Document Databases: MongoDB

Big Data Management

Faculty of Computer Science & Information Technology
University of Malaya

December 3, 2019

Borrowed from http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/

Lecture Outline

Document databases

- Introduction

MongoDB

- Data model
- CRUD operations
  - Insert
  - Update
  - Remove
  - Find: projection, selection, modifiers

Outline

Functionality of MongoDB

- Dynamic schema
  - No DDL
- Document-based database
- Secondary indexes
- Query language via an API
- Atomic writes and fully-consistent reads
  - If system configured that way
- Master-slave replication with automated failover (replica sets)
- Built-in horizontal scaling via automated range-based partitioning of data (sharding)
- No joins nor transactions

Why use MongoDB?

- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data
  - No ERD diagram
- Not well suited for heavy and complex transactions systems

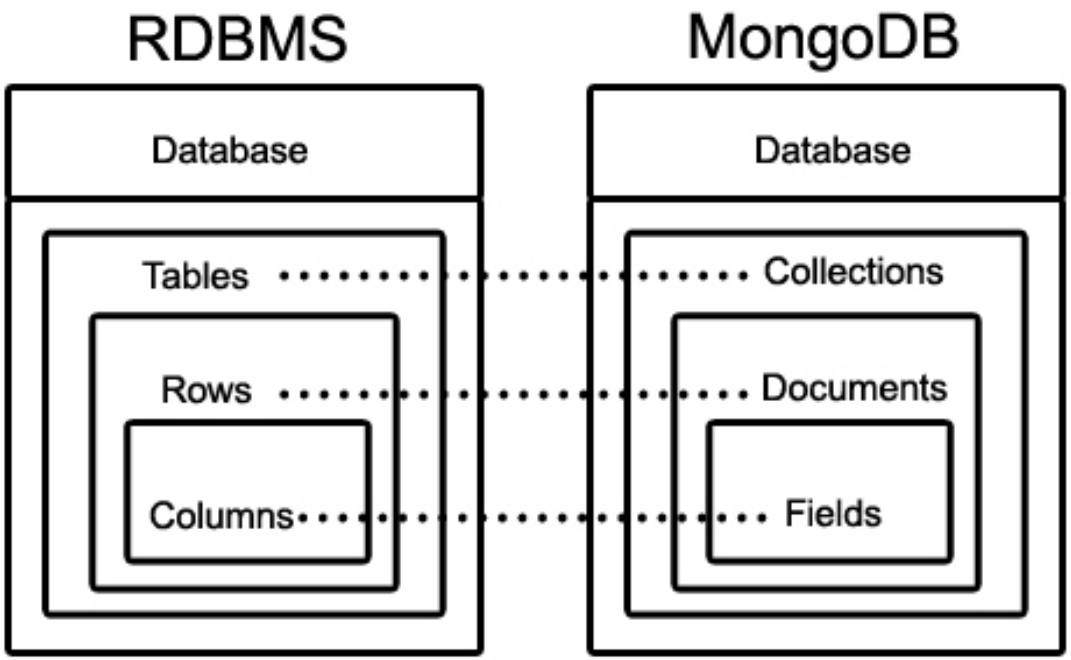
MongoDB: CAP approach

Focus on Consistency and Partition tolerance

- **Consistency**
  - all replicas contain the same version of the data
- **Availability**
  - system remains operational on failing nodes
- **Partition tolarence**
  - multiple entry points
  - system remains operational on system split

MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more **databases**
- A database may have zero or more **collections**
- A collection may have zero or more **documents**
- A document may have one or more **fields**
- MongoDB **Indexes** function much like their RDBMS counterparts.



RDB Concepts to NO SQL

- Database ⇒ Database
- Table, View ⇒ Collection
- Row ⇒ Document (BSON)
- Column ⇒ Field
- Index ⇒ Index
- Join ⇒ Embedded Document
- Foreign Key ⇒ Reference
- Partition ⇒ Shard

Collection is not strict about what it stores

- Schema-less

Hierarchy is evident in the design

- Embedded Document

MongoDB Processes and configuration

- **Mongod** - Database instance
- **Mongos** - Sharding processes
  - Analogous to a **database router.**
  - Processes all requests
  - Decides how many and which mongods should receive the query
  - Collates the results, and sends it back to the client.
- **Mongo** – an interactive shell (a client)
  - Fully functional JavaScript environment for use with a MongoDB
- You can have **one mongos** for the whole system no matter how many mongods you have
- OR you can have **one local mongos** for every client if you wanted to minimize network latency.

Choices made for Design of MongoDB

- **Scale horizontally** over commodity hardware
  - Lots of relatively inexpensive servers
- **Keep the functionality** that works well in RDBMSs
  - Ad hoc queries
  - Fully featured indexes
  - Secondary indexes
- What doesn't distribute well in RDB?
  - Long running multi-row transactions
  - Joins
  - Both artifacts of the relational data model (row x column)

BSON format

- Binary-encoded serialization of JSON-like documents
- Zero or more key/value pairs are stored as a single entity
- Each entry consists of a field name, a data type, and a value
- Large elements in a BSON document are prefixed with a length field to facilitate scanning

Schema Free

- MongoDB does **not** need any **pre-defined data schema**
- Every document in a collection could have **different data**
  - Addresses **NULL** data fields

JSON format

Data is in name/value pairs

- A name/value pair consists of a field name followed by a colon, followed by a value:
  - Example: ′name′: ′R2-D2′
- Data is separated by commas
  - Example: ′name′: ′R2-D2′, race : ′Droid′
- Curly braces hold objects
  - Example: ′name′: ′R2-D2′, race : ′Droid′, affiliation: ′rebels′
- An array is stored in brackets []
  - Example [′name′: ′R2-D2′, race : ′Droid′, affiliation: ′rebels′, ′name′: ′Yoda′, affiliation: ′rebels′]

Document Stores

Data model

- Documents
  - **Self-describing**
  - Hierarchical tree structures (JSON, XML, ...)
    - Scalar values, maps, lists, sets, nested documents, ...
  - Identified by a unique identifier (key, ...)
- Documents are organized into collections

Query patterns

- Create, update or remove a document
- Retrieve documents according to complex query conditions

Observation

- **Extended key-value stores** where the value part is examinable

Outline

MongoDB

JSON document database

- https://www.mongodb.com/
- **Features**
  - Open source, high availability, eventual consistency, automatic sharding, master-slave replication, automatic failover, secondary indices, ...
- Developed by MongoDB
- Implemented in C++, C, and JavaScript
- Operating systems: Windows, Linux, Mac OS X, ...
- Initial release in **2009**

Query Example

Collection of movies

```
{
  _id: ObjectId("1"),
  title: "The Double",
  year: 2006
}
```

```
{
  _id: ObjectId("2"),
  title: "Zombieland",
  year: 2000
}
```

```
{
  _id: ObjectId("3"),
  title: "The Social Network",
  year: 2007
}
```

