

## **MIE-PDB.16: Advanced Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/191-MIE-PDB/>

Lecture 11

# **Document Databases: MongoDB**

**Martin Svoboda**

[martin.svoboda@fit.cvut.cz](mailto:martin.svoboda@fit.cvut.cz)

3. 12. 2019

**Charles University**, Faculty of Mathematics and Physics

**Czech Technical University in Prague**, Faculty of Information Technology

# Lecture Outline

## Document databases

- Introduction

## MongoDB

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

# 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

# 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 → **databases** → **collections** → **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

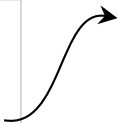
```
{
  _id: ObjectId("2"), title: "Samotáři", year: 2000,
  actors: [
    { firstname: "Jitka", lastname: "Schneiderová" },
    { firstname: "Ivan", lastname: "Trojan" },
    { firstname: "Jiří", lastname: "Macháček" }
  ]
}
```

# 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"),
  first: "Jitka",
  last: "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

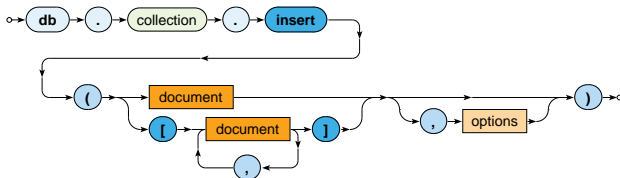
- `db.collection.insert()`
  - Inserts a new document into a collection
- `db.collection.update()`
  - Modifies an existing document / documents or inserts a new one
- `db.collection.remove()`
  - Deletes an existing document / documents
- `db.collection.find()`
  - Finds documents based on filtering 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"  
}
```

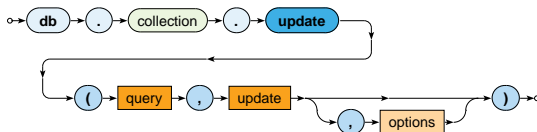
## Insert two new movies

```
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" }  
)
```

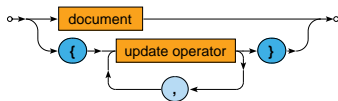
```
{  
  _id: ObjectId("8"),  
  firstname: "Aňa",  
  lastname: "Geislerová"  
}
```

**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

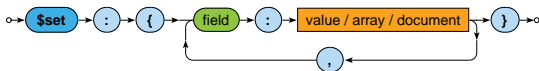


- **Replace**  
when the update parameter contains no update operators
  - **The whole document is replaced** (`_id` is preserved)
- **Update**  
when the update parameter contains only **update operators**
  - **Current document is updated** using these operators
    - `$set`, `$unset`, `$inc`, `$mul`,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

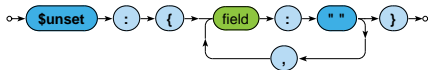
# Update Operators

## Field operators

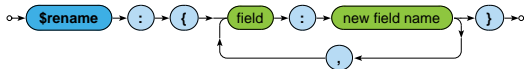
- **\$set** – sets the value of a given field / fields



- **\$unset** – removes a given field / fields



- **\$rename** – renames a given field / fields

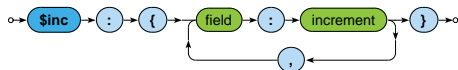




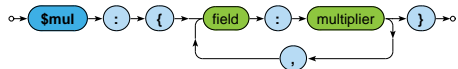
# Update Operators

## Field operators

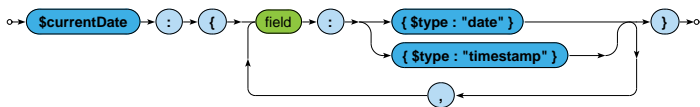
- `$inc` – increments the value of a given field / fields



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



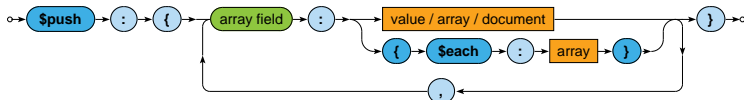
- `$currentDate` – stores the current date time / timestamp to a given field / fields



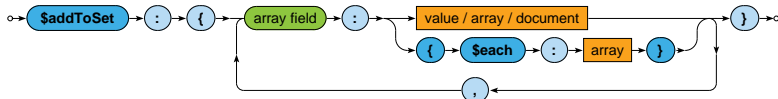
# Update Operators

## Array operators

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



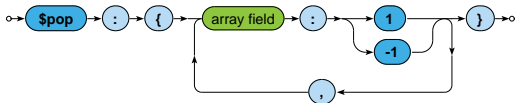
- `$addToSet` – adds one item / all items to the end of an array, but duplicate values are ignored



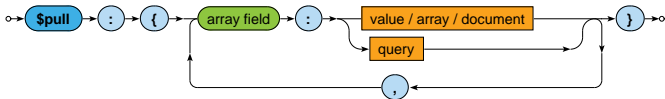
# Update Operators

## 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**  
⇒ **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

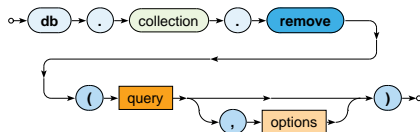
```
db.movies.update(  
  { title: "Tmavomodrý svět", year: { $gt: 2000 } },  
  {  
    $set: {  
      director: { firstname: "Jan", lastname: "Svěrák" },  
      year: 2001  
    },  
    $inc: { rating: 2 }  
  },  
  { upsert: true }  
)
```

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

# **Remove Operation**

# Remove Operation

**Removes** a document / documents from a given collection



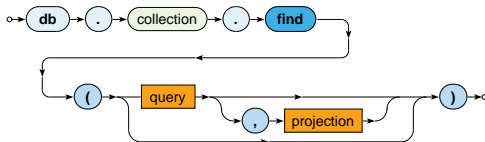
- Parameters
  - **Query:** description of documents to be removed
    - The same behavior as in find operations
  - **Options**
- All the matching documents are removed unless `{ justOne: true }` option is provided

# 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*

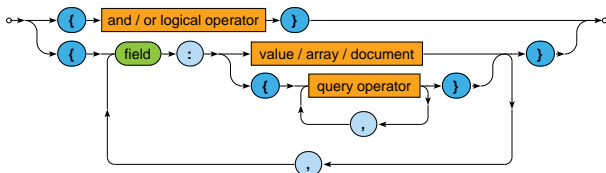
```
db.movies.find( { year: 2000, rating: { $gt: 1 } } )
```

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 one top-level logical operator: \$and, \$or  
Conditions on individual distinct 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

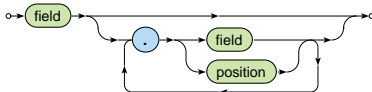
```
db.movies.find( { actors: [ ObjectId("7"), ObjectId("5") ] } )
```

