Lecture 13:

**Document Databases, JSON, MongoDB**

17. 5. 2016

Authors: **Irena Holubová, Martin Svoboda**
Faculty of Mathematics and Physics, Charles University in Prague
Course NDBI040: **Big Data Management and NoSQL Databases**
Document Databases

Basic Characteristics

- Documents are the main concept
  - Stored and retrieved
  - XML, JSON, …

- Documents are
  - Self-describing
  - Hierarchical tree data structures
  - Can consist of maps, collections, scalar values, nested documents, …

- Documents in a collection are expected to be similar
  - Their schema can differ

- Document databases store documents in the value part of the key-value store
  - Key-value stores where the value is examinable
Document Databases

Suitable Use Cases

**Event Logging**
- Many different applications want to log events
  - Type of data being captured keeps changing
- Events can be sharded by the name of the application or type of event

**Content Management Systems, Blogging Platforms**
- Managing user comments, user registrations, profiles, web-facing documents, …

**Web Analytics or Real-Time Analytics**
- Parts of the document can be updated
- New metrics can be easily added without schema changes

**E-Commerce Applications**
- Flexible schema for products and orders
- Evolving data models without expensive data migration
Document Databases
When Not to Use

Complex Transactions Spanning Different Operations
- Atomic cross-document operations
  - Some document databases do support (e.g., RavenDB)

Queries against Varying Aggregate Structure
- Design of aggregate is constantly changing → we need to save the aggregates at the lowest level of granularity
  - i.e., to normalize the data
Document Databases
Representatives

- MongoDB
- CouchDB
- OrientDB
- RavenDB
- Lotus Notes Storage Facility
JSON

JavaScript Object Notation
Introduction

- **JSON** = JavaScript Object Notation
  - Text-based easy-to-read-and-write open standard for data interchange
    - Serializing and transmitting structured data
    - Design goals: *simplicity and universality*
  - Derived from JavaScript, but language independent
    - Uses conventions of the C-family of languages
      (C, C++, C#, Java, JavaScript, Perl, Python, ...)
  - Filename: *.*json
  - Media type: application/json
  - [http://www.json.org/](http://www.json.org/)
Example

```json
{
    "firstName" : "John",
    "lastName" : "Smith",
    "age" : 25,
    "address" : {
        "street" : "21 2nd Street",
        "city" : "New York",
        "state" : "NY",
        "postalCode" : 10021
    },
    "phoneNumbers" : [
        { "type" : "home", "number" : "212 555-1234" },
        { "type" : "fax", "number" : "646 555-4567" }
    ]
}
```
Data Structures

• Built on two general structures
  ▪ Object
    – Collection of name-value pairs
      • Realized as an object, record, struct, dictionary, hash table, keyed list, associative array, ...
  ▪ Array
    – List of values
      • Realized as an array, vector, list, sequence, ...
  ▪ All modern programming languages support them
Data Structures

- **Object**
  - **Unordered** set of name-value pairs
    - Called properties of an object

- **Examples**
  - `{ "name" : "Peter", "age" : 30 }`
  - `{ }`
Data Structures

• **Array**
  - **Ordered** collection of values
    - Called items or elements of an array

  ![Diagram of an array]

  - **Examples**
    - `[ 3, 5, 7, 9 ]`
    - `[ 15, "word", -5.6 ]`
    - `[ ]`
Values

- **Strings**
- **Numbers**
- **Nested objects or arrays**
- **Boolean values**
  - `true` and `false`
- **Null value**
  - Missing information
Values

• String
  ▪ Sequence of Unicode characters
    – Wrapped in double quotes
    – Backslash escaping sequences for special characters
  ▪ Example: "ab \n cd " ef \ \ gh"

• Number
  ▪ Integers or floating point numbers
    – Decimal system only
    – Scientific notation allowed
  ▪ Examples: 10, -0.5, 1.5e3
Example

```json
{
    "firstName" : "John",
    "lastName" : "Smith",
    "age" : 25,
    "address" : {
        "street" : "21 2nd Street",
        "city" : "New York",
        "state" : "NY",
        "postalCode" : 10021
    },
    "phoneNumbers" : [
        { "type" : "home", "number" : "212 555-1234" },
        { "type" : "fax", "number" : "646 555-4567" }
    ]
}
```
BSON
Binary JSON
Introduction

