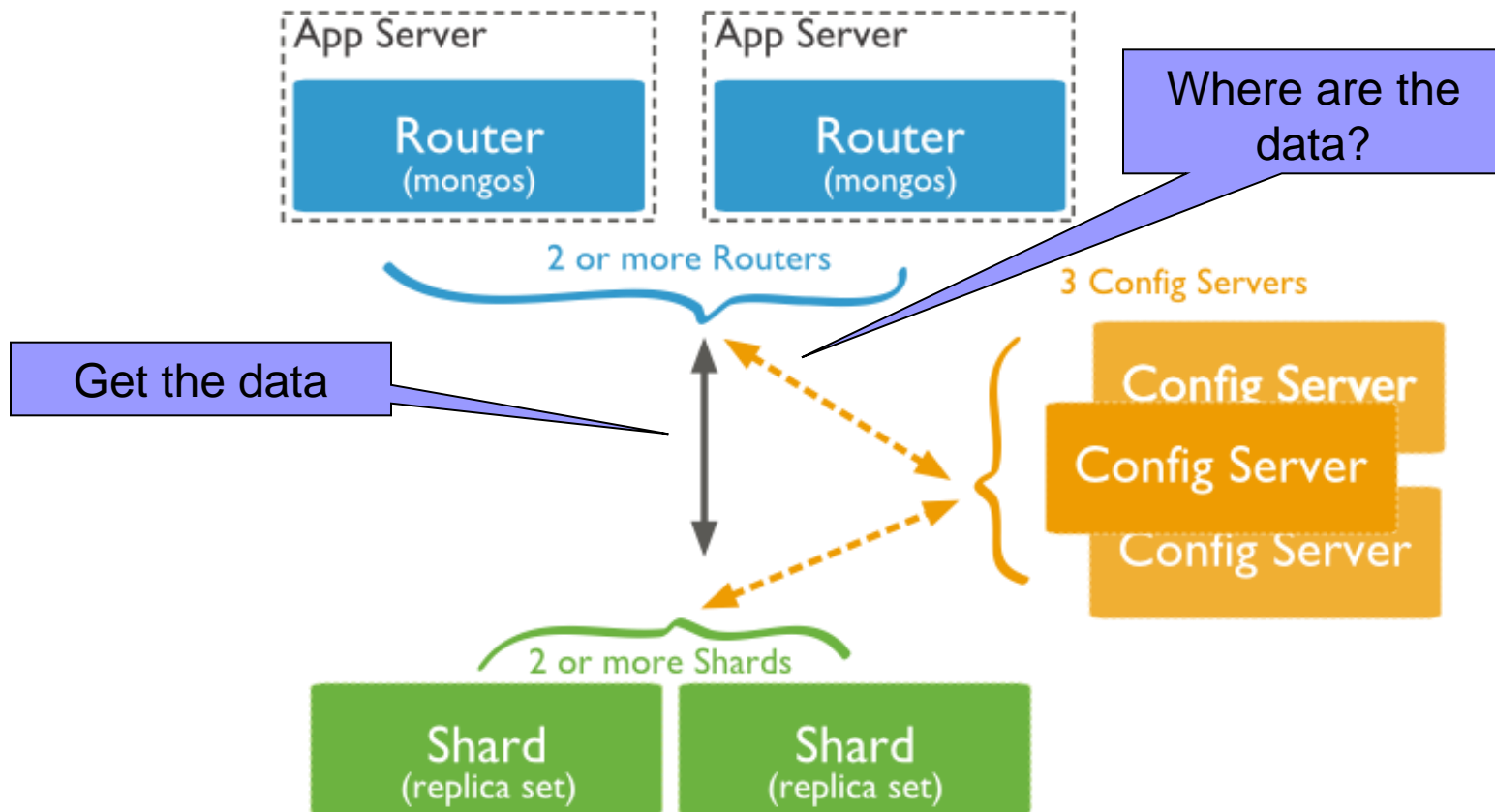
MongoDB

Sharding

- Supported through sharded clusters
- Consisting of:
 - Shards – store the data
 - Each shard is a replica set
 - For testing purposes can be a single node
 - Query routers – interface with client applications
 - Direct operations to the appropriate shard(s) + return the result to the user
 - More than one \Rightarrow to divide the client request load
 - Config servers – store the cluster's metadata
 - Mapping of the cluster's data set to the shards
 - Recommended number: 3

