Handling humongous data with NoSQL/MongoDB

Andreas Hartmann

What does Big Data mean???

What is the Problem with Big Data

Quelle: http://www.codefutures.com/database-sharding/
Datastore Types

Key/Value Store

Document Store

Wide-column Stores

Graph Database

Handling humongous data with NoSQL/MongoDB

Wich is the right one


NoSQL - What ist means

Query

Data is easily and quickly read/stored using primary key

Denormalize data for commonly used queries

► Schema Design is optimized for the most common Use-Cases

Developer

More technologies to have fun with

Broader choice of persistence stores

Probably Polyglot Persistence

► Store name, firstname etc in RDBMS

► Store followers in Graph database

► Store Content in RDBMS

► Store User Generated Content in Document database

Quelle: [http://www.slideshare.net/adessoAG/no-sql-9355109](http://www.slideshare.net/adessoAG/no-sql-9355109)
MongoDB in Detail

MongoDB Basics

Security and Authentication

Replication – Scaling

Map/Reduce – Binary Data Sets

Monitoring – Backup

Schema Design – Connectivity – Ecosystem

mongoDB in Detail

mongoDB - Basics

Conceptual: nested Structures with extendable Attributes

<table>
<thead>
<tr>
<th>name</th>
<th>forename</th>
<th>adress</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meier</td>
<td>Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>street</td>
<td>postcode</td>
</tr>
</tbody>
</table>
|       |          | Deich 7 | 28355 | Bremen | ???

Internal: document oriented View (mostly JSON-Format)

```
```

mongoDB - Basics

Datatypes

JavaScript Object Notation (JSON)

- string, integer, boolean, ...

```
{ "conference" : "wjax" }
```

Binary JSON (BSON)

- date, object id, binary data, ...

```
"\x16\x00\x00\x00\x02conference\x00\x06\x00\x00\x00wjax|x00\x00"
```
Collection types

- Collection
- Capped Collection
  - Logging
  - Caching
  - Archiving

```javascript
db.createCollection("myCollection");
```

```javascript
db.cappedCollection("myCappedCollection", {capped:true, size:100000, max:100});
```

ObjectId – Document

```javascript
_id = 4e7a42120fd0b0cadf316
name = Name 1
forename = Forename 1
email = email1@adesso.de
```

```javascript
db.person.insert( {"name": "Name 1", "forename": "forename 1", "email": "email1@adesso.de" } );
```

```javascript
var lastPerson = db.person.findOne();
db.hotel.insert( {"name": "Motel 1", "stars": 3, "adress": { "adress": "Adress 4", "city": "City", "postcode": 4 }, "guest": [{ "$ref": "person", "$id": lastPerson._id }]} );
```

Embedded Document

```javascript
_id = 4e7a384657da5d3d08b2c1c
name = Name 1
forename = Forename 1
email = email1@adesso.de

adress = { "street": "Street 1", "city": "City 1", "postcode": 1 }
```

```javascript
db.person.insert( {"name": "Name 1", "forename": "forename 1", "email": "email1@adesso.de", "adress": { "street": "Street 1", "city": "City 1", "postcode": 1 }} );
```

DBRef

```javascript
_id = 4e1ca3a0335a0c10869f745
name = Motel 1
stars = 3

adress = { "adress": "Adress 4", "city": "City", "postcode": 4 }
```

```javascript
var lastPerson = db.person.findOne();
db.hotel.insert( {"name": "Motel 1", "stars": 3, "adress": { "adress": "Adress 4", "city": "City", "postcode": 4 }, "guest": [{ "$ref": "person", "$id": lastPerson._id }]} );
```
### MongoDB - Basics

#### CRUD

```javascript
db.crud.insert( {"name": "Name 1", "forename": "forename 1", "email": "email@adesso.de" } );

db.crud.find( {"name": "Name 1" } );

db.crud.update( {"name": "Name 1"}, {"name": "Name 2"} );

db.crud.remove( {"name": "Name 1" } );
```

### MongoDB - Selectors

#### Aggregation

- **Count**
  
  Returns the number of objects in a collection or matching a query

  ```javascript
db.person.count( {"name": "Name 1"} );
=> 2
```

- **Distinct**
  
  Returns a list of distinct values for the given key across a collection

  ```javascript
db.person.distinct( "name" );
=> [ "Name 1", "Name 2" ]
```

### MongoDB - Transactions

MongoDB supports atomic operations on single documents.

MongoDB does not support traditional locking and complex transactions for a number of reasons:

- First, in **sharded environments**, distributed locks could be expensive and slow. MongoDB's goal is to be lightweight and fast.
- We dislike the concept of **deadlocks**. We want the system to be simple and predictable without these sort of surprises.
- We want MongoDB to **work well for realtime problems**. If an operation may execute which locks large amounts of data, it might stop some small light queries for an extended period of time.

