Modern Database Systems

Document stores

Doc. RNDr. Irena Holubova, Ph.D.
Irena.Holubova@matfyz.cuni.cz

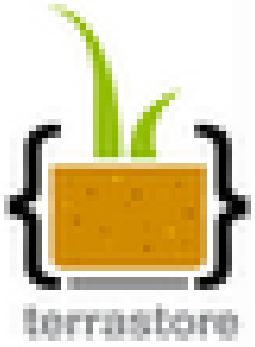
Document Databases

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

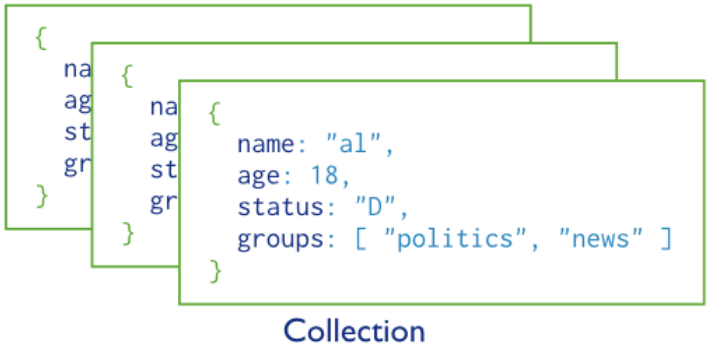
- Design of aggregate is constantly changing → we need to save the aggregates at the lowest level of granularity
  - i.e., to normalize the data

mongoDB

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

http://www.mongoDB.org/

mongoDB
Terminology



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

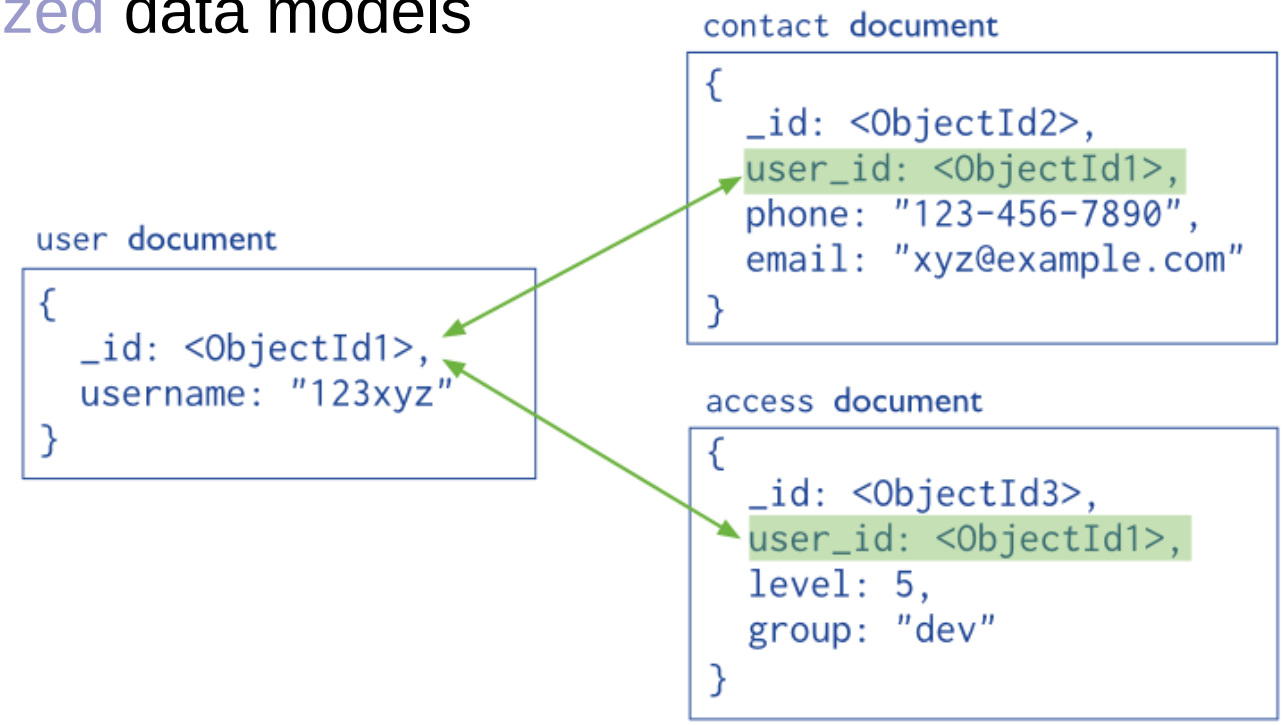
Data Model

- 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 provide more flexibility than embedding
- Use normalized data models:
  - When embedding would result in duplication of data not outweighted 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

mongoDB

Data Model – Embedded Data

- Use embedded data models:
  - 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

BSON (Binary JSON)

- Binary-encoded serialization of JSON documents
  - Allows embedding of documents, arrays, JSON simple data types + other types (e.g., date)

```
{"hello": "world"}                →  \x16\x00\x00\x00                  // total document size
                                     \x02                            // 0x02 = type String
                                     hello\x00                       // field name
                                     \x06\x00\x00\x00world\x00       // field value
                                     \x00                            // 0x00 = type EOO ('end of object')

{"BSON": ["awesome", 5.05, 1986]} →  \x31\x00\x00\x00
                                     \x04BSON\x00
                                     \x26\x00\x00\x00
                                     \x02\x30\x00\x08\x00\x00\x00awesome\x00
                                     \x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40
                                     \x10\x32\x00\xc2\x07\x00\x00
                                     \x00
                                     \x00
```

http://bsonspec.org/

BSON Basic Types

- `byte` – 1 byte (8-bits)
- `int32` – 4 bytes (32-bit signed integer)
- `int64` – 8 bytes (64-bit signed integer)
- `double` – 8 bytes (64-bit IEEE 754 floating point)