Query statement

Titles of movies filmed in 2005 and later, sorted by these titles in descending order

```
db.movies.find(
  { year: { $gt: 2005 } },
  { _id: false, title: true }
).sort({ title: -1 })
```

Query result

```
{ title: "The Double" }
```

```
{ title: "The Social Network" }
```

Data Model

Database system structure

Instance → databases → collections → documents

1. **Database**
2. **Collection**
   - Collection of documents, usually of a **similar structure**
3. **Document**
   - MongoDB document = **one JSON object**
     - i.e. even a complex JSON object with other recursively nested objects, arrays or values
   - Each document has a **unique identifier** (primary key)
     - Technically realized using a **top-level _id field**

Data Model

MongoDB document

- Internally stored in **BSON format** (Binary JSON)
  - Maximal allowed size **16 MB**
  - **GridFS** can be used to split larger files into smaller chunks

Restrictions on field names

- Top-level _id is **reserved** for a primary key
- Field names **cannot** start with $
  - Reserved for **query operators**
- Field names **cannot** contain .
  - Used when **accessing nested fields**

Primary Keys

Features of identifiers

- **Unique** within a collection
- **Immutable** (cannot be changed once assigned)
- Can be of **any type** other than a JSON array

Key management

1. **Natural** identifier
2. **Auto-incrementing** number – not recommended
3. **UUID** (Universally Unique Identifier)
4. **ObjectId** – special 12-byte BSON type (the default option)
   - Small, likely unique, fast to generate, ordered, based on a timestamp, machine id, process id, and a process-local counter

Design Questions

Data modeling (in terms of collections and documents)

- No explicit schema is provided, nor expected or enforced
  - However...
    - documents within a collection are **similar in practice**
    - **implicit schema** is required nevertheless
- **Challenge**
  - Balancing application requirements, performance aspects, data structure, mutual relationships, query patterns, ...

Two main concepts

1. Embedded documents
2. References

Denormalized Data Models

Embedded documents

- **Related** data in a **single document**
  - with **embedded** JSON objects, so called **subdocuments**
- **Pros:** data **manipulation** (fewer queries need to be issued)
- **Cons:** possible data **redundancies**
- Suitable for **one-to-one** or **one-to-many** relationships

```
1 {
2   _id: ObjectId("2"), title: "Zombieland", year: 2000,
3   actors: [
4     { firstname: "Emma", lastname: "Stone" },
5     { firstname: "Woody", lastname: "Harrelson" },
6     { firstname: "Jesse", lastname: "Eisenberg" }
7   ]
8 }
```

Normalized Data Models

References

- **Related** data in **separate documents**
  - These are interconnected via **directed links** (references)
  - Technically expressed using ordinary values with **identifiers of target documents** (i.e. no special construct is provided)
- **Features:** higher flexibility, follow up queries might be needed
- Suitable for **many-to-many** relationships

```
{
  _id: ObjectId("2"),
  title: "Zombieland",
  year: 2000,
  actors: [
    ObjectId("6"),
    ObjectId("4"),
    ObjectId("5") ]
}
```

```
{
  _id: ObjectId("6"),
  firstname: "Emma",
  lastname: "Stone"
}
```

Sample Data

Collection of movies

```
{
  _id: ObjectId("1"),
  title: "The Double", year: 2006,
  actors: [ ObjectId("7"), ObjectId("5") ]
}
```

```
{
  _id: ObjectId("2"),
  title: "Zombieland", year: 2000,
  actors: [ ObjectId("6"), ObjectId("4"), ↩
       ObjectId("5") ]
}
```

```
{
  _id: ObjectId("3"),
  title: "The Social Network", year: 2007,
  actors: [ ObjectId("5"), ObjectId("4") ]
}
```

Collection of actors

```
{   _id: ObjectId("4"),
  firstname: "Woody",
  lastname: "Harrelson" }
```

```
{   _id: ObjectId("5"),
  firstname: "Jesse",
  lastname: "Eisenberg" }
```

```
{   _id: ObjectId("6"),
  firstname: "Emma",
  lastname: "Stone" }
```

```
{   _id: ObjectId("7"),
  firstname: "Mia",
  lastname: "Wasikowska" }
```

Application Interfaces

mongo shell

- Interactive interface to MongoDB
- ./bin/mongo --username user --password pass --host host --port 28015

Drivers

- Java, C, C++, C, Perl, PHP, Python, Ruby, Scala, ...

Query Language

MongoDB query language is based on **JavaScript**

- Single command/entire script
- Read queries return a cursor
  - Allows us to **iterate** over **all the selected documents**
- Each command is always **evaluated** over a **single collection**

Query patterns

- Basic CRUD operations
  - Accessing documents **via identifiers** or **conditions on fields**
- **Aggregations:** MapReduce, pipelines, grouping

CRUD Operations

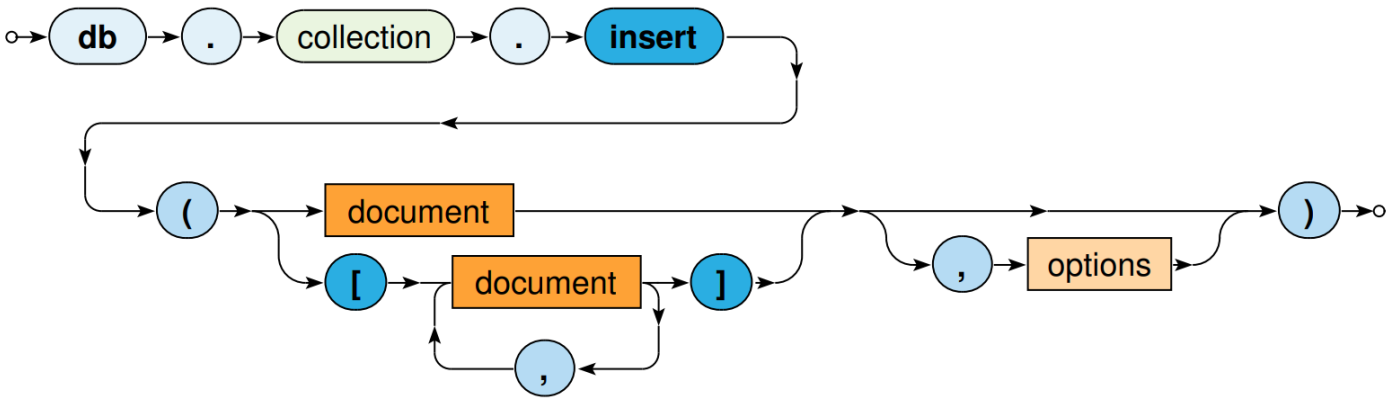
Overview

1. db.collection.insert()
   - **Inserts** a new document into a collection
2. db.collection.update()
   - **Modifies** an existing document/documents or inserts a new one
3. db.collection.remove()
   - **Deletes** an existing document/documents
4. db.collection.find()
   - **Finds** documents based on filtering conditions
   - **Projection** and/or **sorting** may be applied too

