

B4M36DS2: Database Systems 2

<http://www.ksi.mff.cuni.cz/~svoboda/courses/2016-1-B4M36DS2/>

Lecture 6

Document Databases: MongoDB

Martin Svoboda

svoboda@ksi.mff.cuni.cz

7. 11. 2016

Charles University in Prague, Faculty of Mathematics and Physics

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture Outline

Document databases

- General introduction

MongoDB

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

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!

Document Stores

Suitable use cases

- Event logging, content management systems, blogs, web analytics, e-commerce applications, ...
 - I.e. **for structured documents with similar schema**

When not to use

- **Set operations** involving multiple documents
- Design of document structure is constantly changing
 - I.e. when the required level of granularity would outbalance the advantages of aggregates

Document Stores

Representatives

- **MongoDB, Couchbase, Amazon DynamoDB, CouchDB, RethinkDB, RavenDB, Terrastore**
- *Multi-model*: **MarkLogic, OrientDB**, OpenLink Virtuoso, ArangoDB

Document Stores

Representatives



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

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 → **databases** → **collections** → **documents**

- Database
- Collection
 - Collection of documents, usually of a similar structure
- Document
 - MongoDB **document** = **one JSON object**
 - Each document...
 - belongs to exactly one collection
 - has a **unique identifier** `_id`

Data Model

MongoDB **document** = **one JSON object**

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

Restrictions on field names

- _id (at the top level) is reserved for a **primary key**
- Field names **cannot start with \$**
 - Reserved for query operators
- Field names **cannot contain .**
 - Used when accessing nested fields

Data Model

Primary Keys

Features of identifiers

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

Design of identifiers

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

Data Model

Design Questions

Flexible schema

- No document schema is provided, nor expected or enforced
 - However, **documents within a collection are similar in practice**
- MongoDB document = one **JSON** object
 - I.e. even a complex JSON object with other recursively nested objects, arrays or values

Design challenge

- Balancing application requirements, performance aspects, and data retrieval patterns,
- while considering structure of data and mutual relationships

Two main concepts: **references** vs. **embedded documents**

Data Model

Design Questions: Denormalized Data Models

Embedded documents

- **Related data in a single structure** with **subdocuments**
 - Suitable for **one-to-one** or **one-to-many** relationships
- Brings ability to read / write related data in a single operation
 - I.e. better performance, less queries need to be issued

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

Data Model

Design Questions: Normalized Data Models

References

- **Directed links between documents**, expressed via identifiers
 - Idea analogous to foreign keys in relational databases
 - Suitable for **many-to-many** relationships
 - Embedding in this case would result in data duplication
- References provide more flexibility than embedding
 - Follow up queries might be needed, however

```
{
  _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 JavaScript 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

Single JavaScript command / complex script

- Each individual command is evaluated over exactly one collection
- Read queries return a **cursor**
 - Allows us to iterate over all the selected documents

Query patterns

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

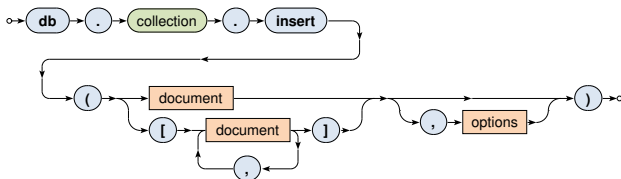
CRUD Operations

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

Inserts a new document / documents into a given collection



- **Document identifier** (`_id` field)
 - The provided value must be unique within the collection
 - When missing, it is generated automatically (**ObjectId**)
- The collection is created automatically when not yet exists

Insert Operation

Examples

Insert a new actor document

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

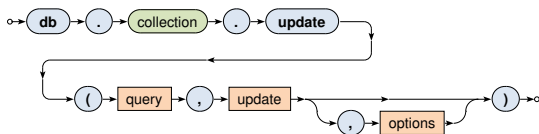
```
{  
  _id: ObjectId("8"),  
  firstname: "Anna",  
  lastname: "Geislerová"  
}
```

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

Modifies / replaces an existing document / documents



- Parameters
 - **Query:** description of documents to be updated
 - **Update:** modification actions to be applied
- **Just 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: "Aña",  
    lastname: "Geislerová" }  
)
```

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

Update all the movies filmed in 2015 or later

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

Update Operation

Update / replace behavior

- **Replace**

when the update parameter contains no update operators

- \$set, \$unset, \$inc, \$mul, ...