BSON Grammar

```
document ::= int32 e_list "\x00"
```

- BSON document
- `int32` = total number of bytes

```
e_list ::= element e_list | ""
```

- Sequence of elements

BSON Grammar

```
element ::= "\x01" e_name double
   | "\x02" e_name string
   | "\x03" e_name document
   | "\x04" e_name document
   | "\x05" e_name binary
   | …
```

- Floating point
- UTF-8 string
- Embedded document
- Array
- Binary data
- …

```
e_name ::= cstring
```

- Key name

```
cstring ::= (byte*) "\x00"
string ::= int32 (byte*) "\x00"
```

- String = `int32` bytes

And so on…

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

Document

```
db.users.insert(
                  {
                     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 has `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`

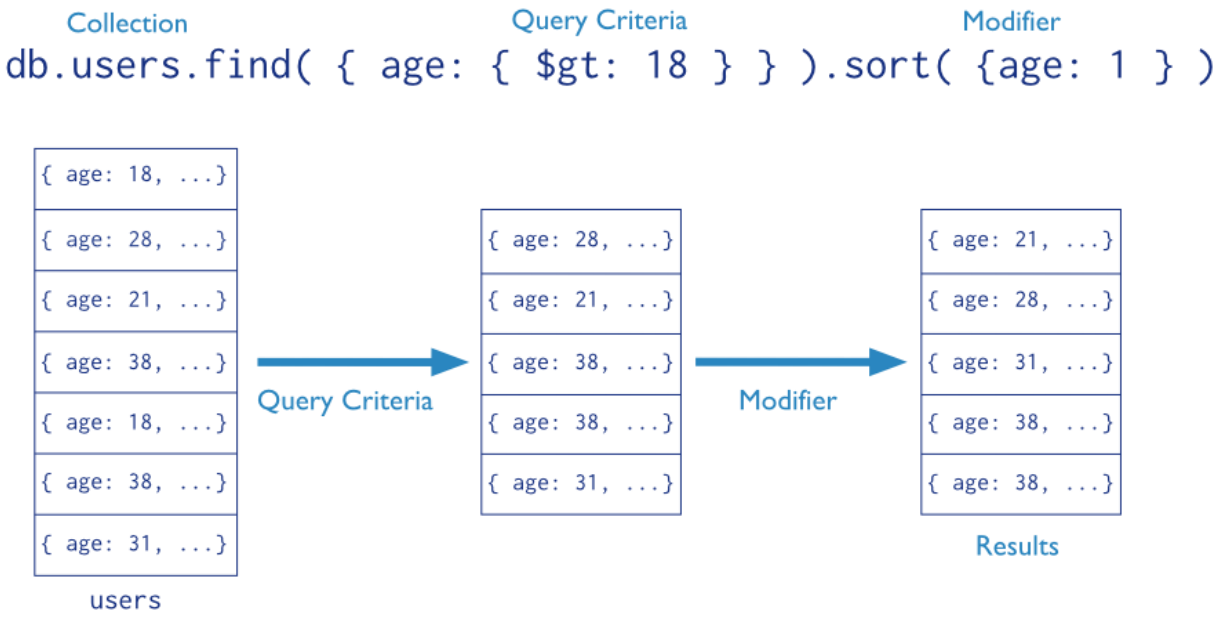
```
db.inventory.save(
    {
      _id: 10,
      type: "misc",
      item: "placard"
    } )
```

- 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 (`$lt`) `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 `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 `9.95`

mongoDB

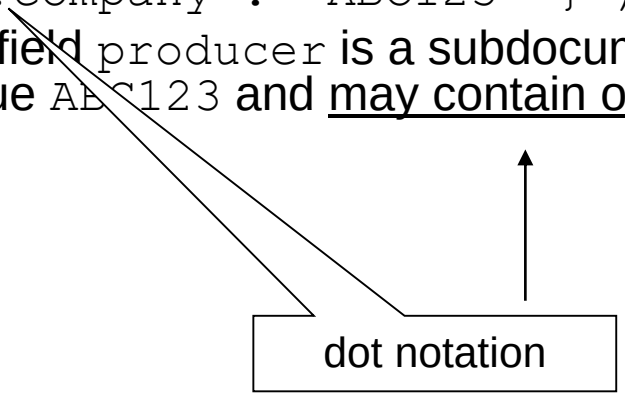
Query – Subdocuments