Outline

Insert Operation

Inserts a new document/documents into a given collection



- **Parameters**
  1. **Document:** one or more documents to be inserted
     - Provided document identifiers (_id fields) must be **unique**
     - When **missing**, they are **generated automatically** (ObjectId)
  2. **Options**
- Collections are **created automatically** when **not yet exist**

Insert Operation: Examples

Insert a new actor document

```
db.actors.insert(
  {
    firstname: "Jennifer",
    lastname: "Lawrence"
  }
)
```

```
{
  _id: ObjectId("8"),
  firstname: "Jennifer",
  lastname: "Lawrence"
}
```

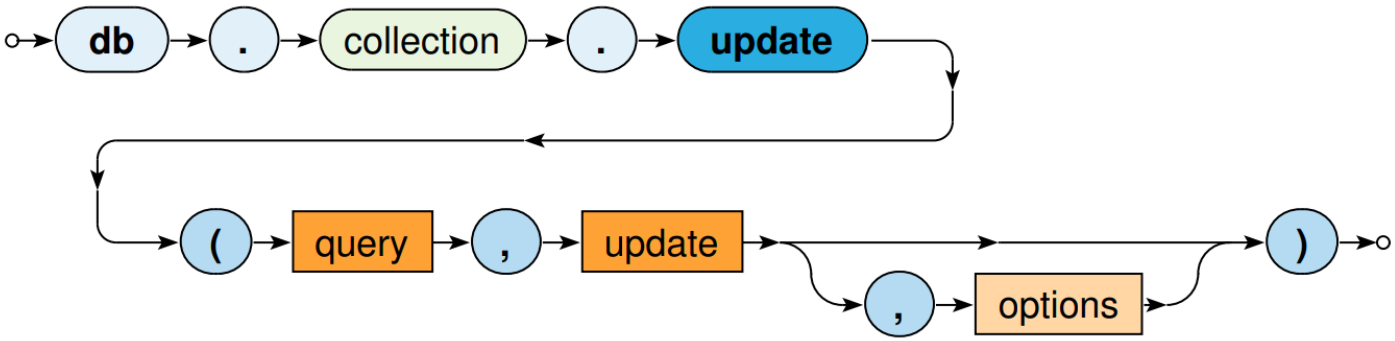
Insert two new movies

```
db.movies.insert(
  [
    {
      _id: ObjectId("9"), title: "The Hunger Games", year: 2003,
      actors: [ ObjectId("4"), ObjectId("8") ]
    },
    { title: "Passengers", year: 2016, actors: [ ObjectId("8") ] },
  ]
)
```

Outline

Update Operation

Modifies/replaces an existing document/documents



- **Parameters**
  1. **Query:** description of documents to be updated
     - The **same** behavior as in find operations
  2. **Update:** modification actions to be applied
  3. **Options**
- **At most one document** is updated by default
  - Unless { **multi: true** } option is specified

Update Operation: Examples

Replace the whole document of at most one specified actor

```
db.actors.update(
  { _id: ObjectId("8") },
  {   firstname: "Ana",
    lastname: "Lawrence" }
)
```

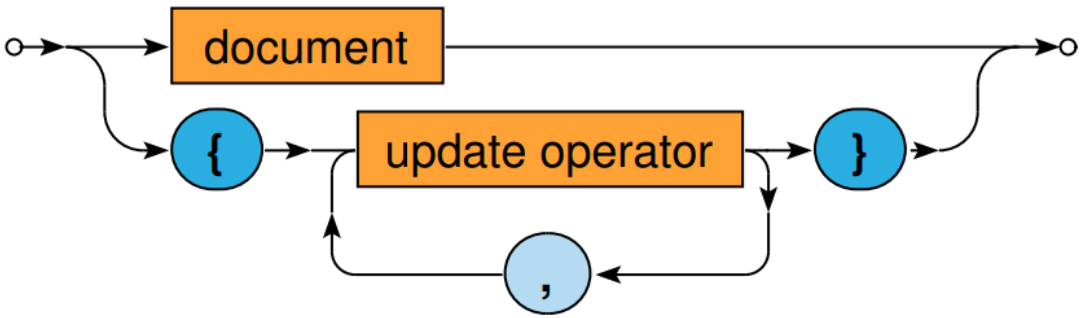
```
{
  _id: ObjectId("8"),
  firstname: "Ana",
  lastname: "Lawrence"
}
```

Update all movies filmed in 2015 or later

```
db.movies.update(
  { year: { $gt: 2015 } },
  {
    $set: { new: true },
    $inc: { rating: 3 }
  },
  { multi: true }
)
```

Update Operation

Update/replace modes

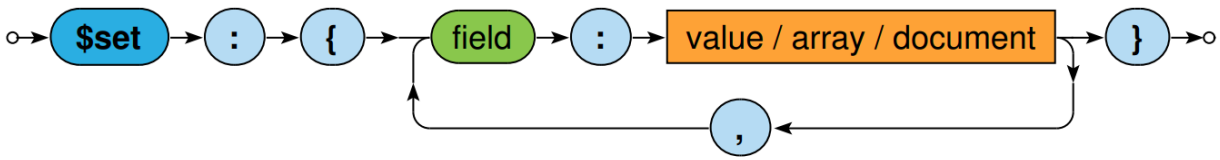


1. **Replace**
   - when the update parameter contains **no update operators**
   - The **whole document** is **replaced** (_id is preserved)
2. **Update**
   - when the update parameter contains **only update operators**
   - **Current document** is **updated** using these operators
     - $set, $unset, $inc, $mul, ...
     - Each operator can be used **at most once**

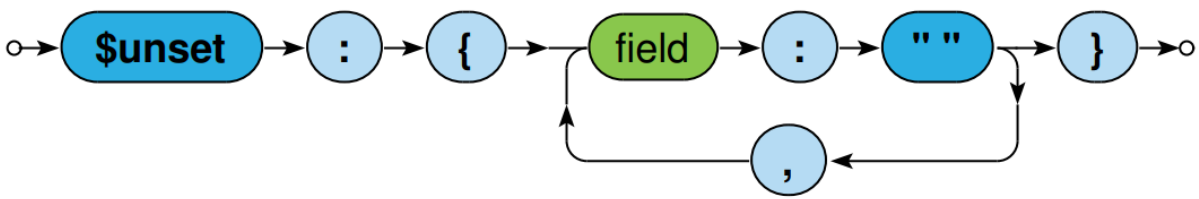
Update Operators

Field operators

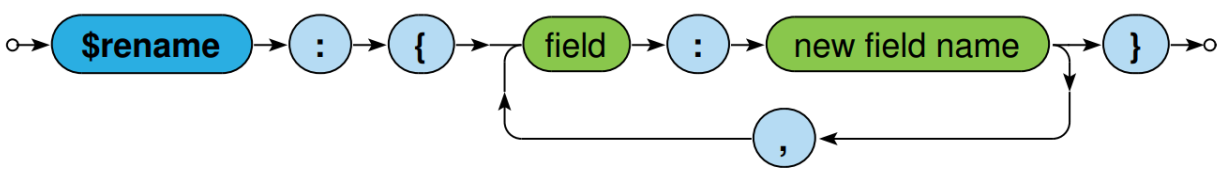
1. **$set** – sets the value of a given field/fields



