NDBI006: **Query Languages II**

http://www.ksi.mff.cuni.cz/~svoboda/courses/212-NDBI006/

Lecture 10

MongoDB

Martin Svoboda

martin.svoboda@matfyz.cuni.cz

19. 4. 2022

Charles University, Faculty of Mathematics and Physics

Lecture Outline

Document databases

- Introduction

Data formats

- JSON, BSON

MongoDB

- Data model
- CRUD operations
  - **Insert**, **update**, **save**, **remove**
  - **Find**: projection, selection, modifiers
- Index structures
- MapReduce

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

JSON

JavaScript Object Notation

Introduction

JSON = *JavaScript Object Notation*

- **Open standard for data interchange**
- Design goals
  - **Simplicity**: text-based, easy to read and write
  - **Universality**: object and array data structures
    - Supported by majority of modern programming languages
    - Based conventions of the C-family of languages (C, C++, C#, Java, JavaScript, Perl, Python, …)
- Derived from JavaScript (but language independent)
- Started in 2002
- File extension: ***.json**
- Content type: `application/json`
- http://www.json.org/

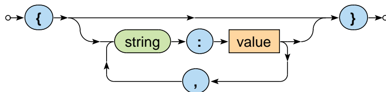
Example

```
{
  "title" : "Medvídek",
  "year" : 2007,
  "actors" : [
    {
      "firstname" : "Jiří",
      "lastname" : "Macháček"
    },
    {
      "firstname" : "Ivan",
      "lastname" : "Trojan"
    }
  ],
  "director" : {
    "firstname" : "Jan",
    "lastname" : "Hřebejk"
  }
}
```

Data Structure

Object

- **Unordered collection of name-value pairs** (properties)
  - Correspond to structures such as objects, records, structs, dictionaries, hash tables, keyed lists, associative arrays, …
- Values can be of different types, **names should be unique**



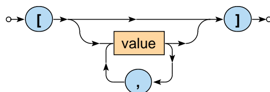
Examples

- `{ "name" : "Ivan Trojan", "year" : 1964 }`
- `{ }`

Data Structure

Array

- **Ordered collection of values**
  - Correspond to structures such as arrays, vectors, lists, sequences, …
- Values can be of different types, duplicate values are allowed



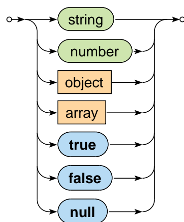
Examples

- `[ 2, 7, 7, 5 ]`
- `[ "Ivan Trojan", 1964, -5.6 ]`
- `[ ]`

Data Structure

Value

- Unicode string
  - Enclosed with double quotes
  - Backslash escaping sequences
  - Example: `"a \n b \" c \\ d"`
- Number
  - Decimal integers or floats
  - Examples: `1`, `-0.5`, `1.5e3`
- **Nested object**
- **Nested array**
- Boolean value: `true`, `false`
- Missing information: `null`



JSON Conclusion

JSON constructs

- Collections: object, array
- Scalar values: string, number, boolean, null

Schema languages

- JSON Schema

Query languages

- JSONiq, JMESPath, JAQL, …

BSON

Binary JSON

Introduction

BSON = *Binary JSON*

- **Binary-encoded serialization of JSON documents**
  - Extends the set of basic data types of values offered by JSON (such as a string, …) with a few new specific ones
- Design characteristics: **lightweight**, **traversable**, **efficient**
- Used by **MongoDB**
  - Document NoSQL database for JSON documents
  - Data storage and network transfer format
- File extension: ***.bson**
- http://bsonspec.org/

Example

JSON

```
{ "title" : "Medvídek", "year" : 2007 }
```

BSON

```
24 00 00 00 02 74 69 74 6C 65 00 0A 00 00 00 4D 65 64 76 C3 AD 64 65 6B
00 10 79 65 61 72 00 D7 07 00 00 00
```