```
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



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

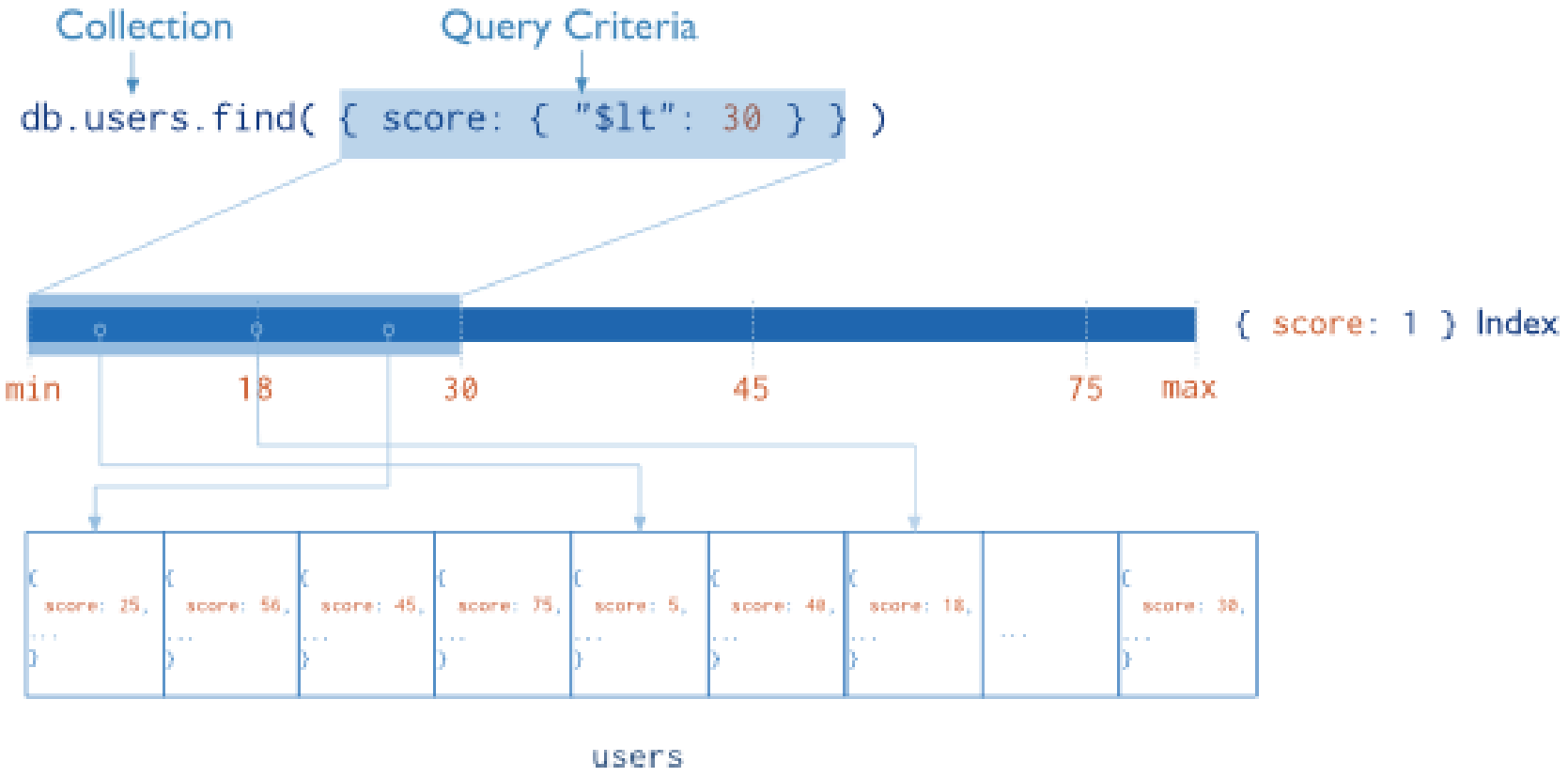
mongoDB

Indices

- Without indices:
  - mongoDB must scan every document in a collection to select those documents that match the query statement
- Indices store a portion of the collection's data set in an easy to traverse form
  - Stores the value of a specific field or a 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