MongoDB

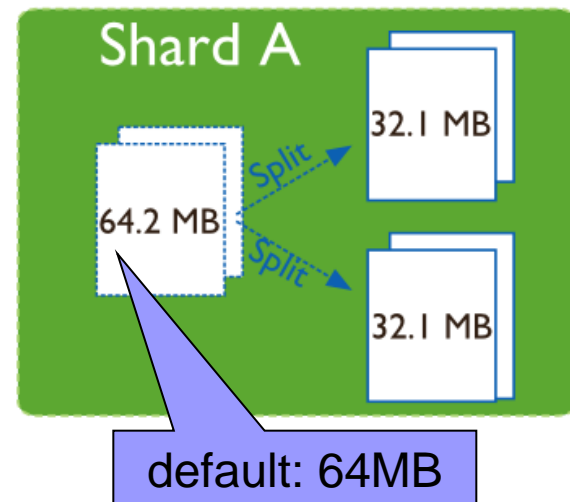
Sharded Cluster



MongoDB

Data Partitioning

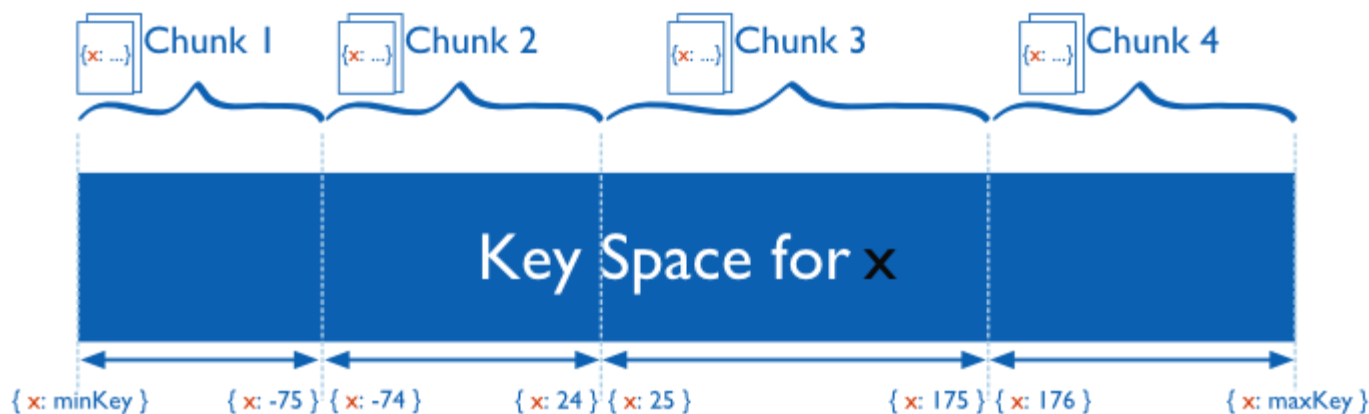
- Partitions a collection's data by the **shard key**
 - Indexed (possibly compound) field that exists in every document in the collection
 - Immutable
 - Divided into chunks distributed across shards
 - **Range-based partitioning**
 - **Hash-based partitioning**
 - When a chunk grows beyond the chunk size, it is split
 - Small chunks \Rightarrow more even distribution at the expense of more frequent migrations
 - Large chunks \Rightarrow fewer migrations