2. **$unset** – removes a given field/fields



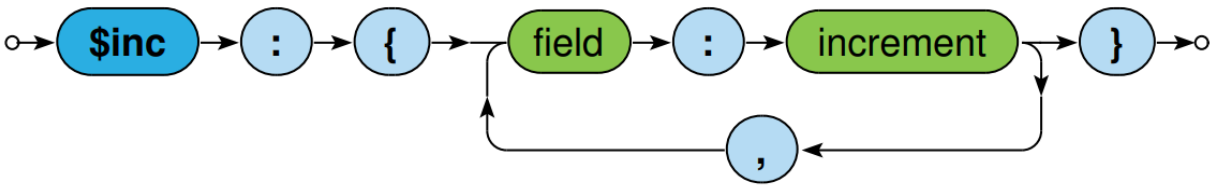
3. **$rename** – renames a given field/fields



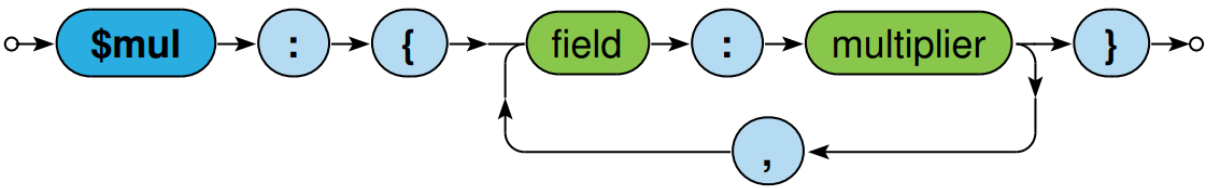
Update Operators

Field operators

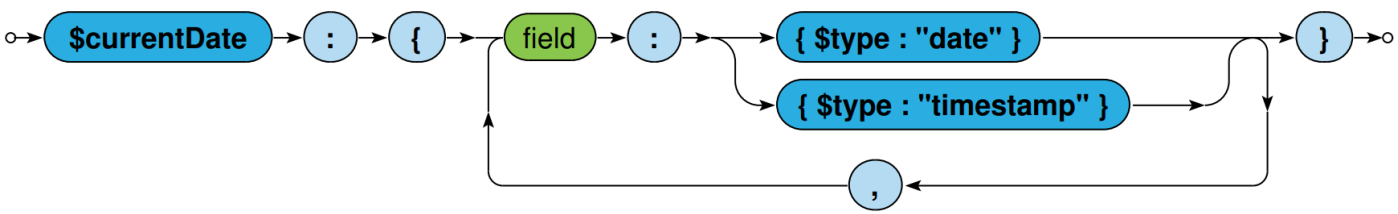
1. **$inc** – increments the value of a given field/fields



2. **$mul** – multiplies the value of a given field/fields



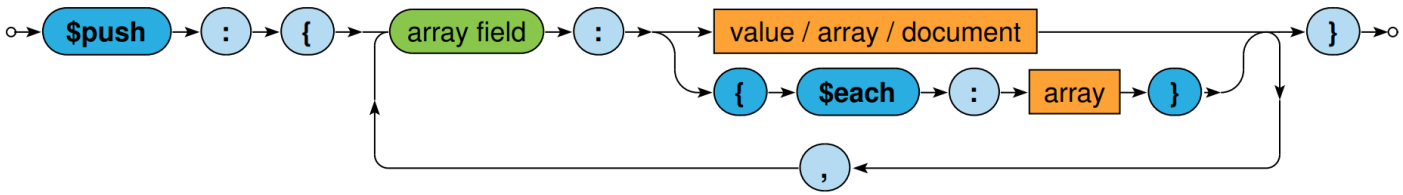
3. **$currentDate** – stores the current date time/timestamp to a given field/fields



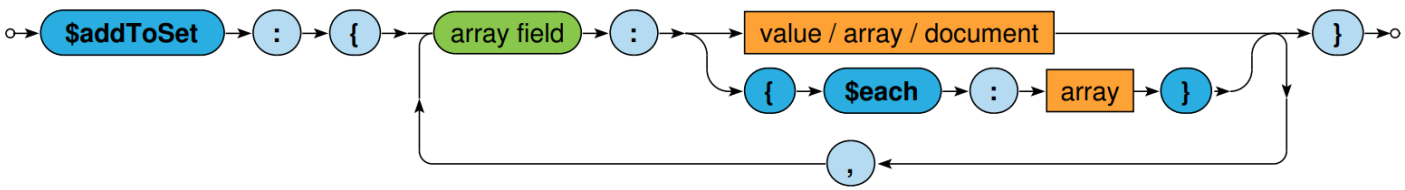
Update Operators

Array operators

1. **$push** – adds one item/all items to the end of an array



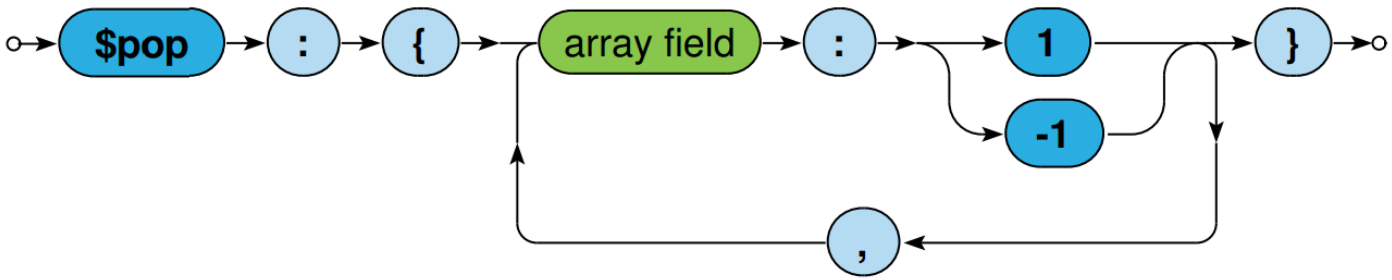
2. **$addToSet** – adds one item/all items to the end of an array, but duplicate values are ignored



