BERLIN EXPERT DAYS 2013 {join the talk}

adesso business. people. technology.

# Handling humongous data with NoSQL/MongoDB

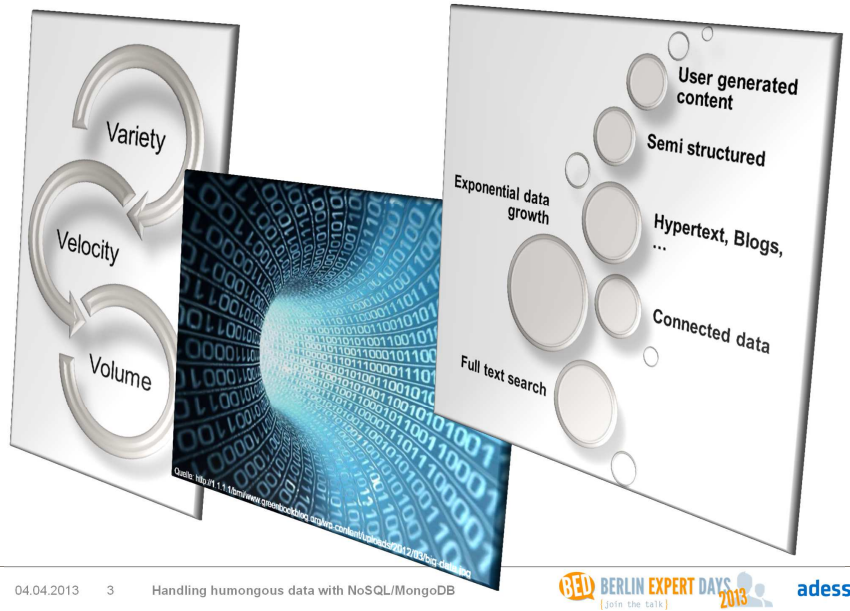
**Andreas Hartmann**



## NoSQL Tagcloud

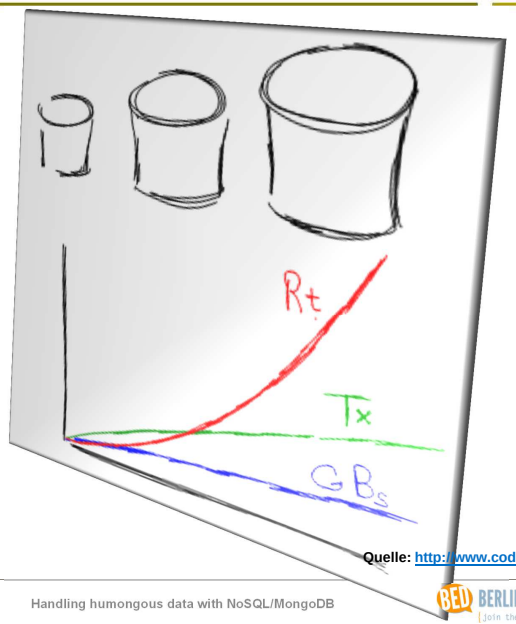



## What does Big Data mean???






## What is the Problem with Big Data

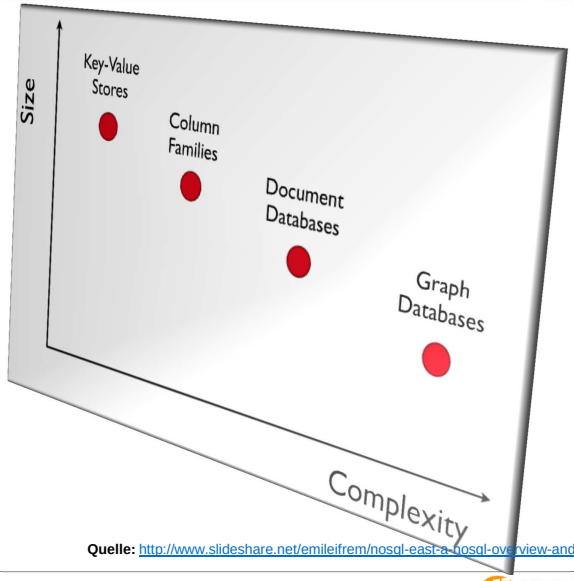



Quelle: http://www.codefutures.com/database-sharding/

## Datastore Types




## Wich is the right one

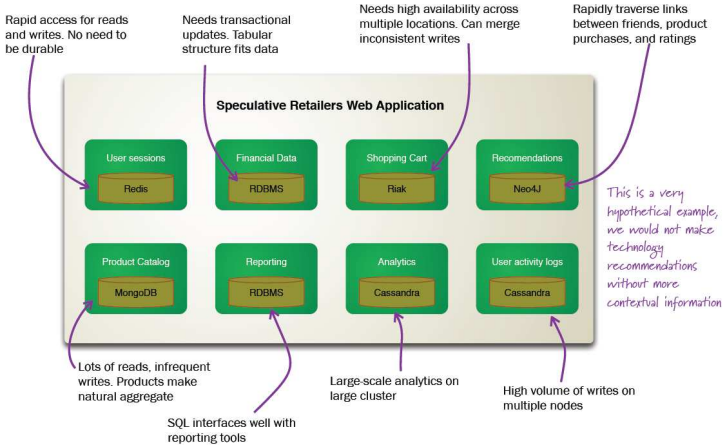



**Quelle:** http://www.slideshare.net/emileifrem/nosql-east-a-nosql-overview-and-the-benefits-of-graph-databases



## Wich is the right one

what might Polyglot Persistence look like?




**Quelle:** http://martinfowler.com/articles/nosql-intro.pdf



## NoSQL - What ist means

**Query**

**Data is easily and quickly read/stored using primary key**

**Denormalize data for commonly used queries**

- Shema Design is optimized for the most common Use-Cases