MongoDB

Range-Based Partitioning

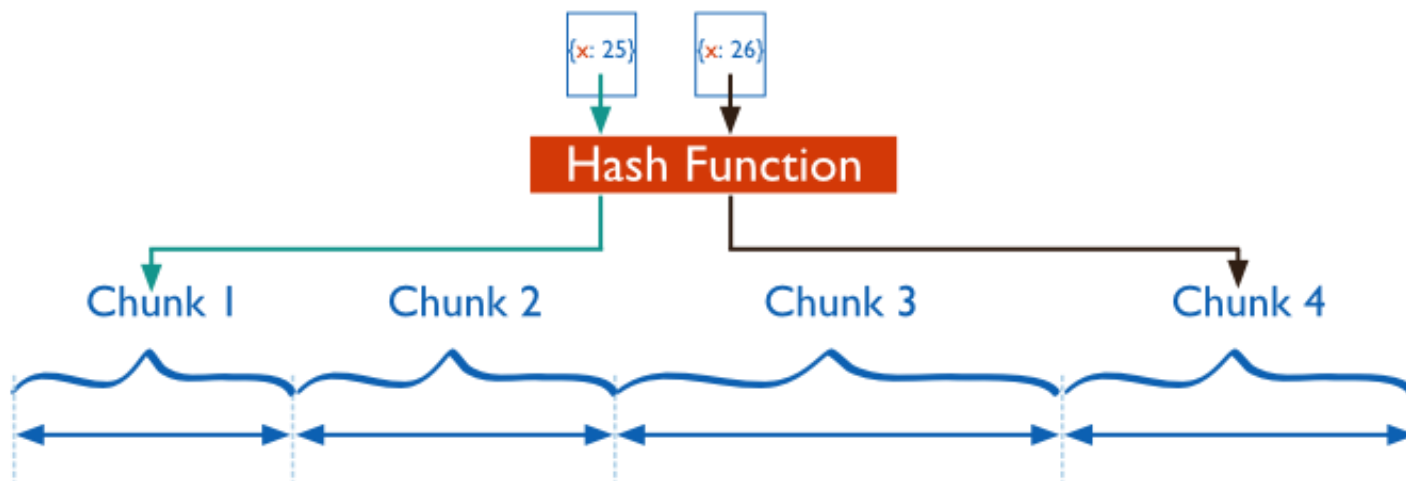
- Each value of the shard key falls at some point on line from negative infinity to positive infinity
- The line is partitioned into non-overlapping chunks
- Documents with “close” shard key values are likely to be in the same chunk
 - More efficient range queries
 - Can result in an uneven distribution of data



MongoDB

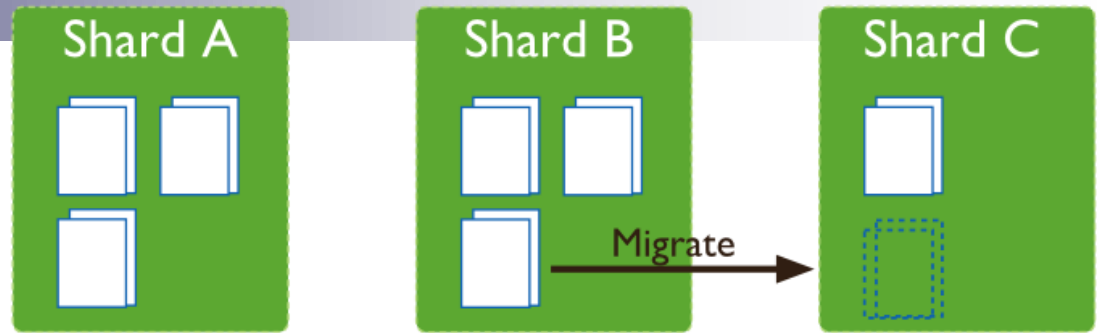
Hash-Based Partitioning

- Computes a hash of a field's value
 - Hashes form chunks
- Ensures a more random distribution of a collection in the cluster
 - Documents with “close” shard key values are unlikely to be a part of the same chunk
 - A range query may need to target most/all shards



MongoDB

Balancing



- Balancer = background process that manages chunk migrations
 - Responsible for redistributing the chunks of a sharded collection evenly among the shards for every sharded collection
 - When the distribution is uneven
 - From shard with the largest to shard with the lowest number of chunks
- Steps:
 1. During migration operations work with the original shard
 2. Destination shard captures and applies all changes made to the data during migration
 3. Destination shard updates the metadata regarding the location on config server

MongoDB

Journaling

- Journaling = MongoDB stores and applies write operations in memory and in a journal before the changes are done in the data files
 - To bring the database to a consistent state after hard shutdown
 - Can be switched on/off
- Journal directory – holds journal files
- Journal file = write-ahead redo logs
 - Append only file
 - Deleted when all the writes are performed
 - When it holds 1GB of data, MongoDB creates a new journal file
 - The size can be modified
- Clean shutdown removes all the files in the journal directory

MongoDB

Transactions

- Write operations are **atomic** at the level of a **single document**
 - Including nested documents
 - Sufficient for many cases, but not all
- When a single write operation modifies multiple documents, it is not atomic
 - Other operations may interleave
- Transactions:
 - **Isolation of a single write** operation that affects multiple documents
 - No client sees the changes until the operation completes or errors out
 - `db.foo.update({ field1 : 1 , $isolated : 1 } , { $inc : { field2 : 1 } } , { multi: true })`
 - **Two-phase commit**
 - Multi-document updates
 - Transaction-like semantics