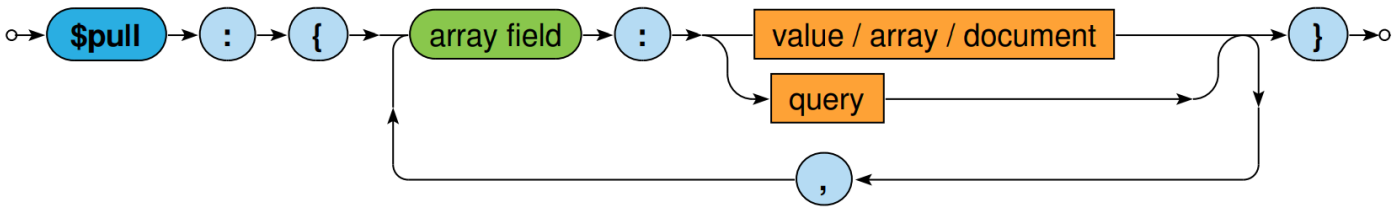
Update Operators

Array operators

1. **$pop** – removes the first/last item of an array



2. **$pull** – removes all array items that match a specified query



Upsert Mode

Upsert behavior of update operation

- When { **upsert: true** } option is **specified**, and,
- **no document** was **updated** ⇒ new document is **inserted**

What this document will contain?

- In case of the **replace mode**...
  - All the **fields** (i.e. value fields) from the **update parameter**
- In case of the **update mode**...
  - All the **value fields** from the **query parameter**, and
  - All the **outcome** of all the **update operators** from the update parameter
- _id field is **preserved**, or **newly generated** if necessary

Upsert Mode: Example

Unsuccessful update of a movie resulting to an insertion

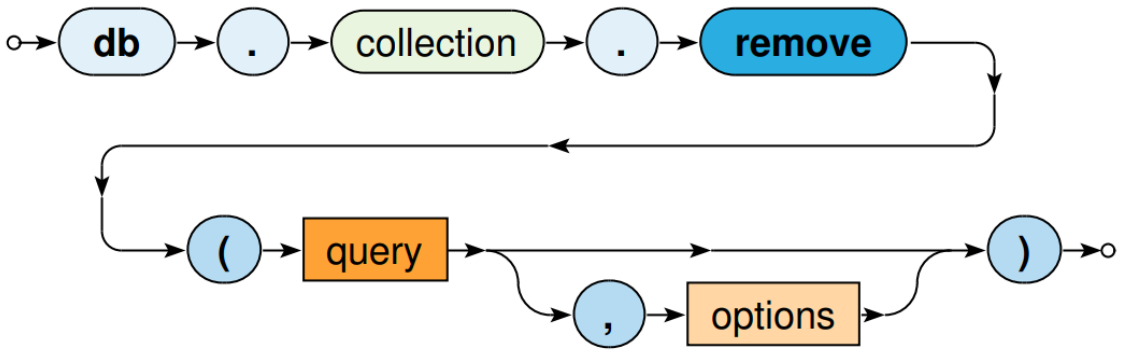
```
db.movies.update(
  { title: "Joy", year: { $gt: 2000 } },
  {
    $set: {
      director: { firstname: "David", lastname: "Russell" },
      year: 2001
    },
    $inc: { rating: 2 }
  },
  { upsert: true }
)
```

```
{ _id: ObjectId("11"),
  title: "Joy",
  director: { firstname: "David", lastname: "Russell" },
  year: 2001,
  rating: 2
}
```

Outline

Remove Operation

Removes a document/documents from a given collection

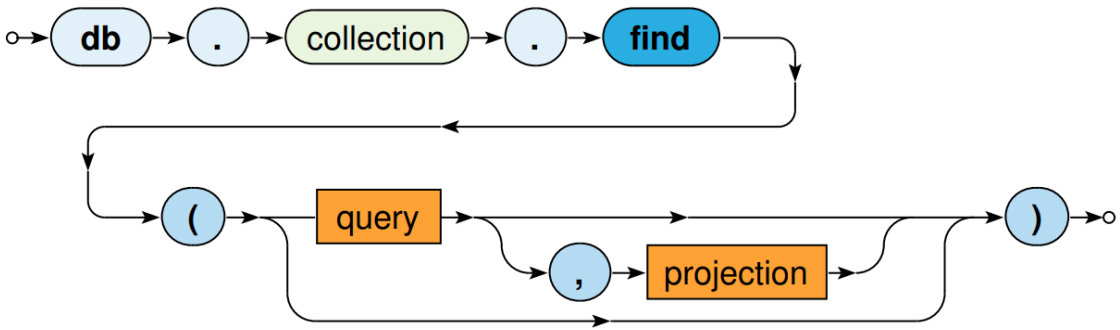


- **Parameters**
  1. **Query:** description of documents to be removed
     - The **same behavior** as in find operations
  2. **Options**
- All the matching documents are **removed unless** { justOne: true } option is provided

Outline

Find Operation

Selects documents from a given collection



- **Parameters**
  1. **Query:** description of documents to be **selected**
  2. **Projection:** fields to be **included/excluded** in the result
- Matching documents are returned via an **iterable cursor**
  - This allows us to chain further **sort**, **skip** or **limit** operations

Find Operation: Examples

Select all movies from our collection

```
db.movies.find()
```

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

Select a particular movie based on its document identifier

```
db.movies.find( { _id: ObjectId("2") } )
```

Select movies filmed in 2000 with a rating greater than 1

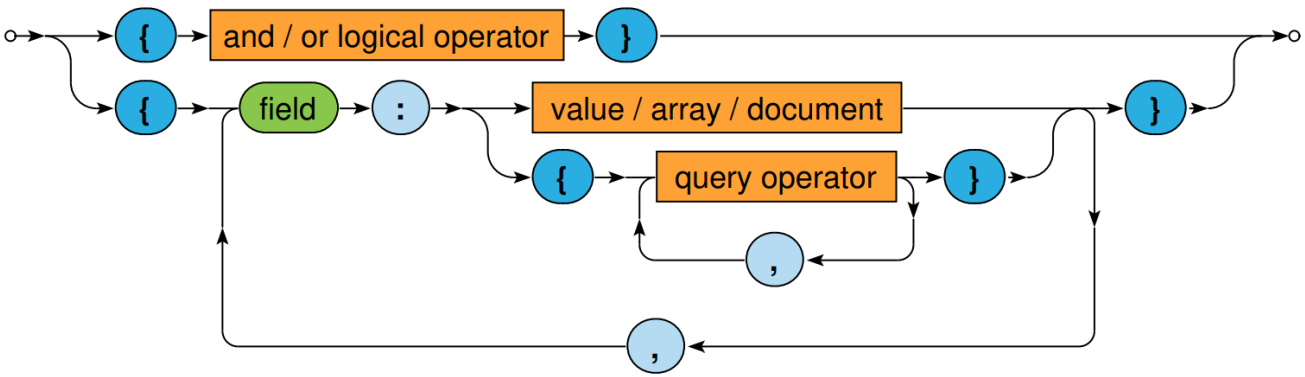
```
db.movies.find( { year: 2000, rating: { $gt: 1 } } )
```

