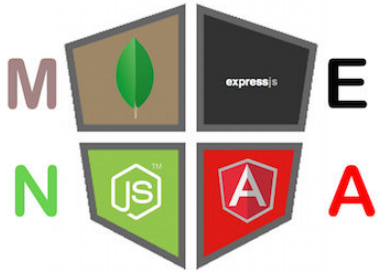
Karlsruhe Institute of Technology

LSDMA

MongoDB as a NoSQL Database

Parinaz Ameri, Marek Szuba

Steinbuch Centre for Computing

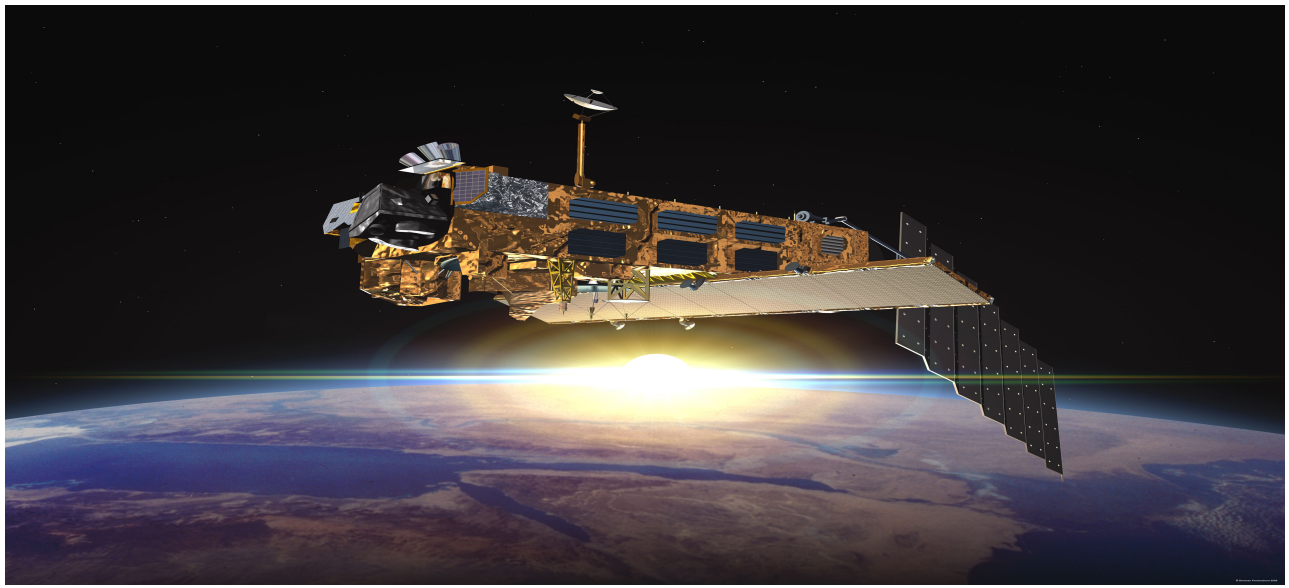


(c) edureka.co, rivetlogic.com, adrianmejia.c

Introduction

Parinaz Ameri (parinaz.ameri@kit.edu)

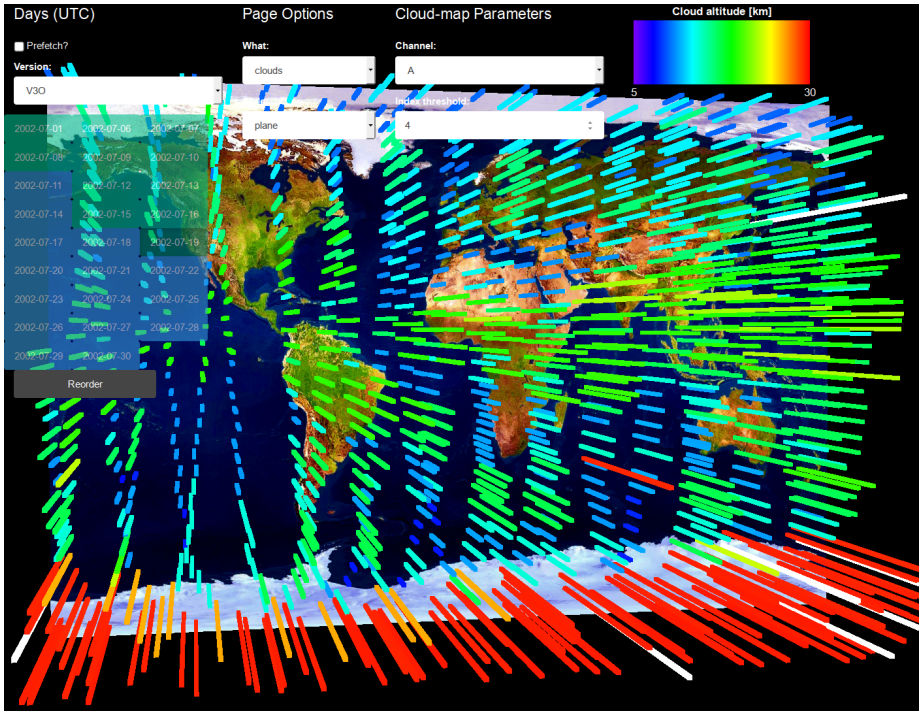
- PhD researcher in computer science at KIT
- Working at DLCL Climatology of the LSDMA project
- Working on two projects using MongoDB
- Focus on geospatial / meteorological data