```
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)

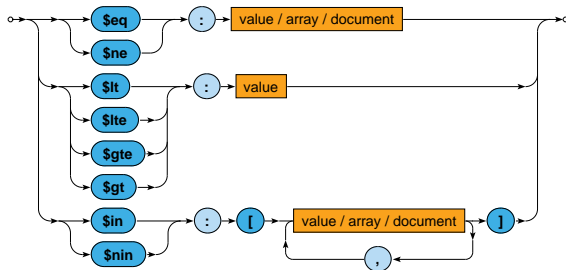
Select movies having a specific director

```
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

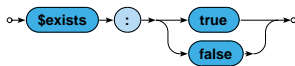
## Comparison operators

- `$eq`, `$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**



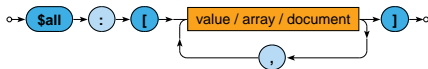
## Evaluation operators

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

# 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)



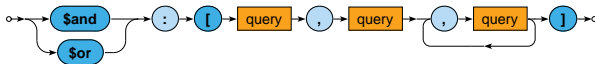
- `$elemMatch` – tests whether a given array **contains at least one item** that satisfies all the involved query operations



# 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 }
```

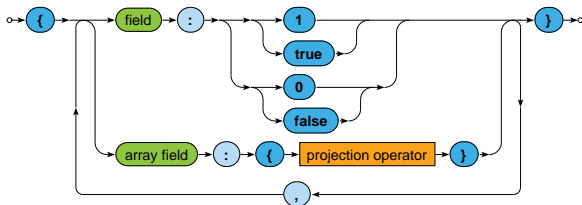
```
{ ratings: [ 3, 7, 5 ] }
```

```
{ 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



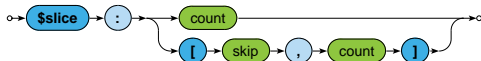
# Projection Operators

## 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



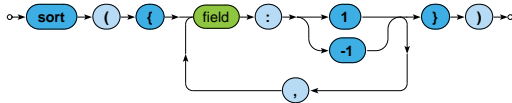
- **skip** – skips a certain number of documents from the beginning



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



# 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