• BSON
  ▪ Binary-encoded serialization of JSON documents
    – Allows embedding of JSON objects, arrays and standard simple data types together with a few new ones
  ▪ MongoDB database
    – NoSQL database built on JSON documents
      • [http://www.mongodb.com/](http://www.mongodb.com/)
    – Primary data representation = BSON
      • Data storage and network transfer format
  ▪ Filename: *.bson
Example

- **JSON**
  - `{ "hello" : "world" }`

- **BSON**
  - `\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00`

  - Document size
  - String data type
  - Field name
  - Field value
  - End of object
Grammar

• Document
  - Encodes one JSON object (or array or value)
    • There can be more documents in one *.bson file
      - JSON array is first transformed into an object
    • E.g.: [ "red", "blue" ] → { "0": "red", "1": "blue" }

• Structure
  - **Total document size** in a number of bytes
  - Sequence of elements
  - Terminating 0x00
Grammar

• **Element**
  - Encodes one object property (name-value pair)

• **Structure**
  - **Type** selector
    - 0x01 = double
    - 0x10 = 4B integer
    - 0x12 = 8B integer
    - 0x08 = boolean
    - 0x0A = null
    - 0x09 = datetime
    - 0x11 = timestamp
    - ...
  - **Field** name
  - **Field** value
Grammar

- **Element name**
  - Unicode **string**
    - 0x00 not allowed inside
  - Terminating 0x00

- **String**
  - Total string **length**
  - Unicode **string**
  - Terminating 0x00
Grammar

• Basic types
  ▪ byte – 1 byte (8-bits)
  ▪ int32 – 4 bytes (32-bit signed integer)
  ▪ int64 – 8 bytes (64-bit signed integer)
  ▪ double – 8 bytes (64-bit IEEE 754 floating point)
MongoDB
MongoDB

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

http://www.mongodb.org/
MongoDB Terminology

<table>
<thead>
<tr>
<th>Oracle</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>database instance</td>
<td>MongoDB instance</td>
</tr>
<tr>
<td>schema</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>row</td>
<td>document</td>
</tr>
<tr>
<td>rowid</td>
<td>_id</td>
</tr>
<tr>
<td>join</td>
<td>DBRef</td>
</tr>
</tbody>
</table>

Terminology in Oracle and MongoDB

- Each MongoDB instance has multiple databases
- Each database can have multiple collections
- When we store a document, we have to choose database and collection
MongoDB

Documents

- Use JSON
- Stored as BSON
  - Binary representation of JSON
- Have maximum size: 16MB (in BSON)
  - Not to use too much RAM
  - GridFS tool divides larger files into fragments

Restrictions on field names:
- _id is reserved for use as a primary key
  - Unique in the collection
  - Immutable
  - Any type other than an array
- The field names cannot start with the $ character
  - Reserved for operators
- The field names cannot contain the . character
  - Reserved for accessing fields
MongoDB

Data Model

- Documents have flexible schema
  - Collections do not enforce structure of data
  - In practice the documents are similar

- Challenge: Balancing
  - the needs of the application
  - the performance characteristics of database engine
  - the data retrieval patterns

- Key decision: references vs. embedded documents
  - Structure of data
  - Relationships between data
MongoDB
Data Model – References

- Including links / references from one document to another
- Normalized data models

```json
user document
{
  _id: <ObjectId1>,
  username: "123xyz"
}

contact document
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}

access document
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```
MongoDB
Data Model – References

- References provides more flexibility than embedding
- Use normalized data models:
  - When embedding would result in duplication of data not outweighed by read performance
  - To represent more complex many-to-many relationships
  - To model large hierarchical data sets
- Disadvantages:
  - Can require more roundtrips to the server (follow up queries)
MongoDB
Data Model – Embedded Data

- Related data in a single document structure
  - Documents can have subdocuments (in a field of array)
  - Applications may need to issue less queries
- Denormalized data models
- Allow applications to retrieve and manipulate related data in a single database operation