Select movies filmed between 2005 and 2015

```
db.movies.find( { year: { $gte: 2005, $lte: 2015 } } )
```

Selection

Query parameter describes the documents we are interested in



Conditions on fields (each **field** can be used **at most once**)

1. **Value equality**
   - The actual field value must be **identical to** the **specified value** (including, e.g., the order of nested fields or array items)
2. **Query operators** (each operator can be used **at most once**)
   - The actual field value must **satisfy all** the provided operators

Value Equality Conditions: Examples

Select movies having a specific director

```
db.movies.find(
  { director: { firstname: "David", lastname: "Russell" } }
)
```

```
db.movies.find(
  { director: { lastname: "Russell", firstname: "David" } }
)
```

Select movies having specific actors

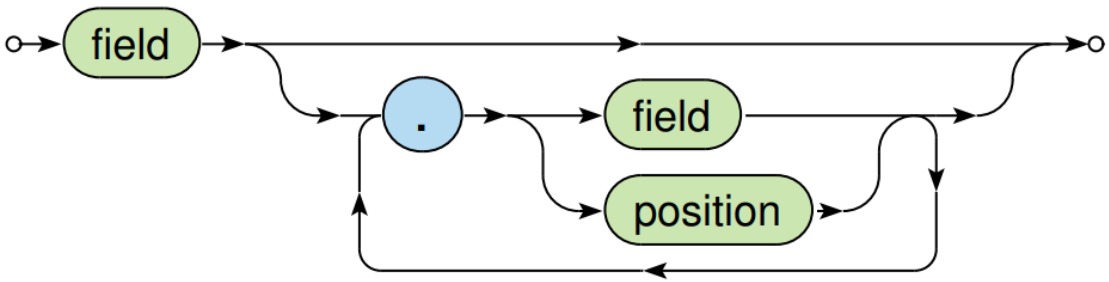
```
db.movies.find( { actors: [ ObjectId("7"), ObjectId("5") ] } )
```

```
db.movies.find( { actors: [ ObjectId("5"), ObjectId("7") ] } )
```

Queries in both the pairs are not equivalent!

Dot Notation

The dot notation for field names



1. **Accessing fields** of **embedded documents**
   - "field.subfield"
     - e.g.: "director.firstname"
2. **Accessing items** of **arrays**
   - "field.index"
     - e.g.: "actors.2"
     - Positions start at 0

Value Equality Conditions

Example (revisited)

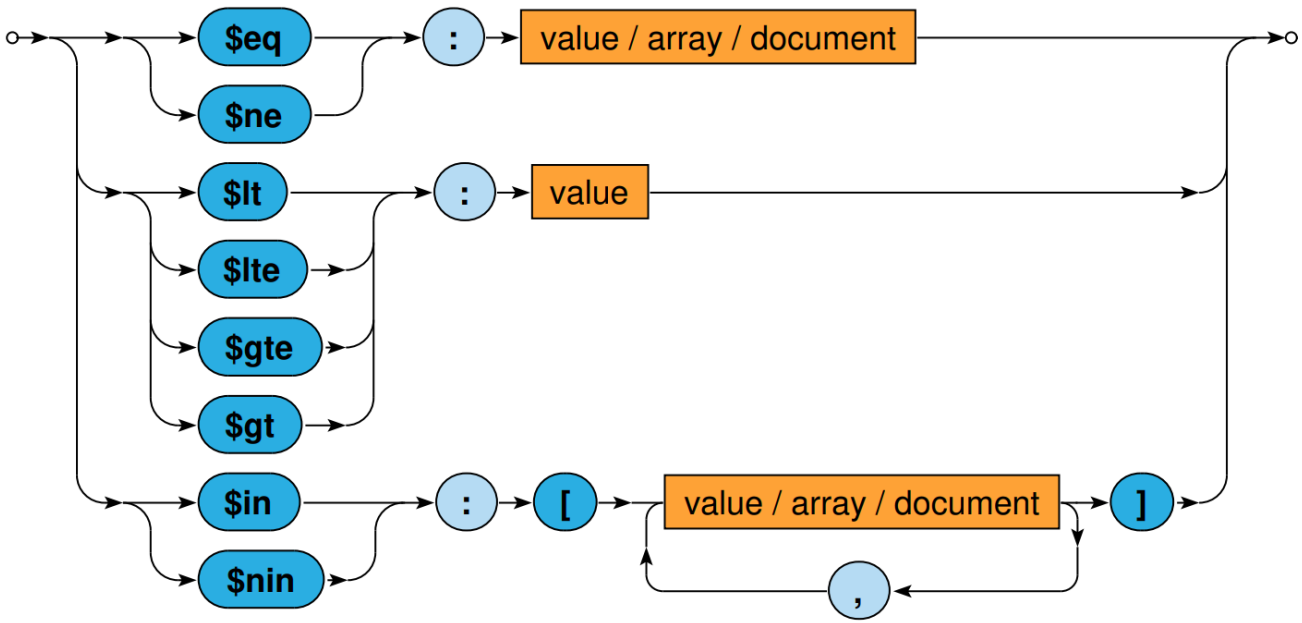
Select movies having a specific director

```
db.movies.find(
  { director: { firstname: "David", lastname: "Russell" } }
)
```

```
db.movies.find(
  { "director.firstname": "David", "director.lastname": "Russell" }
)
```

Query Operators

Comparison operators



Comparisons take particular **BSON data types** into account

- Certain **numeric conversions** are **automatically applied**

Query Operators

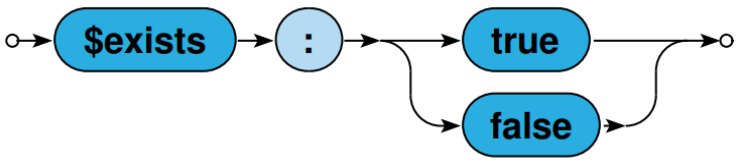
Comparison operators

1. **$eq, $ne**
   - Tests the actual field value for **equality/inequality**
     - The same behavior as in case of value equality conditions
2. **$lt, $lte, $gte, $gt**
   - Tests whether the actual field value **is** less than/less than or equal/greater than or equal/greater than the provided value
3. **$in**
   - Tests whether the actual field value is **equal to** **at least one** of the provided values
4. **$nin**
   - Negation of $in