Introduction

Marek Krzysztof Szuba (`marek.szuba@kit.edu`)

- PhD in high-energy particle physics
- Currently working at DLCL Climatology of LSDMA
- Focus: retrieval and visualisation of satellite climate data
  - a MEAN application using WebGL



Round of Introduction

Karlsruhe Institute of Technology

Let us get to know you!

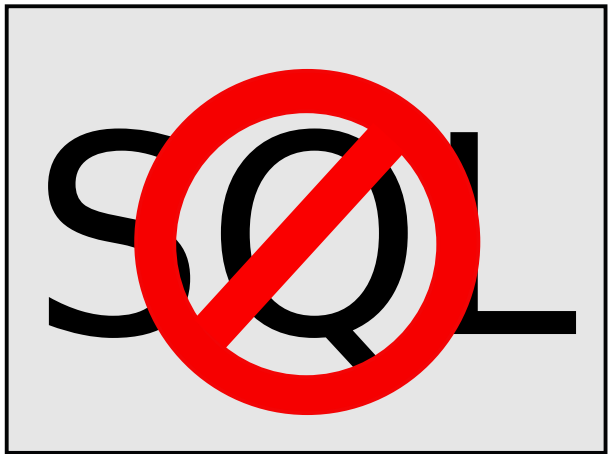
Outline

Karlsruhe Institute of Technology

ACID Transactions

- **Atomicity**: "all or nothing"
  - no interruptions mid-transaction
- **Consistency**: "valid data"
  - takes the database from one consistent state to another consistent state
- **Isolation**: "one after the other"
  - concurrent transactions will be treated as if they were serial
- **Durability**: "no loss"
  - ensures that any transaction committed to the database will not be lost

Where Did the Problem Start?



- RDBMS and the Web 2.0 era (Big Data)
- RDBMS and transactions (Scalability)
- RDBMS and JOIN (Partitioning)
- RDBMS and new hardware

CAP Theorem (Brewer, Lynch)

- **Consistency**:
  Do all applications see all the same data?
- **Availability**:
  Can I interact with the system in the presence of a failure?
- **Partitioning**:
  If two segments of your system can not talk to each other, can they progress by they own?
  - if yes, you sacrifice consistency
  - if no, you sacrifice availability

CAP Theorem (Brewer, Lynch)

- **Consistency**:
  Do all applications see all the same data?
- **Availability**:
  Can I interact with the system in the presence of a failure?
- **Partitioning**:
  If two segments of your system can not talk to each other, can they progress by they own?
  - if yes, you sacrifice consistency
  - if no, you sacrifice availability