```json
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
    phone: "123-456-7890",
    email: "xyz@example.com"
  },
  access: {
    level: 5,
    group: "dev"
  }
}
```
Use embedded data models when:
- When we have “contains” relationships between entities
  - One-to-one relationships
- In one-to-many relationships, where child documents always appear with one parent document

Provides:
- Better performance for read operations
- Ability to retrieve/update related data in a single database operation

Disadvantages:
- Documents may significantly grow after creation
  - Impacts write performance
    - The document must be relocated on disk if the size exceeds allocated space
    - May lead to data fragmentation
MongoDB

Data Modification

- Operations: create, update, delete
  - Modify the data of a single collection of documents
- For update / delete: criteria to select the documents to update / remove
db.users.insert(
    {
        name: "sue",
        age: 26,
        status: "A",
        groups: [ "news", "sports" ]
    }
)
MongoDB

Data Insertion

```javascript
db.inventory.insert( { _id: 10, type: "misc", item: "card", qty: 15 } )
```

- Inserts a document with three fields into collection inventory
  - User-specified _id field

```javascript
db.inventory.update(
    { type: "book", item : "journal" },
    { $set : { qty: 10 } },
    { upsert : true }
)
```

- Creates a new document if no document in the inventory collection contains { type: "books", item : "journal" }
  - MongoDB adds the _id field and assigns as its value a unique ObjectId
  - The result contains fields type, item, qty with the specified values
MongoDB
Data Insertion and Removal

```javascript
db.inventory.save( { type: "book", item: "notebook", qty: 40 } )
```
- Creates a new document in collection `inventory` if `_id` is not specified or does not exist in the collection.

```javascript
db.inventory.remove( { type: "food" } )
```
- Removes all documents that have `type` equal to `food` from the `inventory` collection.

```javascript
db.inventory.remove( { type: "food" }, 1 )
```
- Removes one document that have `type` equal to `food` from the `inventory` collection.
MongoDB
Data Updates

db.inventory.update(
    { type: "book" },
    { $inc: { qty: -1 } },
    { multi: true }
)

- Finds all documents with type equal to book and modifies their qty field by -1

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

- Replaces document with _id equal to 10
MongoDB
Query

- Targets a specific collection of documents
- Specifies criteria that identify the returned documents
- May include a **projection** that specifies the fields from the matching documents to return
- May impose limits, sort orders, …

```javascript
db.users.find({ age: { $gt: 18 } }).sort({ age: 1 })
```

![Diagram of MongoDB query execution](image)
MongoDB
Query – Basic Queries, Logical Operators

db.inventory.find( {} )
db.inventory.find()
- All documents in the collection

db.inventory.find( { type: "snacks" } )
- All documents where the type field has the value snacks

db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
- All documents where value of the type field is either food or snacks

db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
- All documents where the type field has the value food and the value of the price field is less than 9.95
MongoDB
Query – Logical Operators

```javascript
db.inventory.find(
  { $or: [
    { qty: { $gt: 100 } },
    { price: { $lt: 9.95 } } ] } )
```

- All documents where the field `qty` has a value greater than ($gt) 100 or the value of the `price` field is less than ($lt) 9.95

```javascript
db.inventory.find( { type: 'food', $or: [ 
  { qty: { $gt: 100 } },
  { price: { $lt: 9.95 } } ] } )
```

- All documents where the value of the `type` field is `food` and either the `qty` has a value greater than ($gt) 100 or the value of the `price` field is less than ($lt) 9.95
MongoDB
Query – Subdocuments

```javascript
db.inventory.find( { 
    producer: { 
        company: 'ABC123',
        address: '123 Street'
    }
} )
```

- All documents where the value of the field `producer` is a subdocument that contains only the field `company` with the value `ABC123` and the field `address` with the value `123 Street`, in the exact order