**Quelle:** [http://www.mongodb.org/display/DOCS/Atomic+Operations](http://www.mongodb.org/display/DOCS/Atomic+Operations)
Applying to Multiple Objects At Once

- Multi-update apply the same modifier to every relevant object.
- To make it fully isolated you can use the \$atomic modifier.
- It holds the global write lock while updating the selected documents, **blocking all other read and write operations until it is done**.

```javascript
db.collection.update(query, update, <upsert>, <multi>)
```

```javascript
db.transaction.update( { "name": "Name 3", $atomic: 1 }, { $inc: { "age": 1 } }, false, true );
```

MongoDB Indexes are similar like DB Systems

- Accelerate query Documents
- Default Index on the \_id Field

Index Types

- Simple Indices
  ```javascript
db.person.ensureIndex( { "name": -1 } );
```

- Compound Indices
  ```javascript
db.person.ensureIndex( { "name": -1, "forename": -1 } );
```

- Unique Indices
  ```javascript
db.person.ensureIndex( { "email": -1 }, { unique: true } );
```

Show all Indexes

```javascript
db.system.indexes.find();
```

Measure Query Execution

```javascript
db.person.find("name": "Name 1").explain();
```
mongoDB – Security and Authentication

Supports only basic security.

Authenticates a username and password in the context of a particular database.

Once authenticated, a normal user has full read and write access to the database.

Read-only users

Passwords send encrypted.

mongoDB – Replication

Replica Sets
- A replica set consists of two or more nodes that are copies of each other.
- The replica set automatically selects a primary (master).
- Drivers can automatically detect when a replica set primary changes and will begin sending writes to the new primary.

Why Replica Sets
- Automated Failover
- Read Scaling (slaveOkay Method)
- Maintenance
- Disaster Recovery

mongoDB – Scaling

Sharding
- Horizontal scaling across multiple nodes

Sharding Key
- collection: monkey
- monkeykey: location
- users: name = Miller
- users: name = Nessman
- shards:

Characteristics
- Inserts are balanced between shards
- Common queries are routed to a subset of the shards

Quelle: http://www.mongodb.org/display/DOCS/Sharding+Introduction
**MongoDB – Map/Reduce**

**Parallel processing huge datasets on distributed systems**

**Map**

```javascript
var map = function() {
    emit( this.author, {
        pages: this.pages
    });
};
```

**Reduce**

```javascript
var reduce = function( key, values ) {
    var sum = 0;
    values.forEach( function( doc ) {
        sum += doc.pages;
    } );
    return { "pages": sum }
};
```

**Execute**

```javascript
db.bookstore.mapReduce( map, reduce,
    { out: "myresultcollection" } );
```

**Result**

```
{ "result" : "myresultcollection", "timeMillis" : 156, "counts" : { "input" : 7, "emit" : 7, "reduce" : 3, "output" : 3 }, "ok" : 1 }
```
**mongoDB – Binary Data Sets**

**GridFS**
- GridFS is a storage specification for large objects in MongoDB.
- It works by *splitting large object into small chunks*.
- Each chunk is stored as a separate document in a chunks collection.
- **Metadata about the file**, including the filename, content type, and any optional information needed by the developer, is *stored as a document in a files collection*.
- So for any given file stored using GridFS, there will exist one document in files collection and one or more documents in the chunks collection.

**mongoDB – Schema Design**

**Relational World**
- Correct design for a given entity relationship model is independent of the Use Case

**MongoDB World**
- Schema design is not only a function of the data to be modeled but also of the Use Case
- Schema design is optimized for the most common Use Cases

**mongoDB – Backup**

**Export/Import**
- Scope on Single Collections
- Supported Types are JSON or CSV
- Don’t work with Binary Data
- Specify a filter for the query, or a list of fields to output

**Dump/Restore**
- Scope on the whole Database
- Exporttype is BSON

**mongoDB – Schema Design**

When do we embed data versus linking?

How many collections do we have, and what are they?

When do we need atomic operations? These operations can be performed within the scope of an document, but not across documents.

Must we shard? How will we shard? What is the shard key?

What indexes will we create to make query and updates fast?

*Quelle: [http://www.mongodb.org/display/DOCS/Schema+Design](http://www.mongodb.org/display/DOCS/Schema+Design)*
mongoDB - Connectivity

<table>
<thead>
<tr>
<th>Connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>C# and .NET</td>
</tr>
<tr>
<td>C++</td>
</tr>
<tr>
<td>Erlang</td>
</tr>
<tr>
<td>Haskell</td>
</tr>
<tr>
<td>Java</td>
</tr>
<tr>
<td>Java POJO Mapper</td>
</tr>
<tr>
<td>JavaScript Shell</td>
</tr>
<tr>
<td>Perl</td>
</tr>
</tbody>
</table>

Connectivity - Java

Classpath: mongo.jar

```java
import com.mongodb.BasicDBObject;
import com.mongodb.DB;
Mongo mongo = new Mongo("localhost", 27017);
DB db = mongo.getDB("driverDB");
```

Case Studies

```java
import com.mongodb.DBCollection;
DBCollection personCollection = db.getCollection("Person");
```