Example

JSON

```
{
  "title" : "Medvídek",
  "year" : 2007
}
```

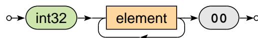
BSON

```
24 00 00 00
02 74 69 74 6C 65 00 0A 00 00 00 4D 65 64 76 C3 AD 64 65 6B 00
10 79 65 61 72 00 D7 07 00 00
00
```

Document Structure

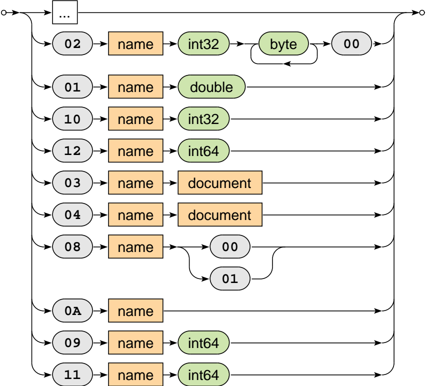
Document = serialization of **one JSON object or array**

- JSON object is serialized directly
- JSON array is first transformed to a JSON object
  - Property names derived from numbers of positions
  - E.g.:
    `[ "Trojan", "Svěrák" ]` →
    `{ "0" : "Trojan", "1" : "Svěrák" }`
- Structure
  - **Document size** (total number of bytes)
  - Sequence of **elements** (encoded JSON properties)
  - Terminating hexadecimal 00 byte



Document Structure

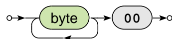
Element = serialization of **one JSON property**



Document Structure

Element = serialization of **one JSON property**

- Structure
  - **Type** selector
    - 02 (**string**)
    - 01 (double), 10 (32-bit **integer**), 12 (64-bit integer)
    - 03 (**object**), 04 (**array**)
    - 08 (**boolean**)
    - 0A (**null**)
    - 09 (datetime), 11 (timestamp)
    - …
  - Property **name**
    - Unicode string terminated by 00



  - Property **value**

MongoDB Document Database

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: "Vratné lahve",
  year: 2006
}
```

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

```
{
  _id: ObjectId("3"),
  title: "Medvídek",
  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: "Vratné lahve" }
```

```
{ title: "Medvídek" }
```

Data Model

Database system structure

Instance $\rightarrow$ **databases** $\rightarrow$ **collections** $\rightarrow$ **documents**

- Database
- Collection
  - Collection of documents, usually of a similar structure
- 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 fields

- **Top-level `_id`** is reserved for a **primary key**
- Field names **cannot start with** `$` and **cannot contain** `.`
  - `$` is reserved for query operators
  - `.` is used when accessing nested fields
- The order of fields is preserved
  - Except for `_id` fields that are always moved to the beginning
- **Names of fields must be unique**

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

- Natural identifier
- Auto-incrementing number – not recommended
- UUID (*Universally Unique Identifier*)
- **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

- References
- Embedded documents

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

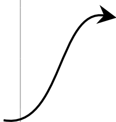
```
{
  _id: ObjectId("2"), title: "Samotáři", year: 2000,
  actors: [
    { firstname: "Jitka", lastname: "Schneiderová" },
    { firstname: "Ivan", lastname: "Trojan" },
    { firstname: "Jiří", lastname: "Macháček" }
  ]
}
```

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: "Samotáři",
  year: 2000,
  actors: [ ObjectId("6"),
            ObjectId("4"),
            ObjectId("5") ]
}
```



```
{
  _id: ObjectId("6"),
  firstname: "Jitka",
  lastname: "Schneiderová"
}
```

...