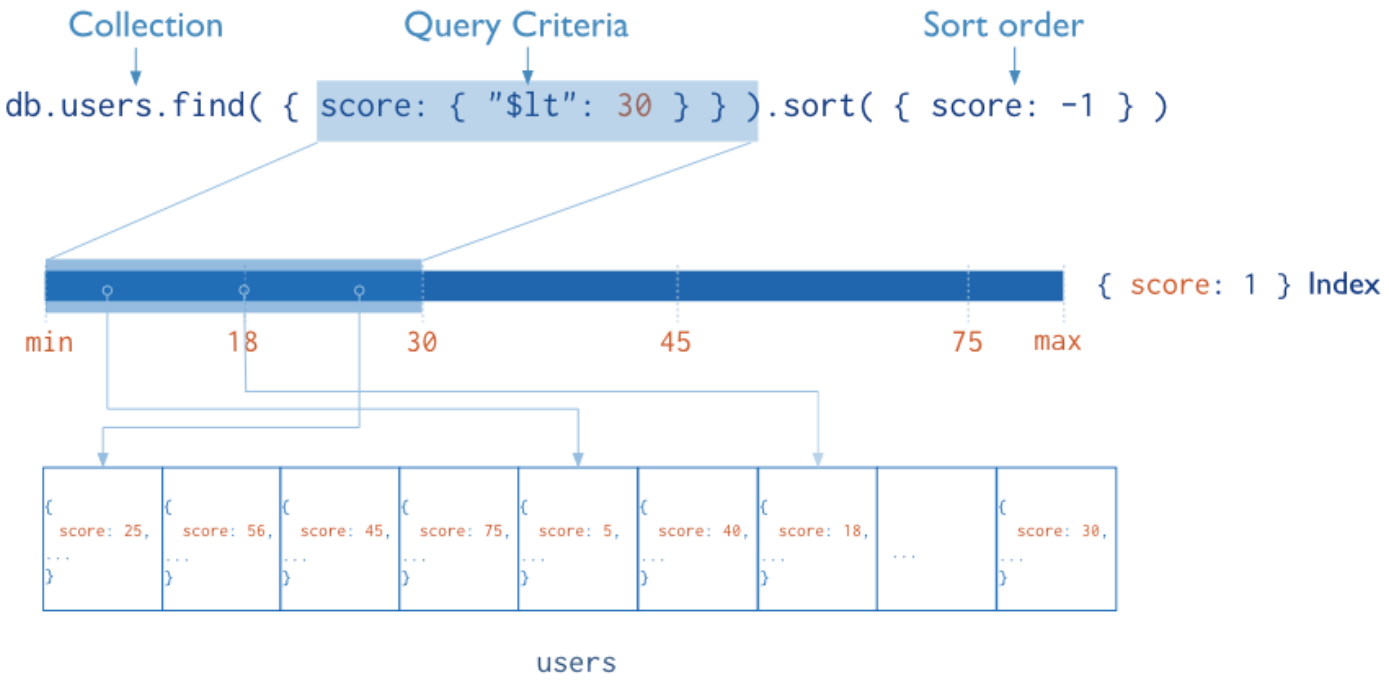
mongoDB

Indices – Example



mongoDB

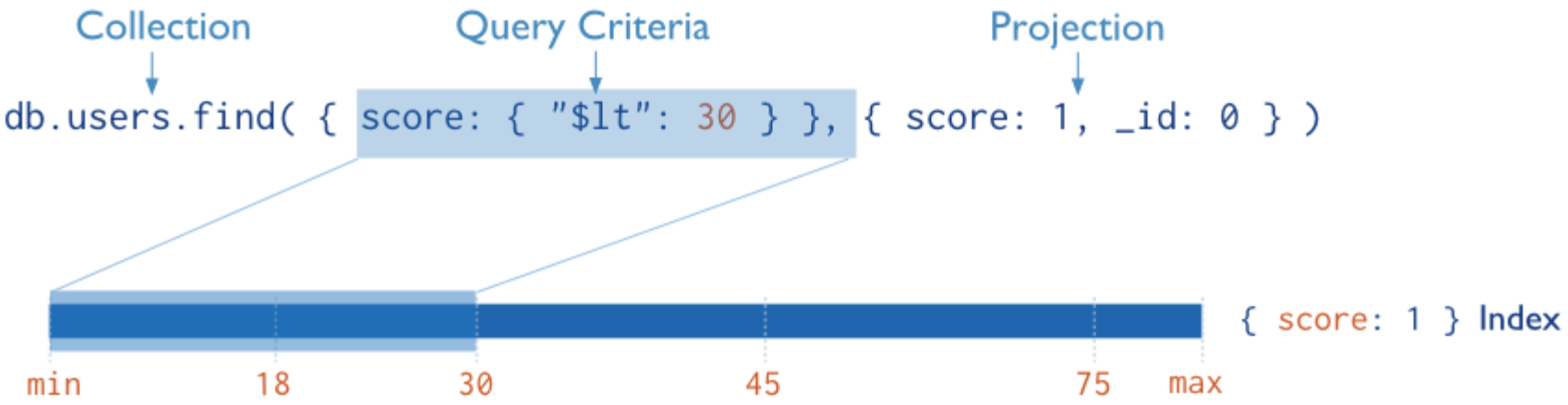
Indices – Usage for Sorted Results



- 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

Indices – Usage for Covered Results

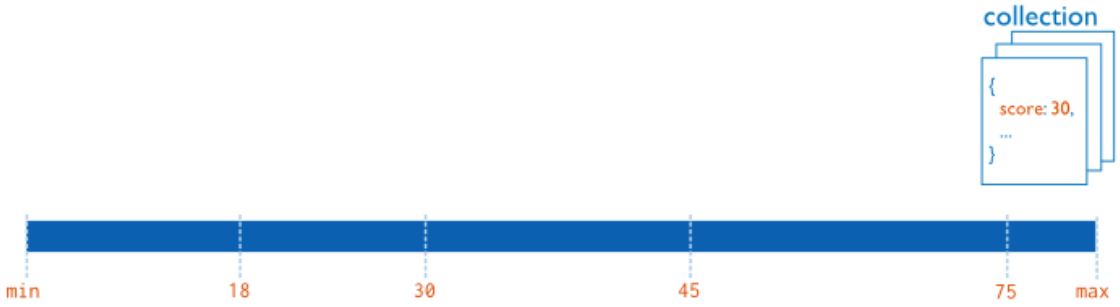


- mongoDB does not need to inspect data outside of the index to fulfil the query

mongoDB

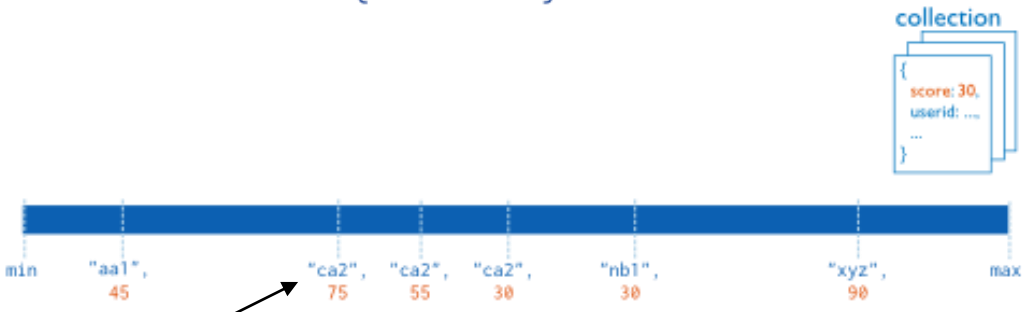
Index Types

- **Default `_id`**
  - Exists by default
    - If applications do not specify `_id`, it is created automatically
  - Unique by default
- **Single Field**
  - User-defined indices on a single field of a document
- **Compound**
  - User-defined indices on multiple fields
- **Multikey index**
  - To index the content stored in arrays
  - Creates separate index entry for every element of the array