MongoDB

Two-phase Commit – Example (part I.)

```
db.accounts.save({name: "A", balance: 1000,  
  pendingTransactions: []})
```

```
db.accounts.save({name: "B", balance: 1000,  
  pendingTransactions: []})
```

- Creating of a collection of (two) accounts (A and B)

```
db.transactions.save({source: "A", destination: "B",  
  value: 100, state: "initial"})
```

- **Step 1.** Create a transaction (having an **initial** state) and store it into collection of transactions
 - e.g., transferring money from account A to B
 - Other states of a transaction: initial, pending, applied, done, canceling, and canceled

MongoDB

Two-phase Commit – Example (part II.)

```
t = db.transactions.findOne({state: "initial"})
db.transactions.update({_id: t._id},
  { $set: {state: "pending"} })
```

■ Step 2. Set transaction state to pending

```
db.accounts.update({ name: t.source,
  pendingTransactions: {$ne: t._id} },
  { $inc: {balance: -t.value},
    $push: {pendingTransactions: t._id}})
db.accounts.update({ name: t.destination,
  pendingTransactions: {$ne: t._id} },
  { $inc: {balance: t.value},
    $push: {pendingTransactions: t._id}})
```

Condition ensuring atomic operation: If not in pending, apply and add to pending

■ Step 3. Apply transaction to both accounts + add as pending

MongoDB

Two-phase Commit – Example (part III.)

```
db.transactions.update({_id: t._id},  
  { $set: {state: "applied"} })
```

- **Step 4. Set transaction state to **applied****

```
db.accounts.update({name: t.source},  
  { $pull: {pendingTransactions: t._id} })  
db.accounts.update({name: t.destination},  
  { $pull: {pendingTransactions: t._id} })
```

- **Step 5. Remove pending transaction for the accounts**

```
db.transactions.update({_id: t._id},  
  { $set: {state: "done"} })
```

- **Step 6. Set transaction state to **done****



MongoDB

Two-phase Commit – Failures

- Between step 1 (initial state) and 3 (application)
 - Applications should get a list of transactions in the pending state and resume from step 2 (switch to pending)
- Between step 3 (application) and step 6 (setting as done)
 - Application should get a list of transactions in the applied state and resume from step 5 (remove pending)

MongoDB

Two-phase Commit – Rollback

- When the application needs to “cancel” the transaction
 - e.g., it can never recover since one of the accounts does not exist/stops existing during the transaction, ...
- Cases:
 - **After application of transaction** (step 3) – create an inverse transaction
 - e.g., switch the values in source and destination fields
 - **After creation of transaction** (step 1) – execute rollback (see next slide)

MongoDB

Two-phase Commit – Rollback

```
db.transactions.update({_id: t._id},  
  {$set: {state: "cancelling"}})
```

- Set the transaction to **cancelling**

```
db.accounts.update({name: t.source,  
  pendingTransactions: t._id}, {$inc: {balance:  
  t.value}, $pull: {pendingTransactions: t._id}})
```

```
db.accounts.update({name: t.destination,  
  pendingTransactions: t._id}, {$inc: {balance: -  
  t.value}, $pull: {pendingTransactions: t._id}})
```

- Undo the transaction

```
db.transactions.update({_id: t._id}, {$set: {state:  
  "cancelled"}})
```

- Set the transaction to **cancelled**

Atomic operation: If in pending, undo and remove from pending

MongoDB

Two-phase Commit – Multiple Applications

- Requirement: only one application can handle a given transaction at any point in time
- Solution:
 1. Create a marker in the transaction document to **identify executing application**
 2. Use `findAndModify` method to modify the transaction

```
t = db.transactions.findAndModify(  
  {query: {state: "initial", application: {$exists: 0}},  
  update: {$set: {state: "pending", application: "A1"}},  
  new: true})
```

- Atomically modifies and returns the document, if the application is not specified

MongoDB Enterprise



- Commercial edition of MongoDB
- Includes:
 - **Advanced Security** – Kerberos authentication
 - **Management Service** – a suite of tools for managing MongoDB deployments
 - Monitoring, backup capabilities, helping users optimize clusters, ...
 - **Enterprise Software Integration** – SNMP support to integrate MongoDB with other tools
 - **Certified OS Support** – has been tested and certified on Red Hat/CentOS, Ubuntu, SuSE and Amazon Linux
 - ...



References

- Eric Redmond – Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Pramod J. Sadalage – Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**
- Tiny MongoDB Browser Shell: <http://try.mongodb.org/>
- MongoDB Manual: <http://docs.mongodb.org/manual/>