Sample Data

Collection of **movies**

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

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

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

Collection of **actors**

```
{ _id: ObjectId("4"),
  firstname: "Ivan",
  lastname: "Trojan" }
```

```
{ _id: ObjectId("5"),
  firstname: "Jiří",
  lastname: "Macháček" }
```

```
{ _id: ObjectId("6"),
  firstname: "Jitka",
  lastname: "Schneiderová" }
```

```
{ _id: ObjectId("7"),
  firstname: "Zdeněk",
  lastname: "Svěrák" }
```

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

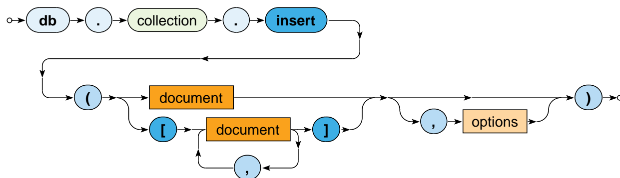
Overview

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

Insert Operation

Insert Operation

Inserts a new document / documents into a given collection



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

Insert Operation: Examples

Insert a new actor document

```
db.actors.insert(
  {
    firstname: "Anna",
    lastname: "Geislerová"
  }
)
```

```
{
  _id: ObjectId("8"),
  firstname: "Anna",
  lastname: "Geislerová"
}
```

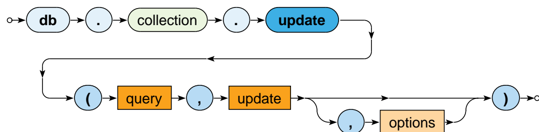
Insert two new movies

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

Update Operation

Update Operation

Modifies / replaces an existing document / documents



- Parameters
  - **Query**: description of documents to be updated
    - The same behavior as in find operations
  - **Update**: modification actions to be applied
  - **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: "Aňa",
    lastname: "Geislerová" }
)
```

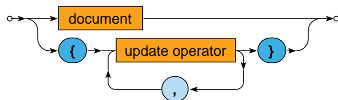
```
{
  _id: ObjectId("8"),
  firstname: "Aňa",
  lastname: "Geislerová"
}
```

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

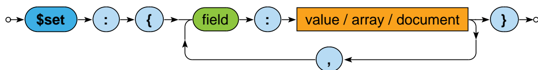


- **Replace**
  when the `update` parameter contains no update operators
  - **The whole document is replaced** (`_id` is preserved)
- **Update**
  when the `update` parameter contains only **update operators**
  - **Current document is updated** using these operators
    - `$set`, `$unset`, `$inc`,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

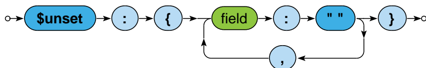
Update Operators

Field operators

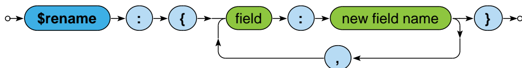
- `$set` – sets the value of a given field / fields



- `$unset` – removes a given field / fields



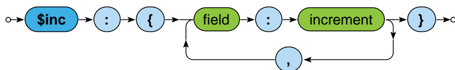
- `$rename` – renames a given field / fields



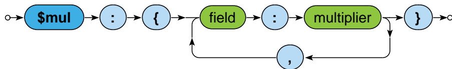
Update Operators

Field operators

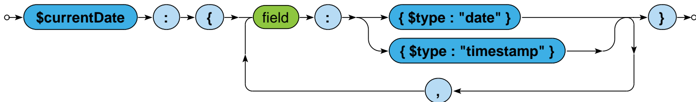
- `$inc` – increments the value of a given field / fields



- `$mul` – multiplies the value of a given field / fields



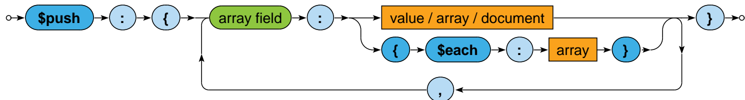
- `$currentDate` – stores the current date time / timestamp to a given field / fields



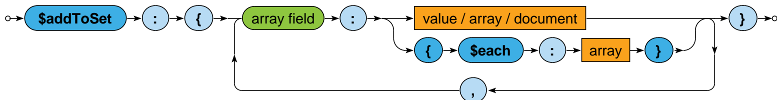
Update Operators

Array operators

- `$push` – adds one item / all items to the end of an array



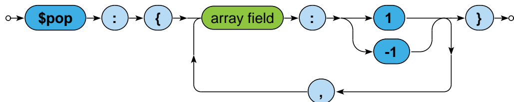
- `$addToSet` – adds one item / all items to the end of an array, but duplicate values are ignored



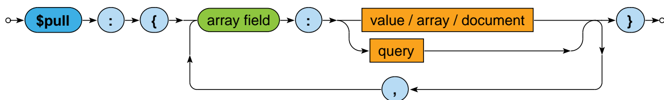
Update Operators

Array operators

- `$pop` – removes the first / last item of an array



- `$pull` – removes all array items that match a specified query



Upsert Mode

Upsert behavior of update operation

- When `{ upsert: true }` option is specified,
  and, at the same time, **no document was updated**
  $\Rightarrow$ **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 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

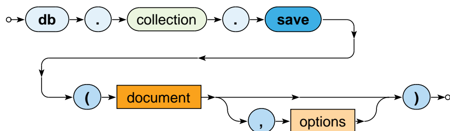
```
db.movies.update(
  { title: "Tmavomodrý svět", year: { $gt: 2000 } },
  {
    $set: {
      director: { firstname: "Jan", lastname: "Svěrák" },
      year: 2001
    },
    $inc: { rating: 2 }
  },
  { upsert: true }
)
```