**Developer**

**More technologies to have fun with**

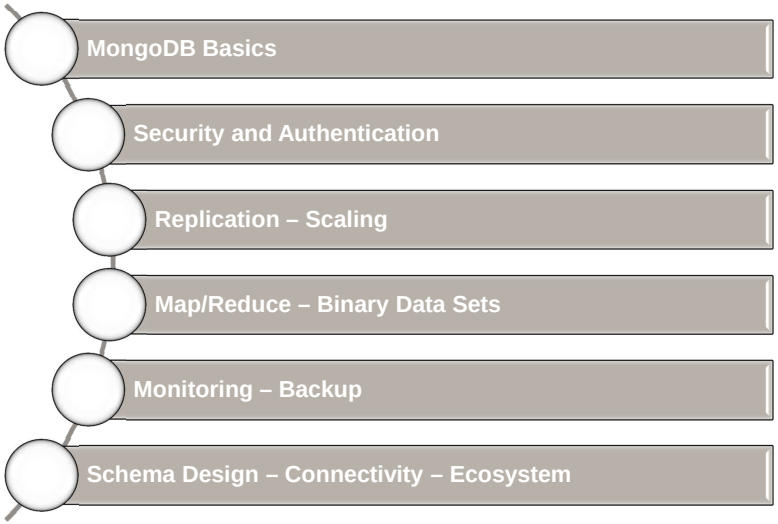
**Broader choice of persistence stores**

**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

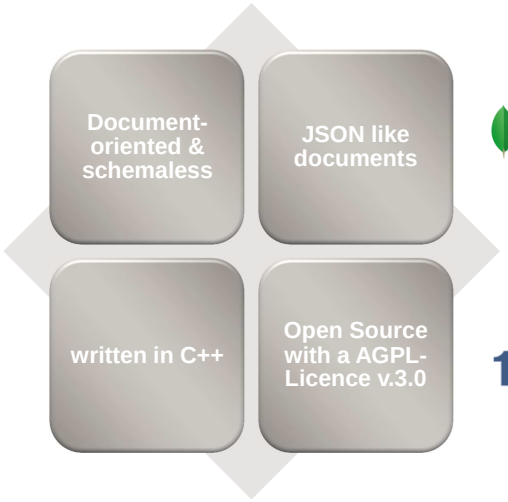
**Quelle:** http://www.slideshare.net/adessoAG/no-sql-9355109

## MongoDB in Detail






## mongoDB in Detail






## mongoDB - Basics

- Conceptual: nested Structures with extendable Attributes

| name | forename | adress | | | . . . |
|---|---|---|---|---|---|
| | | street | postcode | city | |
| Meier | Max | Deich 7 | 28355 | Bremen | ??? |
| . . . | . . . | . . . | . . . | . . . | . . . |

- Internal: document oriented View (mostly JSON-Format)

```
{"name":"Meier",
 "forename":"Max",
 "adress":{"street":"Deich 7",
           "postcode":28355,
           "city":"Bremen"}
 ". . .":"???"}
```



## 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

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

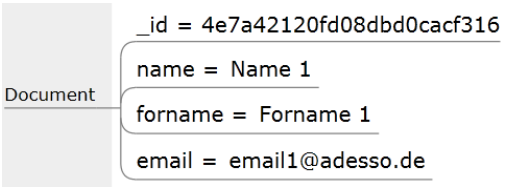
- Capped Collection
  - Logging
  - Caching
  - Archiving