Query Operators

Element operators

1. **$exists** – tests whether a given field **exists/not exists**



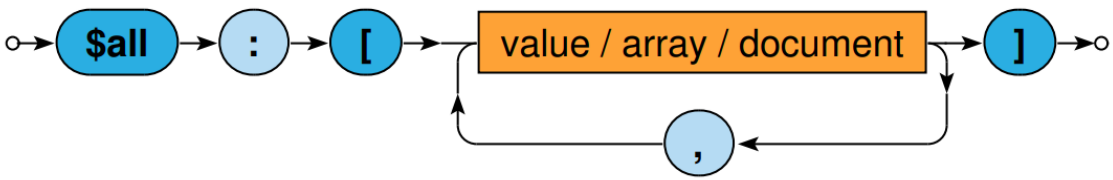
Evaluation operators

1. **$regex** – tests whether a given field value **matches** a specified **regular expression** (PCRE)
2. **$text** – **performs text search** (text index must exists)

Query Operators

Array operators

1. **$all** – tests whether a given array **contains all** the specified items **(in any order)**



Example (revisited)

Select movies having specific actors

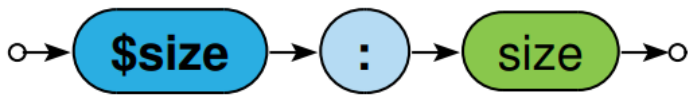
```
db.movies.find(
  { actors: [ ObjectId("5"), ObjectId("7") ] }
)
```

```
db.movies.find(
  { actors: { $all: [ ObjectId("5"), ObjectId("7") ] } }
)
```

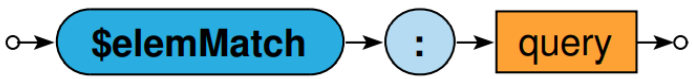
Query Operators

Array operators

1. **$size** – tests the size of a given array against a fixed number (and not, e.g., a range, unfortunately)



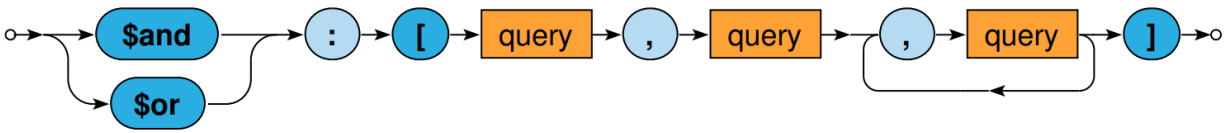
2. **$elemMatch** – tests whether a given array contains **at least one item** that satisfies all the involved query operations



Query Operators

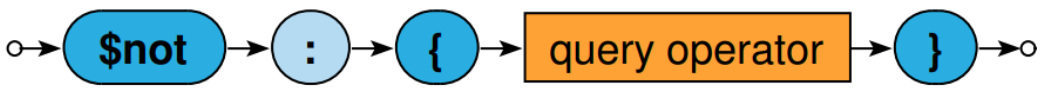
Logical operators

1. **$and, $or**



- Logical connectives for conjunction/disjunction
- At least two involved query expressions must be provided
- Only allowed at the **top level** of a query

2. **$not**



- Logical negation of **exactly one** involved query operator
- i.e. cannot be used **at the top level of a query**

Querying Arrays

Condition based on **value equality** is satisfied when...

1. the given **field** as a **whole** is **identical** to the provided value, or
2. at **least one item** of the **array** is **identical** to the provided value

```
db.movies.find( { actors: ObjectId("5") } )
```

```
{ actors: ObjectId("5") }
```

```
{ actors: [ ObjectId("5"), ObjectId("7") ] }
```

Querying Arrays

Condition based on **query operators** is satisfied when...

1. the given **field** as a **whole** **satisfies** all the involved operators, or
2. **at least one** item of the array **satisfies** each of the involved operators
   - note, however, that this item might **not** be the **same** for all the **individual operators**

```
db.movies.find( { ratings: { $gte: 2, $lte: 3 } } )
```

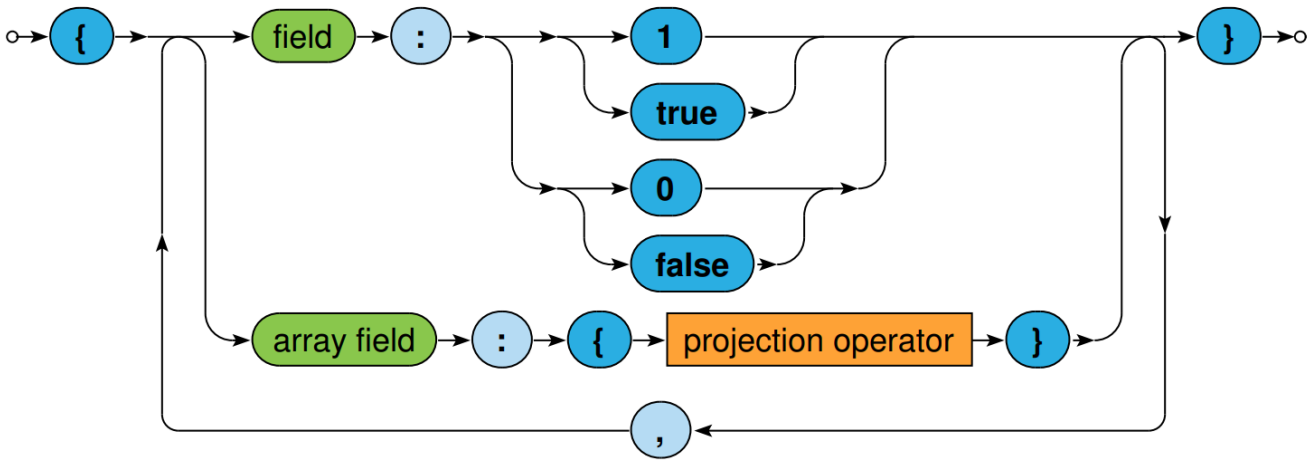
```
{ ratings: 3 }
```

```
{ ratings: [ 3, 7, 5 ] }
```

Use $elemMatch when just one array item should be found for all the operators

Projection

Projection allows us to determine the **fields returned** in the result

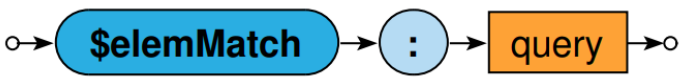


- **true or 1** for fields to be **included**
- **false or 0** for fields to be **excluded**
- **Positive** and **negative** enumerations **cannot** be **combined!**
  - The only exception is _id which is included by default
- **Projection operators** – allow to **select** particular array items