- **Update**

otherwise

- It must then only have update operators and no value fields!
- I.e. mutual combinations of the both are not allowed

Update Operation: Upsert

Upsert behavior

- **New single document is inserted** when
 - `{ upsert : true }` option is specified, and, at the same time, no document was updated
- The new document contains
 - *when replacement takes place*
all the (value) fields from the update parameter, or
 - *otherwise*
all the value fields from the query parameter (i.e. **comparison operations** are omitted) + the outcome of all the update operations from the update parameter,
 - and a newly generated `_id` if necessary

Update Operation: Upsert

Examples

Unsuccessful update of a movie resulting to an insertion

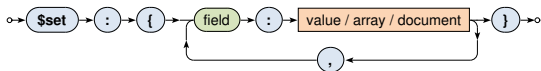
```
db.movies.update(  
  { title: "Tmavomodrý svět", year: { $gt: 2000 } },  
  {  
    $set: {  
      director: { firstname: "Jan", lastname: "Svěrák" },  
      year: 2001  
    },  
    $inc: { rating: 2 }  
  },  
  { upsert: true }  
)
```

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

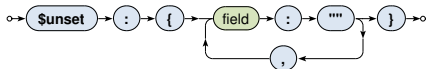
Update Operation: Operators

Field operators

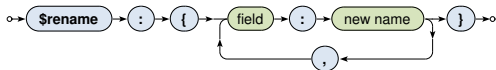
- **\$set** – sets the value of a given field / fields



- **\$unset** – removes a specified field



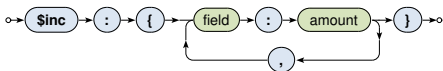
- **\$rename** – renames a field



Update Operation: Operators

Field operators

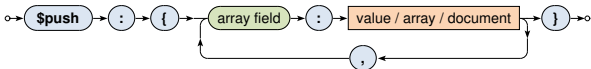
- `$inc` – increments the value of a field by the specified amount



- `$mul` – multiplies the value of a field by the specified amount
- ...

Array operators

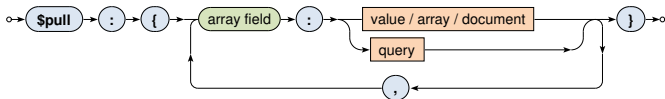
- `$push` – adds one item at the end of an array



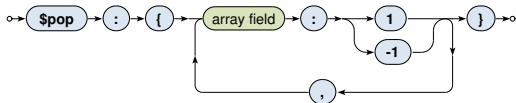
Update Operation: Operators

Array operators

- `$addToSet` – adds a value to an array unless already present
- `$pull` – removes all array items that match a specified query



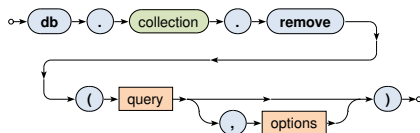
- `$pop` – removes the first / last item of an array



- ...

Remove Operation

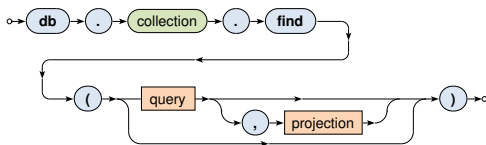
Removes a document / documents from a given collection



- **Query** parameter describes documents to be removed
- All the matching documents are removed unless `{ justOne: true }` option is provided in options

Find Operation

Selects documents from a given collection



- **Query** parameter describes documents to be selected
- **Projection** parameter enumerates fields to be included / excluded in / from 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 the 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 all the movies from year *2000* with a rating greater than *1*