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





**ObjectId – Document**




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





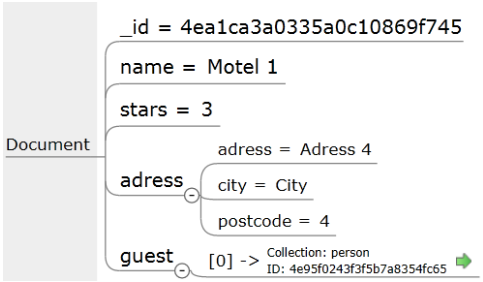
**Embedded Document**


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





**DBRef**




```
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**

```
db.crud.insert( {"name": "Name 1", "forename": "forename
1", "email": "email1@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**

```
db.person.insert( {"name": "Name 1", "age": 20 } );
db.person.insert( {"name": "Name 1", "age": 38 } );
db.person.insert( {"name": "Name 2", "age": 42 } );
```

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

```
db.person.count( {"name": "Name 1"} );

=> 2
```

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

```
db.person.distinct( "name" );

=> [ "Name 1", "Name 2" ]
```



## mongoDB - Selectors

**Aggregation**

- Group
  Returns an array of grouped items

```
db.person.group( {
      key: { "name": true },
      cond: { "name": "Name 1" },
      initial: { "count": 0 },
      reduce: function( document, prev ) {
            prev.count += document.age;
      }
} );

db.person.insert( {"name": "Name 1", "age": 20 } );
db.person.insert( {"name": "Name 1", "age": 38 } );
db.person.insert( {"name": "Name 2", "age": 42 } );

=> [ { "name" : "Name 1", "count" : 58 } ]
```



## 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. Mongo DB'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 Mongo DB 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

## mongoDB - Transactions

**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 hold the global write lock while. Updating the selected documents, ***blocking all other read and write operations until it is done***.

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

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



## mongoDB - Indexes

**MongoDB Indexes are similar like DB Systems**

- acelerate query Documents
- Default Index on the `_id` Field

**Index Types**

- Simple Indices

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

- Compound Indices

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

- Unique Indices

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



## mongoDB - Indexes

**Show all Indexes**

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

```
. . .

{ "v" : 1,                    => Index Version;
                                 1 is new in Mongo 2.0
"key" : { "forename" : -1 }   => Fields with Index
                                 1 Ascending
                                 2 Descending
"ns" : "consoleDB.person",    => Namespace (DB.Coolection)
"name" : "forename_-1" }      => Generated from Index
                                 Fields
```



## mongoDB - Indexes

**Measure Query Execution**

```
db.person.find( {"name": "Name 1" } ).explain();
```

```
{
    "cursor" : "BasicCursor",
    "nscanned" : 3,
    "nscannedObjects" : 3,
    "n" : 1,
    "millis" : 0,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "isMultiKey" : false,
    "indexOnly" : false,
    "indexBounds" : {

    }
}
```

```
{
    "cursor" : "BtreeCursor name_-1",
    "nscanned" : 1,
    "nscannedObjects" : 1,
    "n" : 1,
    "millis" : 0,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "isMultiKey" : false,
    "indexOnly" : false,
    "indexBounds" : {
        "name" : [
            [
                "Name 1",
                "Name 1"
            ]
        ]
    }
}
```

## 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



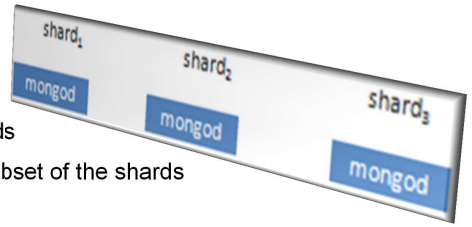

## mongoDB – Scaling

**Sharding**

- Horizontal scaling across multiple nodes

**Sharding Key**

| collection | minkey | maxkey | location |
|---|---|---|---|
| users | { name : 'Miller' } | { name : 'Nessman' } | $shard_2$ |
| users | { name : 'Nessman' } | { name : 'Ogden' } | $shard_4$ |
| ... | | | |