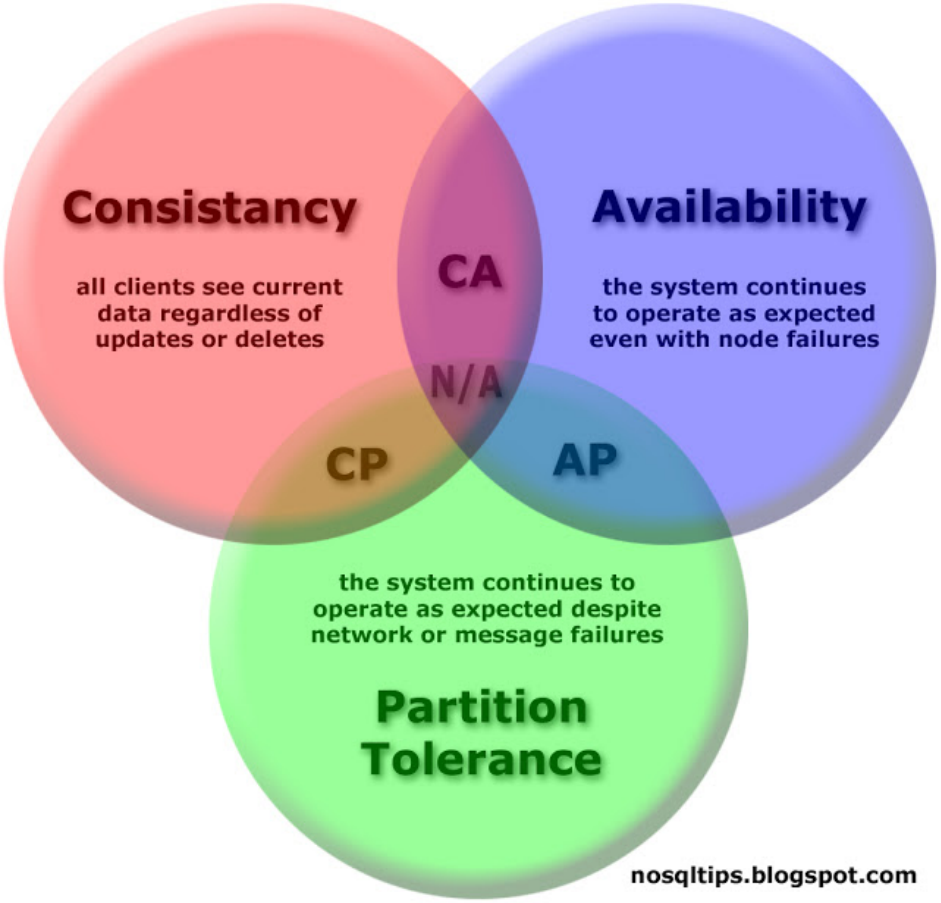
CAP Theorem (Brewer, Lynch)

- **Consistency**:
  Do all applications see all the same data?
- **Availability**:
  Can I interact with the system in the presence of a failure?
- **Partitioning**:
  If two segments of your system can not talk to each other, can they progress by they own?
  - if yes, you sacrifice consistency
  - if no, you sacrifice availability

CAP Theorem (Brewer, Lynch)

- **Consistency**:
  Do all applications see all the same data?
- **Availability**:
  Can I interact with the system in the presence of a failure?
- **Partitioning**:
  If two segments of your system can not talk to each other, can they progress by they own?
  - if yes, you sacrifice consistency
  - if no, you sacrifice availability