```javascript
db.inventory.find( { 'producer.company': 'ABC123' } )
```

- All documents where the value of the field `producer` is a subdocument that contains a field `company` with the value `ABC123` and may contain other fields

**dot notation**
MongoDB
Query – Arrays

db.inventory.find( { tags: [ 'fruit', 'food', 'citrus' ] } )

- All documents where the value of the field `tags` is an array that holds exactly three elements, `fruit`, `food`, and `citrus`, in this order

db.inventory.find( { tags: 'fruit' } )

- All documents where value of the field `tags` is an array that contains `fruit` as one of its elements

db.inventory.find( { 'tags.0' : 'fruit' } )

- All documents where the value of the `tags` field is an array whose first element equals `fruit`
**MongoDB**

**Query – Arrays of Subdocuments**

```
db.inventory.find( { 'memos.0.by': 'shipping' } )
```
- All documents where the `memos` field contains an array whose first element is a subdocument with the field `by` with the value `shipping`.

```
db.inventory.find( { 'memos.by': 'shipping' } )
```
- All documents where the `memos` field contains an array that contains at least one subdocument with the field `by` with the value `shipping`.

```
db.inventory.find({
    'memos.memo': 'on time',
    'memos.by': 'shipping'
})
```
- All documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to `on time` and the field `by` equal to `shipping`. 
**MongoDB**

**Query – Limit Fields of the Result**

```javascript
db.inventory.find( { type: 'food' }, { item: 1, qty: 1 } )
```
- Only the `item` and `qty` fields (and by default the `_id` field) return in the matching documents

```javascript
db.inventory.find( { type: 'food' }, { item: 1, qty: 1, _id: 0 } )
```
- Only the `item` and `qty` fields return in the matching documents

```javascript
db.inventory.find( { type: 'food' }, { type : 0 } )
```
- The `type` field does not return in the matching documents

- Note: With the exception of the `_id` field we cannot combine inclusion and exclusion statements in projection documents.
MongoDB
Query – Sorting

```
db.collection.find().sort( { age: -1 } )
```
- Returns all documents in collection sorted by the age field in **descending** order

```
db.bios.find().sort( { 'name.last': 1, 'name.first': 1 } )
```
- Specifies the sort order using the fields from a sub-document name
- Sorts first by the last field and then by the first field in **ascending** order
Indexes

Without indexes:
- MongoDB must scan every document in a collection to select those documents that match the query statement.

Indexes store a portion of the collection's data set in an easy to traverse form:
- Stores the value of a specific field or set of fields ordered by the value of the field.
- B-tree like structures.

Defined at collection level.

Purpose:
- To speed up common queries.
- To optimize the performance of other operations in specific situations.
MongoDB
Indexes – Example
The index stores `score` values in ascending order.

MongoDB can traverse the index in either ascending or descending order to return sorted results (without sorting).
MongoDB
Indexes – Usage for **Covered Results**

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

Index Types

- Default `_id`
  - Exists by default
  - If applications do not specify `_id`, it is created automatically
  - Unique by default

- Single Field
  - User-defined indexes on a single field of a document

- Compound
  - User-defined indexes on multiple fields

- Multikey index
  - To index the content stored in arrays
  - Creates separate index entry for every element of the array
Single field index on the **score** field (ascending).

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

Multikey index on the **addr.zip** field sorts first by **userid** and then, within each **userid** value, sort by **score**.
MongoDB Indexes

- `db.people.createIndex( { "phone-number": 1 } )`
  - Creates a **single-field** index on the `phone-number` field of the `people` collection

- `db.products.createIndex( { item: 1, category: 1, price: 1 } )`
  - Creates a **compound** index on the `item`, `category`, and `price` fields

- `db.accounts.createIndex( { "tax-id": 1 }, { unique: true } )`
  - Creates a **unique** index
    - Prevents applications from inserting documents that have duplicate values for the inserted fields

- `db.collection.createIndex( { _id: "hashed" } )`
  - Creates a **hashed** index on `_id`
MongoDB

Index Types

- Geospatial Field
  - 2d indexes = use planar geometry when returning results
    - For data representing points on a two-dimensional plane
  - 2sphere indexes = use spherical (Earth-like) geometry to return results
    - For data representing longitude, latitude

- Text Indexes
  - Searching for string content in a collection

- Hash Indexes
  - Indexes the hash of the value of a field
  - Only support equality matches (not range queries)