{ score: 1 } Index

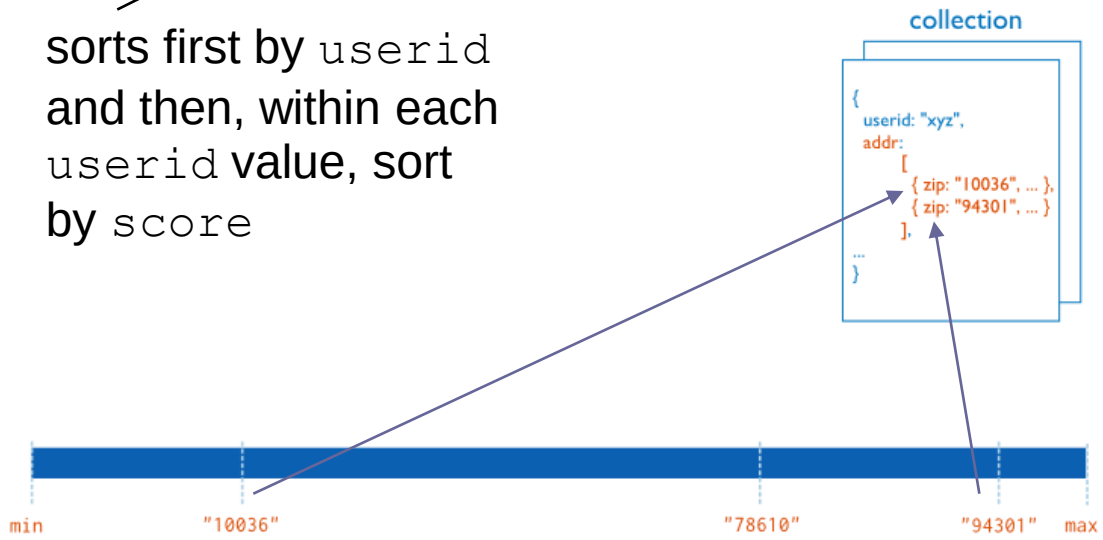
Single field index on the `score` field (ascending).