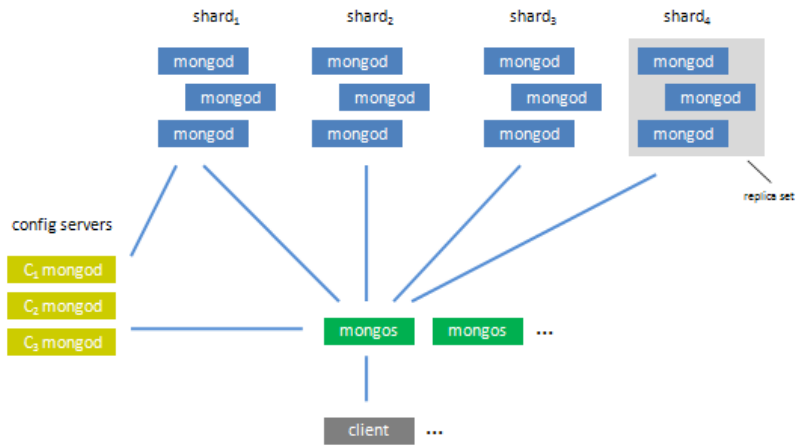

**Characteristics**

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



## mongoDB – Scaling

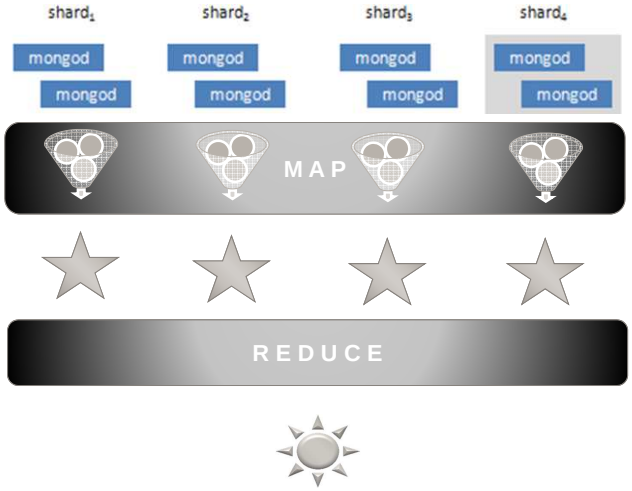
**Replica Sets comes together with Sharding**




Quelle: http://www.mongodb.org/display/DOCS/Sharding+Introduction

## mongoDB – Map/Reduce

**Parallel processing huge datasets on distributed systems**






## mongoDB – Map/Reduce

**Map**

Map → Reduce → Execute

```
var map = function()  {

    emit( this.author,
      { pages: this.pages } );
};
```

```
{
  "_id" : ObjectId("4e9fe8f76bdf87de70353d00"),
  "author" : "Name 1",
  "title" : "Book 1",
  "pages" : 300.0
}
{
  "_id" : ObjectId("4e9fe8f76bdf87de70353d01"),
  "author" : "Name 1",
  "title" : "Book 1",
  "pages" : 100.0
}
{
  "_id" : ObjectId("4e9fe8f76bdf87de70353d02"),
  "author" : "Name 1",
  "title" : "Book 1",
  "pages" : 100.0
}
. . .
```



## mongoDB – Map/Reduce

**Reduce**

Map → Reduce → Execute

```
reduce('Name 1', [{pages: 300.0}, {pages: 100.0}, {pages: 100.0}, . . .]);
```

```
var reduce = function( key, values )  {
      var sum = 0;

      values.forEach( function( doc )  {
                  sum += doc.pages;
      } );

      return { "pages": sum };
};
```



## mongoDB – Map/Reduce

**Execute**

Map → Reduce → Execute

```
db.bookstore.mapReduce( map, reduce,
      { out: "myresultcollection" } );
{
      "result" : "myresultcollection",
      "timeMillis" : 156,
      "counts" : {
            "input" : 7,
            "emit" : 7,
            "reduce" : 3,
            "output" : 3
      },
      "ok" : 1,
}
```

```
{
  "_id" : "Name 1",
  "value" : {
    "pages" : 500.0
  }
}
{
  "_id" : "Name 2",
  "value" : {
    "pages" : 30.0
  }
}
{
  "_id" : "Name 3",
  "value" : {
    "pages" : 600.0
  }
}
```

## 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.

```
_id = 4e9c6c89d61575a507036486
filename = Test_Document.pdf
chunkSize = 262144
uploadDate = 17.10.2011
md5 = badb07a721e18fe9 21704d67e31db9a
length = 9201705
```



## 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

**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

**Quelle:** http://www.mongodb.org/display/DOCS/Schema+Design



## 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

## mongoDB - Connectivity

| Connectivity |
| --- |
| C |
| C# and .NET |
| C++ |
| Erlang |
| Haskell |
| **Java** |
| **Java POJO Mapper** |
| **JavaScript Shell** |
| Perl |

| Connectivity |
| --- |
| PHP |
| Python |
| REST API |
| Ruby |
| Scala |
| ... |



## mongoDB - Connectivity

DRIVERS
Get The Latest Drivers »

**Connectivity - Java**

- Classpath: `mongo.jar`



## mongoDB - Connectivity

DRIVERS
Get The Latest Drivers »

**Connecting to MongoDB**

```
import com.mongodb.BasicDBObject;
import com.mongodb.DB;

Mongo mongo = new Mongo( "localhost" , 27017 );
DB db = mongo.getDB( "driverDB" );
```

**Getting a Collection**

```
import com.mongodb.DBCollection;

DBCollection personCollection = db.getCollection(
"Person" );
```



## Case Studies