```
{ _id: ObjectId("11"),
  title: "Tmavomodrý svět",
  director: { firstname: "Jan", lastname: "Svěrák" },
  year: 2001,
  rating: 2 }
```

Save Operation

Save Operation

Replaces an existing / inserts a new document



- Parameters
  - **Document**: document to be modified / inserted
  - **Options**

Save Operation

Insert mode

- Document identifier must not be specified in the query or must not yet exist in a given collection

```
db.actors.save({ name: "Tatiana Vilhelmova" })
```

```
db.actors.save({ _id: 6, name: "Sasa Rasilov" })
```

Update mode

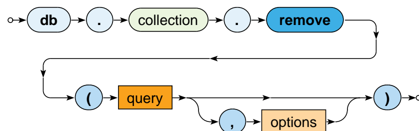
- Document identifier must be specified in the query and must exist in a given collection

```
db.actors.save({ _id: "trojan", name: "Ivan Trojan", year: 1964 })
```

Remove Operation

Remove Operation

Removes a document / documents from a given collection

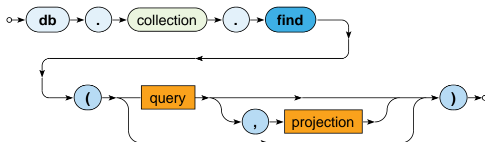


- Parameters
  - **Query**: description of documents to be removed
    - The same behavior as in find operations
  - **Options**
- All the matching documents are removed unless `{ justOne: true }` option is provided

Find Operation

Find Operation

Selects documents from a given collection



- Parameters
  - **Query**: description of documents to be selected
  - **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*

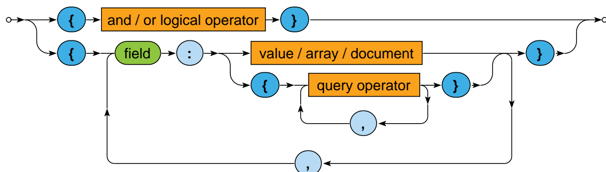
```
db.movies.find( { year: 2000, rating: { $gt: 1 } } )
```

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



Boolean expression with <u>one</u> top-level logical operator: `$and`, `$or`

Conditions on individual <u>distinct</u> fields

- **Value equality**
  - The actual field value must be identical to the specified value
- **Query operators**
  - The actual field value must satisfy all the provided operators

Selection: Field Conditions

Value equality

- The actual field value must be identical to the specified value
- I.e. identical…
  - including the number, order and names of recursively identical values of all nested **object fields**
  - including the number and order of recursively identical **array items**

Query operators

- The actual field value must satisfy all the provided operators
  - Each operator can be used at most once

Value Equality: Examples

Select movies having a specific director

```
db.movies.find(
  { director: { firstname: "Jan", lastname: "Svěrák" } }
)
```