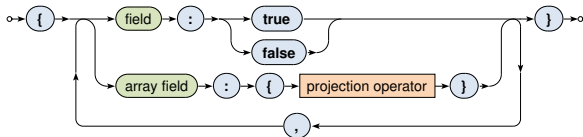
```
db.movies.find( { year: 2000, rating: { $gt: 1 } } )
```

Select all the movies filmed between *2005* and *2015*

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


Find Operation: Projection

Projection allows us to determine the fields returned in the result



- **true** or 1 for the fields to be **included**
- **false** or 0 for the fields to be **excluded**
- Positive and negative enumerations cannot be combined!
 - The only exception is `_id` which is **included by default**
- **Projection operator**
 - Allows us to select particular items from an array
 - `$elemMatch`, `$slice`, ...

Find Operation: Projection

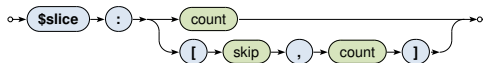
Projection Operators

Projection operators for array fields

- `$elemMatch` – selects the first matching item of an array
This item must satisfy all the operations included in query
When no matching item is found, 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



- ...

Find Operation: Projection

Examples

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

```
db.movies.find(  
  { _id: ObjectId("2") },  
  { title: true, actors: 1 }  
)
```

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

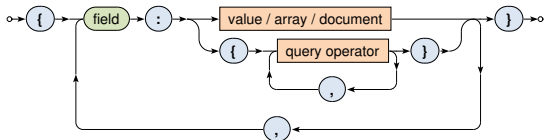
Find all the movies from 2000, select a title and the last two actors

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

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

Find Operation: Selection

Query parameter describes the documents we are interested in



- Condition based on **value equality**
 - The actual field value must be identical to the specified value (including, e.g., the order of nested fields or array items)
- Condition based on **query operators**
 - The actual field value must satisfy all the provided operations

Find Operation: Selection

Value Equality Conditions: Examples

Select all the movies having a specific director

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

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

Select all the movies having specific actors

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

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

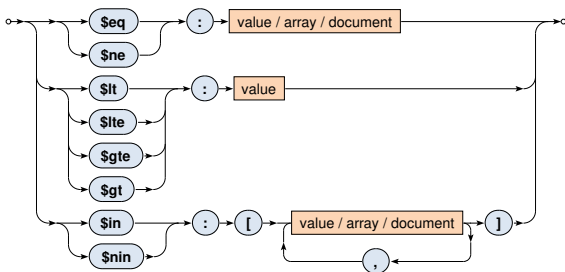
Queries in both the pairs are not equivalent!

Solution: **dot notation** and **\$all** operator respectively (see later)

Find Operation: Selection

Query Operators

Comparison operators



- Comparisons take particular **BSON** data types into account
- Certain numeric conversions might be automatically applied

Find Operation: Selection

Query Operators

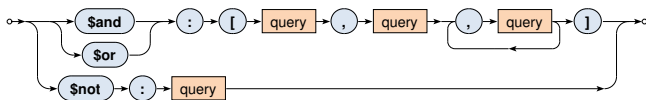
Comparison operators

- `$eq`, `$ne`
 - Tests the actual field value for **equality / inequality**
- `$lt`, `$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`

Find Operation: Selection

Query Operators

Logical operators



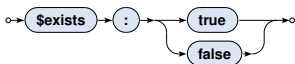
- `$and`, `$or`
 - Logical connectives for **conjunction** / **disjunction**
 - At least 2 involved query expressions must be provided
- `$not`
 - Logical **negation** of right 1 involved query expression
- ...

Find Operation: Selection

Query Operators

Element operators

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



- ...

Evaluation operators

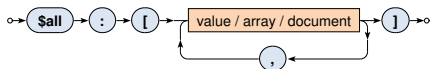
- `$regex` – tests whether the field value matches a **regular expression** (PCRE)
- `$text` – performs **text search** (text index must exist)
- ...