{ userid: 1, score: -1 } Index

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

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



{ "addr.zip": 1 } Index

Multikey index on the `addr.zip` field

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

Indices

```
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" } )
```

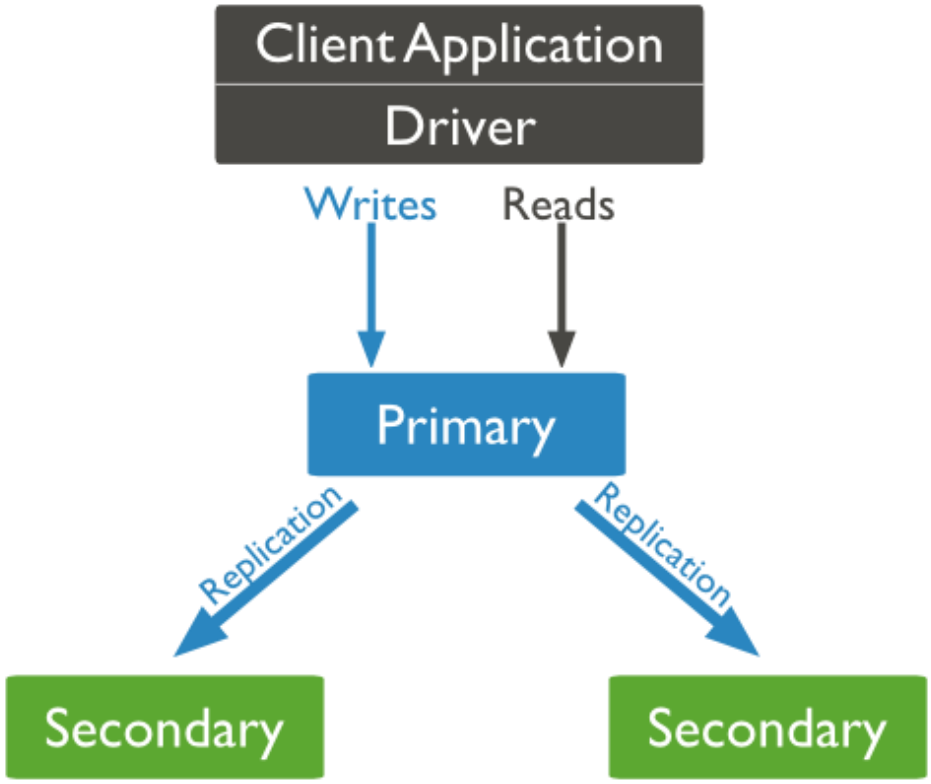
- Creates a hashed index on `_id`

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

default

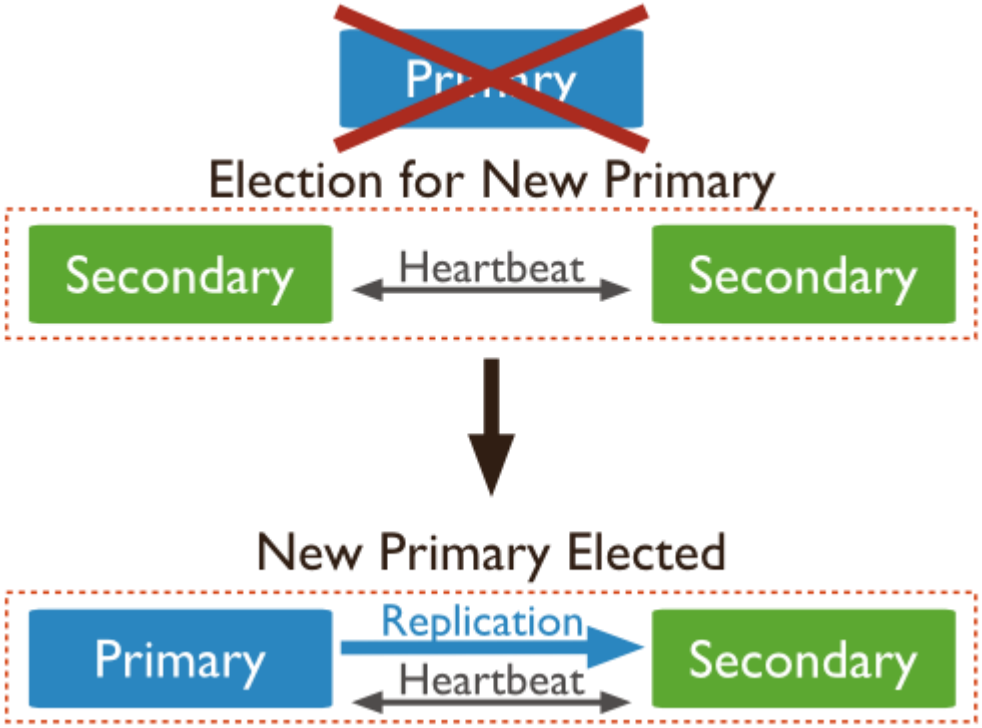
Read Preference Mode	Description
`primary`	operations read from the current replica set primary
`primaryPreferred`	operations read from the primary, but if unavailable, operations read from secondary members
`secondary`	operations read from the secondary members
`secondaryPreferred`	operations read from 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
    - Approx. 1 minute
  - No primary ⇒ no writes



mongoDB

Replica Set Elections – Influencing Factors

- Heartbeat (ping)
  - Every 2s sent to each other
  - No response for 10s ⇒ 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 ⇒ OK
  - A higher-priority member catches up to within 10s of the latest oplog entry of the current primary ⇒ 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