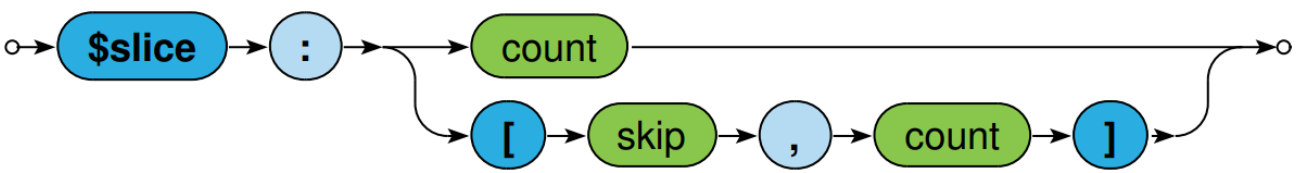
Projection Operators

Array operators

1. **$elemMatch** – selects the **first matching item** of an array
   - This item must **satisfy all** the operators included in query
   - When there is **no such item**, the field is **not returned** at all



2. **$slice** – selects the **first count items** of an array (when count is **positive**)/the **last count items** (when **negative**)
   - Certain number of items can also be skipped



Projection: Examples

Find a particular movie, select its identifier, title and actors

```
db.movies.find(
  { _id: ObjectId("2") },
  { title: true, actors: true }
)
```

```
{
  _id: ObjectId("2"),
  title: "Zombieland",
  actors: [   ObjectId("6"),
        ObjectId("4"),
        ObjectId("5") ]
}
```

Find movies from 2000, select their titles and the last two actors

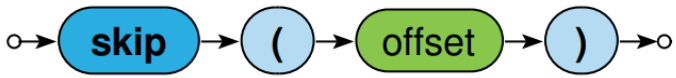
```
db.movies.find(
  { year: 2000 },
  {
    title: 1, _id: 0,
    actors: { $slice: -2 }
  }
)
```

```
{
  title: "Zombieland",
  actors: [   ObjectId("4"),
        ObjectId("5") ]
}
```

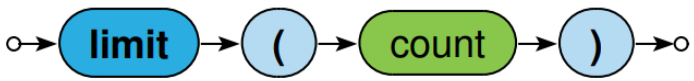
Modifiers

Modifiers change the **order** and **number** of returned documents

1. **sort** – orders the documents in the result
2. **limit** – returns at most a certain number of documents



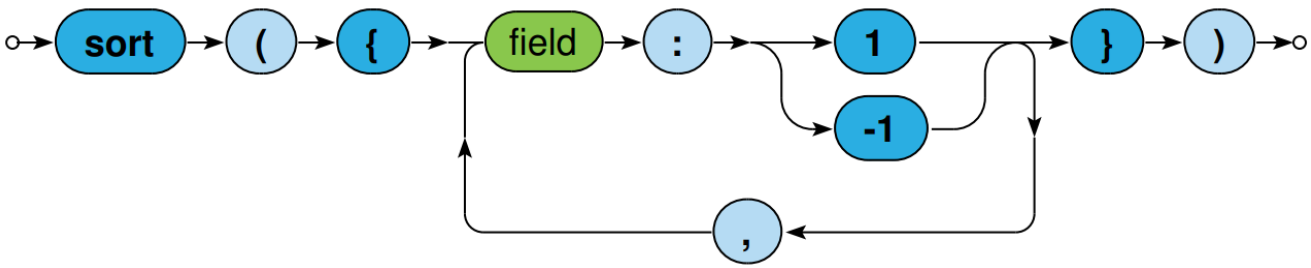
3. **skip** – skips a certain number of documents from the beginning



All the modifiers are **optional**, can be chained in **any order**, but must all be specified **before** any documents are retrieved via a given cursor

Modifiers

Sort modifier orders the documents in the result



- 1 for **ascending**, -1 for **descending** order
- The **order** of documents is **undefined** unless **explicitly sorted**
- Sorting of **larger datasets** should be supported by **indices**
- Sorting happens **before** the **projection phase**
  - i.e. **not included fields** can be used for sorting purposes as well

MongoDB Features

- Document-Oriented storage
- Full Index Support
- Replication High Availability
- Auto-Sharding
- Querying
- Fast In-Place Updates
- Map/Reduce functionality

Lecture Conclusion

MongoDB

- Document database for JSON documents
- Sharding with master-slave replication architecture

Query functionality

- CRUD operations
  - Insert, find, update, remove
  - Complex filtering conditions
- MapReduce
- Index structures

Index Functionality

- B+ tree indexes
- An index is automatically created on the _id field (the primary key)
- Users can create other indexes to improve query performance or to enforce Unique values for a particular field
- Supports single field index as well as Compound index
  - Like SQL order of the fields in a compound index matters
  - If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array
- Sparse property of an index ensures that the index only contain entries for documents that have the indexed field. (so ignore records that do not have the field defined)
- If an index is both unique and sparse – then the system will reject records that have a duplicate key value but allow records that do not have the indexed field defined

Aggregated functionality

Aggregation framework provides SQL-like aggregation functionality

- Pipeline documents from a collection pass through an aggregation pipeline, which transforms these objects as they pass through
- Expressions produce output documents based on calculations performed on input documents

```
db.parts.aggregate(
  {$group : {
    _id: type, totalquantity: { $sum: quanity}
    }
  }
)
```

Map reduce functionality

- Performs complex aggregator functions given a collection of keys, value pairs
- Must provide at least a map function, reduction function and a name of the result set

```
db.collection.mapReduce(
  <mapfunction>, <reducefunction>, {
    out: <collection>,
    query: <document>,
    sort: <document>,
    limit: <number>,
    finalize: <function>,
    scope: <document>,
    jsMode: <boolean>,
    verbose: <boolean>
  }
)
```

Indexes: High performance read

- Typically used for frequently used queries
- Necessary when the total size of the documents exceeds the amount of available RAM.
- Defined on the collection level
  - Can be defined on 1 or more fields
  - Composite index (SQL) $\Rightarrow$ Compound index (MongoDB)
- B-tree index
- Only 1 index can be used by the query optimizer when retrieving data
- Index covers a query - match the query conditions and return the results using only the index;
  - Use index to provide the results.

Replication of data

Ensures redundancy, backup, and automatic failover

- Recovery manager in the RDMS

Replication occurs through groups of servers known as replica sets

- Primary set – set of servers that client tasks direct updates to
- Secondary set – set of servers used for duplication of data
- At the most can have 12 replica sets
  - Many different properties can be associated with a secondary set i.e. secondary-only, hidden delayed, arbiters, non-voting
- If the primary set fails the secondary sets ′vote′ to elect the new primary set

Consistency of data

All read operations issued to the primary of a replica set are consistent with the last write operation

- Reads to a primary have strict consistency
  - Reads reflect the latest changes to the data
- Reads to a secondary have eventual consistency
  - Updates propagate gradually
- If clients permit reads from secondary sets – then client may read a previous state of the database
- Failure occurs before the secondary nodes are updated
  - System identifies when a rollback needs to occur
  - Users are responsible for manually applying rollback changes

Provides Memory Mapped Files

- A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource.
- mmap()

Other additional features

Supports geospatial data of type

- Spherical
  - Provides longitude and latitude
- Flat
  - 2 dimensional points on a plane
- Geospatial indexes