Find Operation: Selection

Query Operators

Array operators

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



- **\$size** – tests the size of a given array against a particular 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

Find Operation: Selection

Dot Notation

The **dot notation** is used when...

- **accessing fields of embedded documents**
 - `"field.subfield"`
 - E.g.: `"director.firstname"`
- **accessing items of arrays**
 - `"field.index"` – used in query selection or projection in order to **access particular array items**, positions start at 0
 - E.g.: `"actors.2"`
 - `"field.$"` – used in query projection in order to **access the very first array item that satisfies the query condition**
 - E.g.: `"actors.$"`

Find Operation: Selection

Value Equality Conditions: Examples Revisited

Select all the movies having a specific director

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

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

Select all the movies having specific actors

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

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

Find Operation: Selection

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") ] }
```

Find Operation: Selection

Querying Arrays

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

- the given field as a whole satisfies all the involved operations, or
- each of the involved operations is satisfied by at least one item of the given array, but this item might not be the same for all the individual operations

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

```
{ ratings: 3 }
```

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

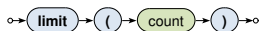
```
{ ratings: [ 1, 4 ] }
```

Note: use `$elemMatch` when exactly one witnessing item should be found for all the operations

Find Operation: Modifiers

Modifiers change the order and number of returned documents

- **sort** – orders the documents in the result
- **limit** – returns at most the specified number of documents



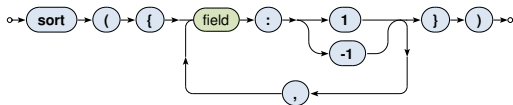
- **skip** – skips the specified number of documents from the beginning



All the modifiers are optional, can be chained in **any mutual order**, but **must be specified before retrieving any documents** via the cursor

Find Operation: Modifiers

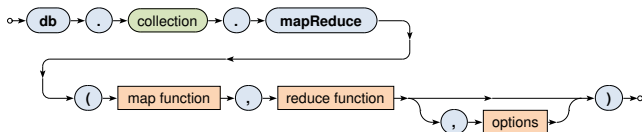
Sort modifier orders the documents in the result



- 1 for **ascending**, -1 for **descending** order
- Multiple fields can be used
- 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. excluded fields can be used for sorting purposes as well

MapReduce

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



- Both map and reduce functions are implemented as ordinary JavaScript functions
 - **Map** function: current document is accessible via `this`, `emit(key, value)` is used for emissions
 - **Reduce** function: key and array of values are provided as arguments, reduced value is published via `return`
- Beside others, query, sort or limit options are accepted
- out option determines the output (e.g. a collection name)

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"  
  }  
)
```

Index Structures

Motivation

- Full **collection scan** must be performed when searching for the documents matching conditions of a given query, at least **unless an appropriate index exists**

Primary index

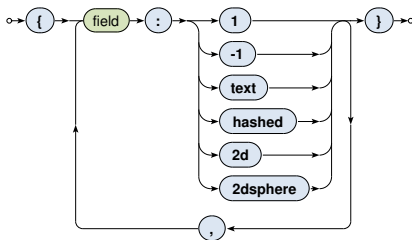
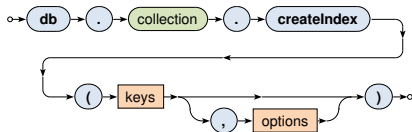
- Unique **index on the `_id` field**, created automatically

Secondary indexes

- Created manually for a given key field / fields, always on a particular collection

Index Structures

Secondary index creation



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

- Single field / **composed index** for multiple fields
- Single value / **multi-key index** for multiple values in arrays

Index properties

- **Unique** – duplicate values are rejected (cannot be inserted)
- **Partial** – only selected documents are indexed
- **Sparse** – only documents with the index field are indexed
- **TTL** – documents are removed when timeout elapses

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

Conclusion

MongoDB

- **JSON document database**
- **Sharding with master-slave replication architecture**

Query functionality

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