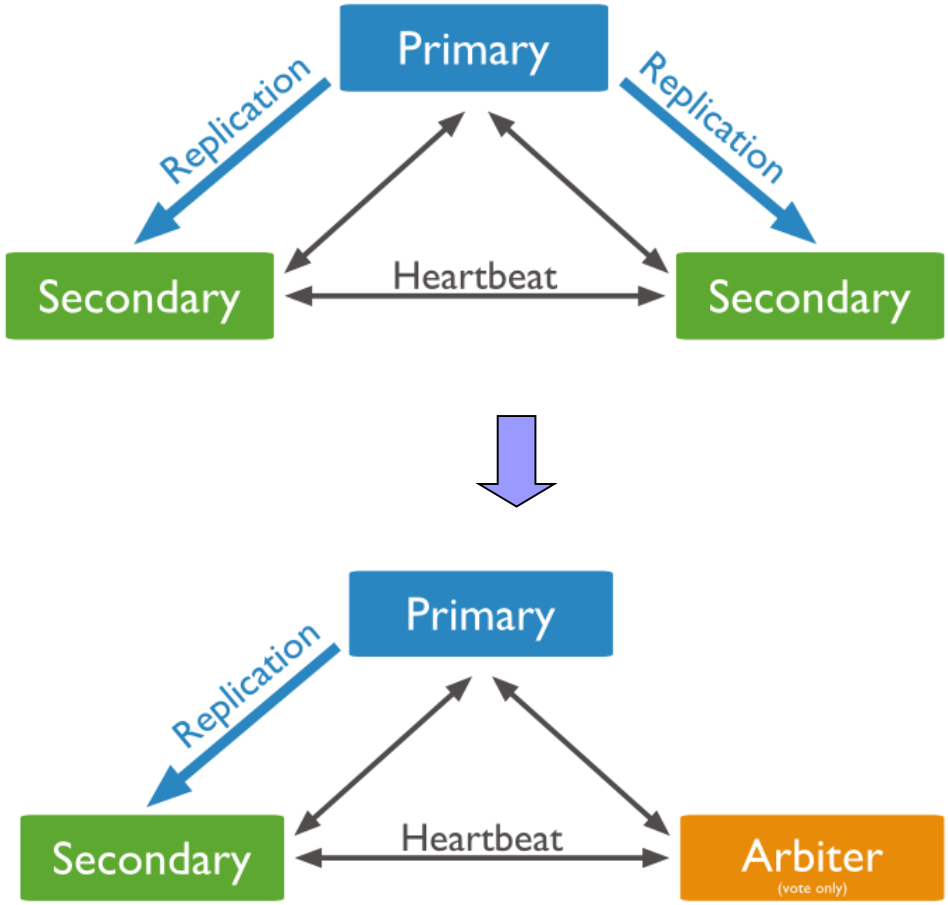
Replica Set Elections – Mechanism

- 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

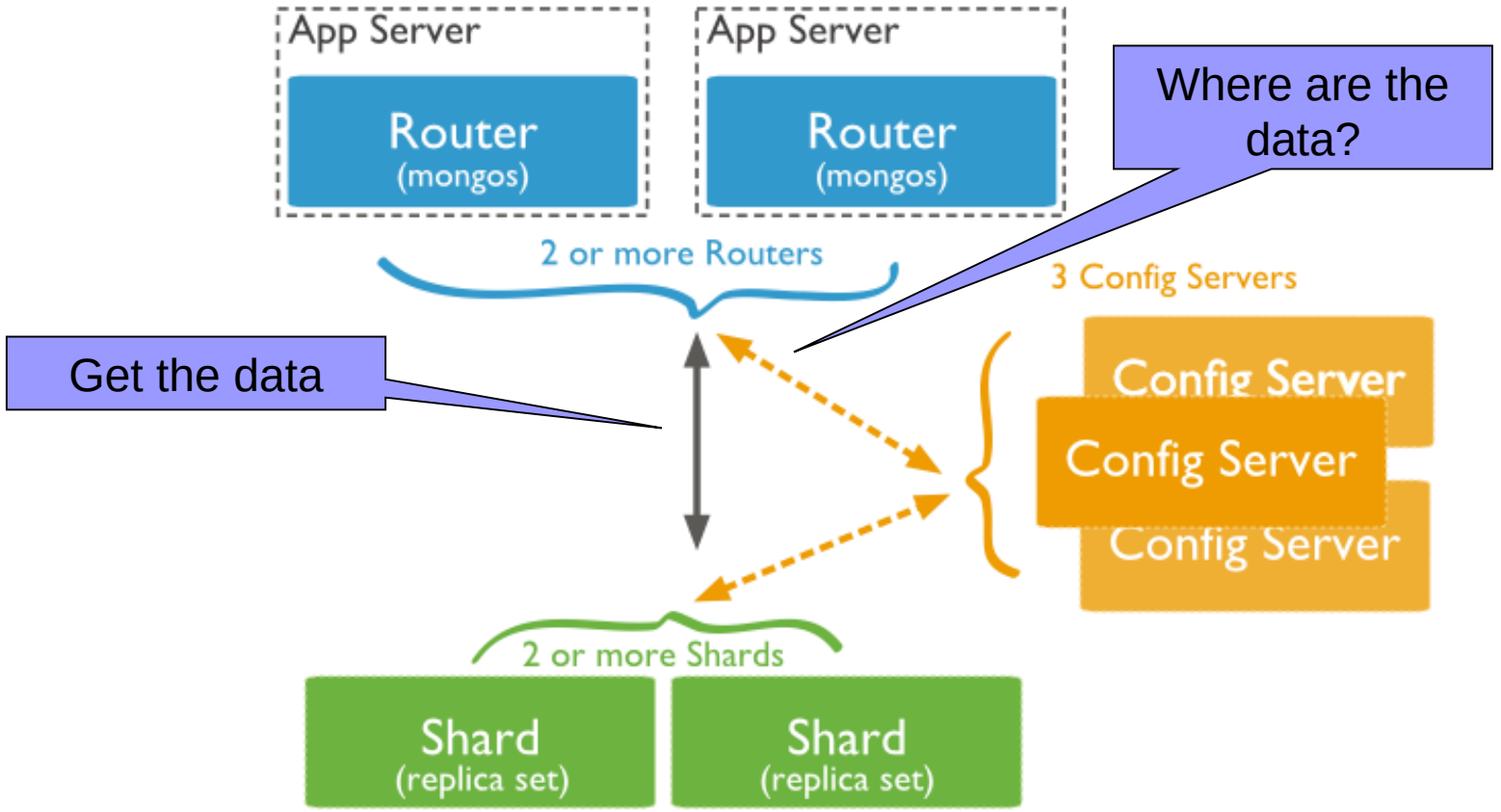
- A secondary can be configured as:
  - Priority 0 – to prevent it from becoming a primary in an election
    - e.g., a standby
  - Hidden – to prevent applications from reading from it
    - Just replicates the data for special usage
    - Can vote in elections
  - Delayed – to keep a running “historical”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

mongoDB

Sharding

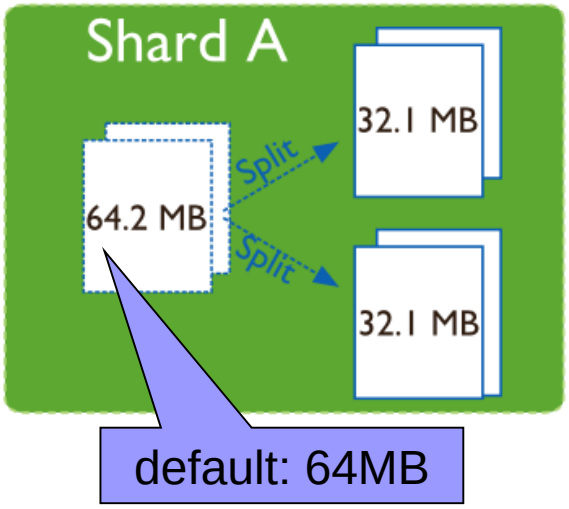
- Supported through sharded clusters
- Consisting of:
  - Shards – store the data
    - Each shard is a replica set
      - For testing purposes can be a single node
  - Query routers – interface with client applications
    - Direct operations to the appropriate shard(s) + return the result to the user
    - 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