```
db.movies.find(
  { director: { lastname: "Svěrák", firstname: "Jan" } }
)
```

Select movies having specific actors

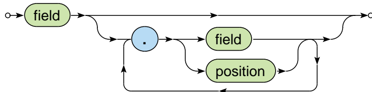
```
db.movies.find( { actors: [ ObjectId("7"), ObjectId("5") ] } )
```

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

Queries in both the pairs are not equivalent!

Dot Notation

The **dot notation** for field names



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

Value Equality

Example (revisited)

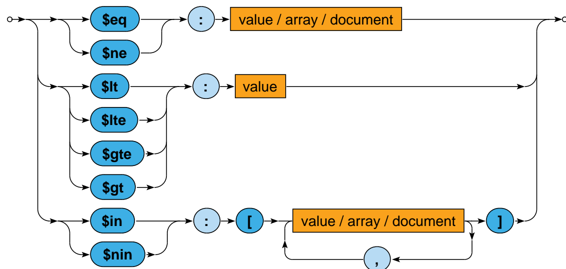
Select movies having a specific director

```
db.movies.find(
  { director: { firstname: "Jan", lastname: "Svěrák" } }
)
```

```
db.movies.find(
  { "director.firstname": "Jan", "director.lastname": "Svěrák" }
)
```

Query Operators

Comparison operators



- Comparisons take particular BSON data types into account
  - Certain numeric conversions are automatically applied

Query Operators

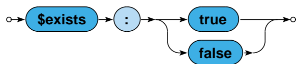
Comparison operators

- `$eq`, `$ne`
  - Tests the actual field value for **equality** / **inequality**
    - The same behavior as in case of value equality conditions
- `$lt`, `$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
- `$in`
  - Tests whether the actual field value is equal to **at least one** of the provided values
- `$nin`
  - Negation of `$in`

Query Operators

Element operators

- `$exists` – tests whether a given field **exists** / **not exists**



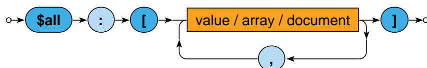
Evaluation operators

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

Query Operators

Array operators

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



Example (revisited)

Select movies having specific actors

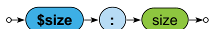
```
db.movies.find(
  { actors: [ ObjectId("5"), ObjectId("7") ] }
)
```

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

Query Operators

Array operators

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

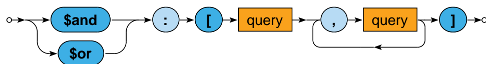


- `$elemMatch` – tests whether a given array **contains at least one item** that satisfies all the involved query operations

Query Operators

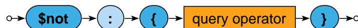
Logical operators

- `$and`, `$or`



  - Logical connectives for **conjunction** / **disjunction**
  - At least 2 involved query expressions must be provided
  - **Only allowed at the top level** of a query

- `$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...

- the given field as a whole is identical to the provided value, or
- 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...

- the given field as a whole satisfies all the involved operators, or
- each of the involved operators is satisfied by at least one item of the given array
  - note, however, that this item **may not be the same** for all the individual operators

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

```
{ ratings: 3 }
```

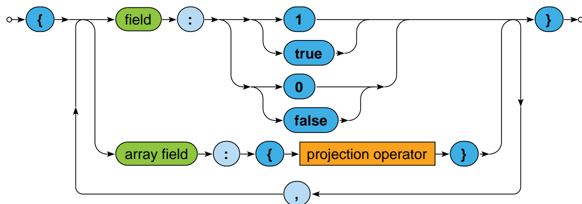
```
{ ratings: [ 3, 7, 5 ] }
```