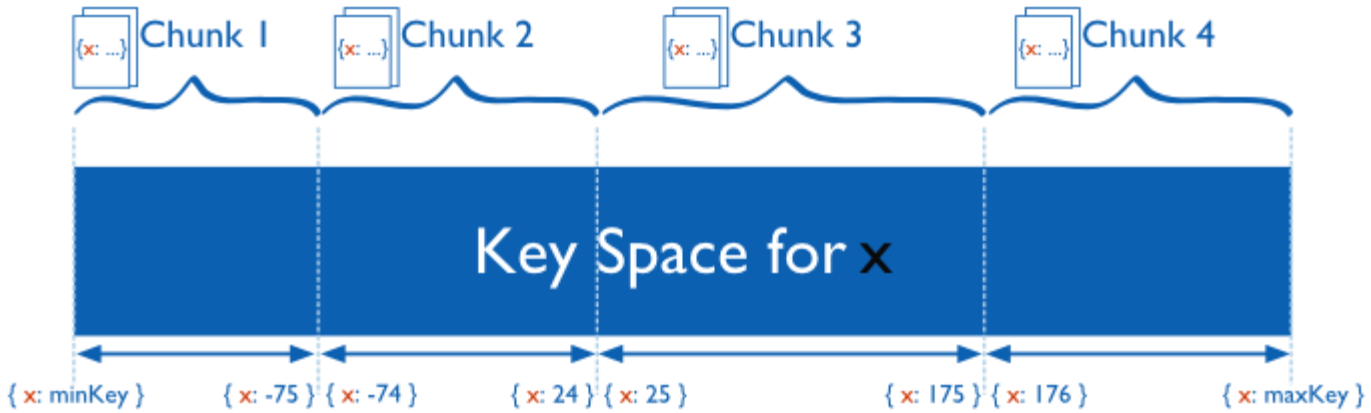
- Partitions a collection's data by the shard key
  - Indexed (possibly compound) field that exists in every document in the collection
    - Immutable
  - Divided into chunks distributed across shards
    - **Range-based partitioning**
    - **Hash-based partitioning**
  - When a chunk grows beyond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



mongoDB

Range-Based Partitioning

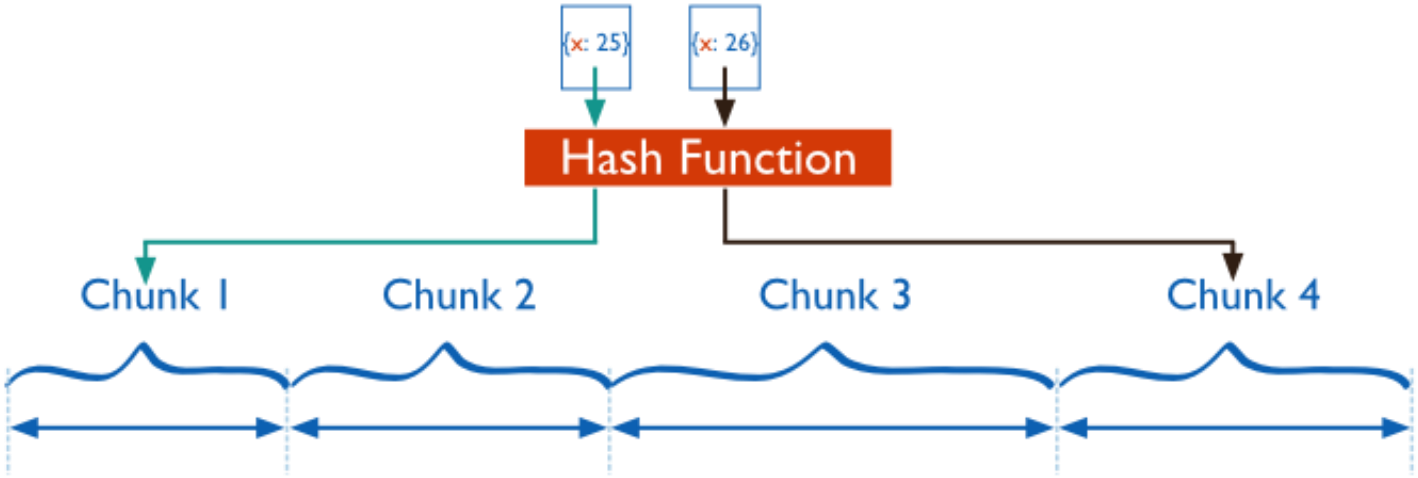
- Each value of the shard key falls at some point on line from negative infinity to positive infinity
- The line is partitioned into non-overlapping chunks
- Documents with “close” shard key values are likely to be in the same chunk
  - More efficient range queries
  - Can result in an uneven distribution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

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
    - Transaction-like semantics for multi-document updates
    - Idea:
      - Store all the information about the steps of an operation in a transaction and store the transaction
      - Retrieve the transaction and perform its steps
      - After performing each step, update the state of the transaction to reflect that.
      - A transaction is complete when it is in a final state

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**

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

```
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**

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
  - …

https://www.mongoDB.com/products/mongoDB-enterprise

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: https://mongoplayground.net/
- mongoDB Manual: http://docs.mongoDB.org/manual/