```
{ ratings: [ 1, 4 ] }
```

Use `$elemMatch` when just a single array item should be found for all the operators

Projection

Projection allows us to determine the fields returned in the result

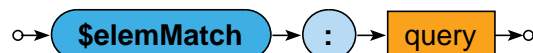


- `true` or `1` for fields to be **included**
- `false` or `0` for fields to be **excluded**
- Positive and negative enumerations cannot be combined!
  - The only exception is `_id` which is **included by default**
- **Projection operators** – allow to select particular array items

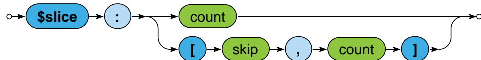
Projection Operators

Array operators

- `$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



- `$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: "Samotáři",
  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 }
  }
)
```

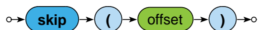
```
{
  title: "Samotáři",
  actors: [ ObjectId("4"),
            ObjectId("5") ]
}
```

Modifiers

Modifiers change the order and number of returned documents

- `sort` – orders the documents in the result
- `limit` – returns at most a certain number of documents

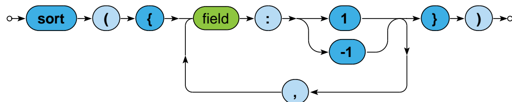
- `skip` – skips a certain number of documents from the beginning



All the modifiers are optional, can be chained in **any order** (without any implications), 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

Index Structures

Index Structures

Motivation

- Full **collection scan** must be conducted when searching for documents **unless an appropriate index exists**

Primary index

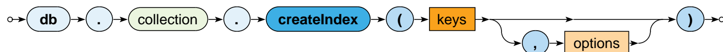
- Unique index on values of the **_id field**
- Created automatically

Secondary indexes

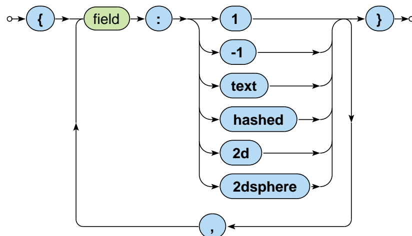
- Created manually for values of a given key field / fields
- Always within just a single collection

Index Structures

Secondary **index creation**



Definition of keys (fields) to be involved



Index Structures

Index types

- `1`, `-1` – standard ascending / descending value indexes
  - Both scalar values and embedded documents can be indexed
- `hashed` – hash values of a single field are indexed
- `text` – basic full-text index
- `2d` – points in planar geometry
- `2dsphere` – points in spherical geometry

Index Structures

Index forms

- One key / multiple keys (**composed index**)
- Ordinary fields / array fields (**multi-key index**)

Index properties

- **Unique** – duplicate values are rejected (cannot be inserted)
- **Partial** – only certain documents are indexed
- **Sparse** – documents without a given field are ignored
- **TTL** – documents are removed when a timeout elapses

Just some type / form / property combinations can be used!

Index Structures

Execution plan

```
db.actors.find({ movies: "medvidek" }).explain()
```

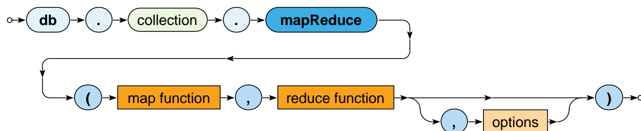
Index creation

```
db.actors.createIndex({ movies: 1 })
```

MapReduce

MapReduce

Executes a **MapReduce** job on a selected collection



- Parameters
  - **Map**: JavaScript implementation of the Map function
  - **Reduce**: JavaScript implementation of the Reduce function
  - **Options**

MapReduce

Map function

- Current document is accessible via `this`
- `emit(key, value)` is used for emissions

Reduce function

- Intermediate key and values are provided as arguments
- Reduced value is published via `return`

Options

- `query`: only matching documents are considered
- `sort`: they are processed in a specific order
- `limit`: at most a given number of them is processed
- `out`: output is stored into a given collection

MapReduce: Example

Count the number of movies filmed in each year, starting in *2005*

```
db.movies.mapReduce(
  function() {
    emit(this.year, 1);
  },
  function(key, values) {
    return Array.sum(values);
  },
  {
    query: { year: { $gte: 2005 } },
    sort: { year: 1 },
    out: "statistics"
  }
)
```

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
- Index structures
- MapReduce