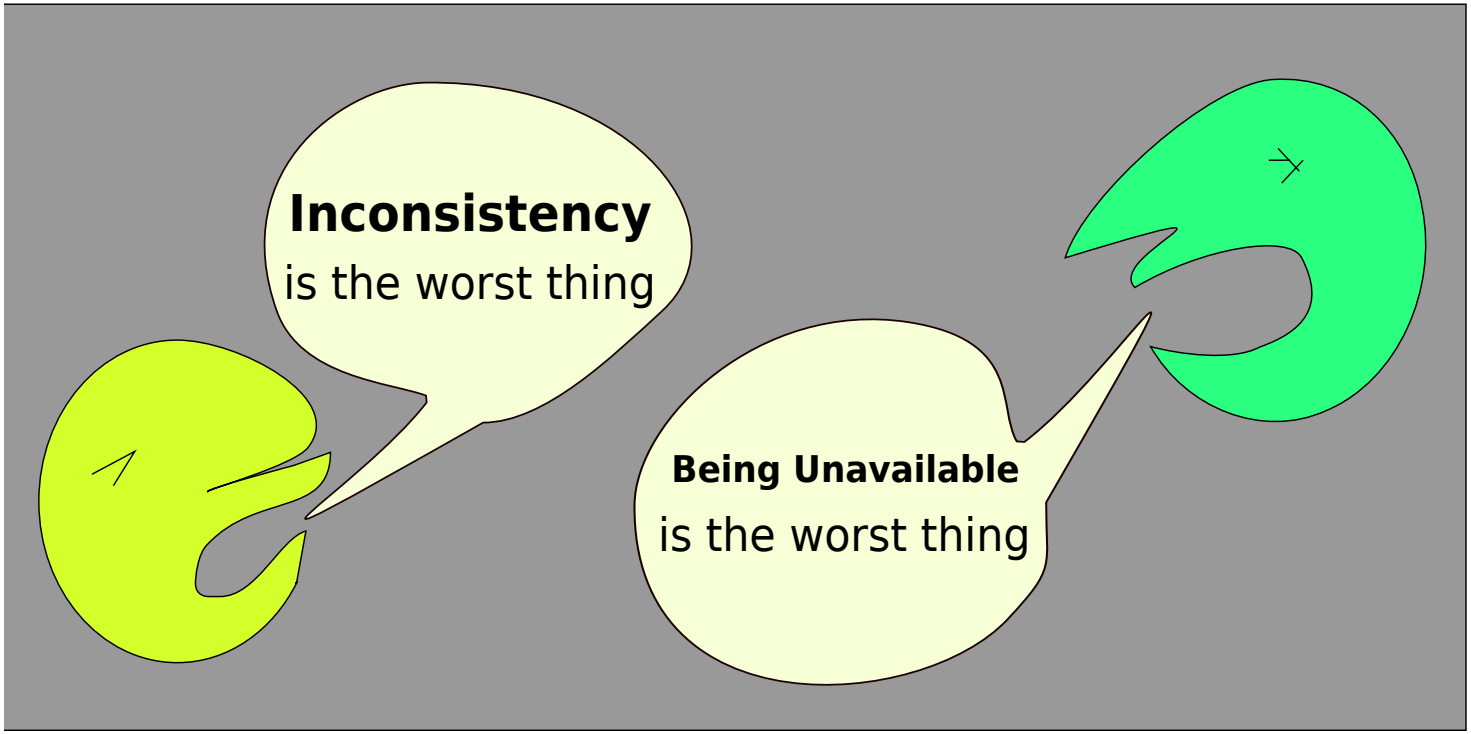
CAP Theorem


Consistancy
all clients see current data regardless of updates or deletes
Availability
the system continues to operate as expected even with node failures
CA
N/A
CP
AP
the system continues to operate as expected despite network or message failures
Partition Tolerance
nosqltips.blogspot.com

ACID *vs.* BASE

BASE:

- **B**asic **A**vailability
- **S**oft-state
- **E**ventual Consistency



NoSQL Databases

Key/Value	Column	Document	Graph
Dynamo	BigTable	MongoDB	Neo4j
Redis	HBase	CouchDB	FockDB
	Cassandra	Riak	

Typical Usage:

- *Key/Value*: distributed hash table, caching
- *Column families*: distributed data storage
- *Document Store*: Web applications
- *Graph databases*: social networking, Recommendations

Intro to Mongo

- MongoDB from "hu*mongo*us"
- Open Source
- Written in C++
- Developed by MongoDB Inc. (formerly 10gen)
- First release in 2009
- Last released stable version (as of late August 2015) is 3.0.6
  - 2.6.11 on the 2.6 branch
- `mongo` shell — an interactive JavaScript shell
- Both MongoDB and its drivers are available under free licenses
- Commercial support, MongoDB Enterprise

MongoDB: Overview

- Document-oriented database system:
  - schema-less
  - key-value store
  - JSON documents instead of rows of a table:

```
{
  mykey: myvalue, answer: 42,
  myarray: [ "red", "green", "blue" ]
}
```

- Storage in BSON (short for Binary JSON):
  - a serialization format used to store documents and make remote procedure calls in MongoDB
- Excellent documentation:
  - `https://www.mongodb.org/`
  - Web pages, videos, workshops, training, newsletters, ...

MongoDB: Overview

- Document-oriented database system:
  - schema-less
  - key-value store
  - JSON documents instead of rows of a table:

```
{
  mykey: myvalue, answer: 42,
  myarray: [ "red", "green", "blue" ]
}
```

- Storage in BSON (short for Binary JSON):
  - a serialization format used to store documents and make remote procedure calls in MongoDB
- Excellent documentation:
  - https://www.mongodb.org/
  - Web pages, videos, workshops, training, newsletters, ...

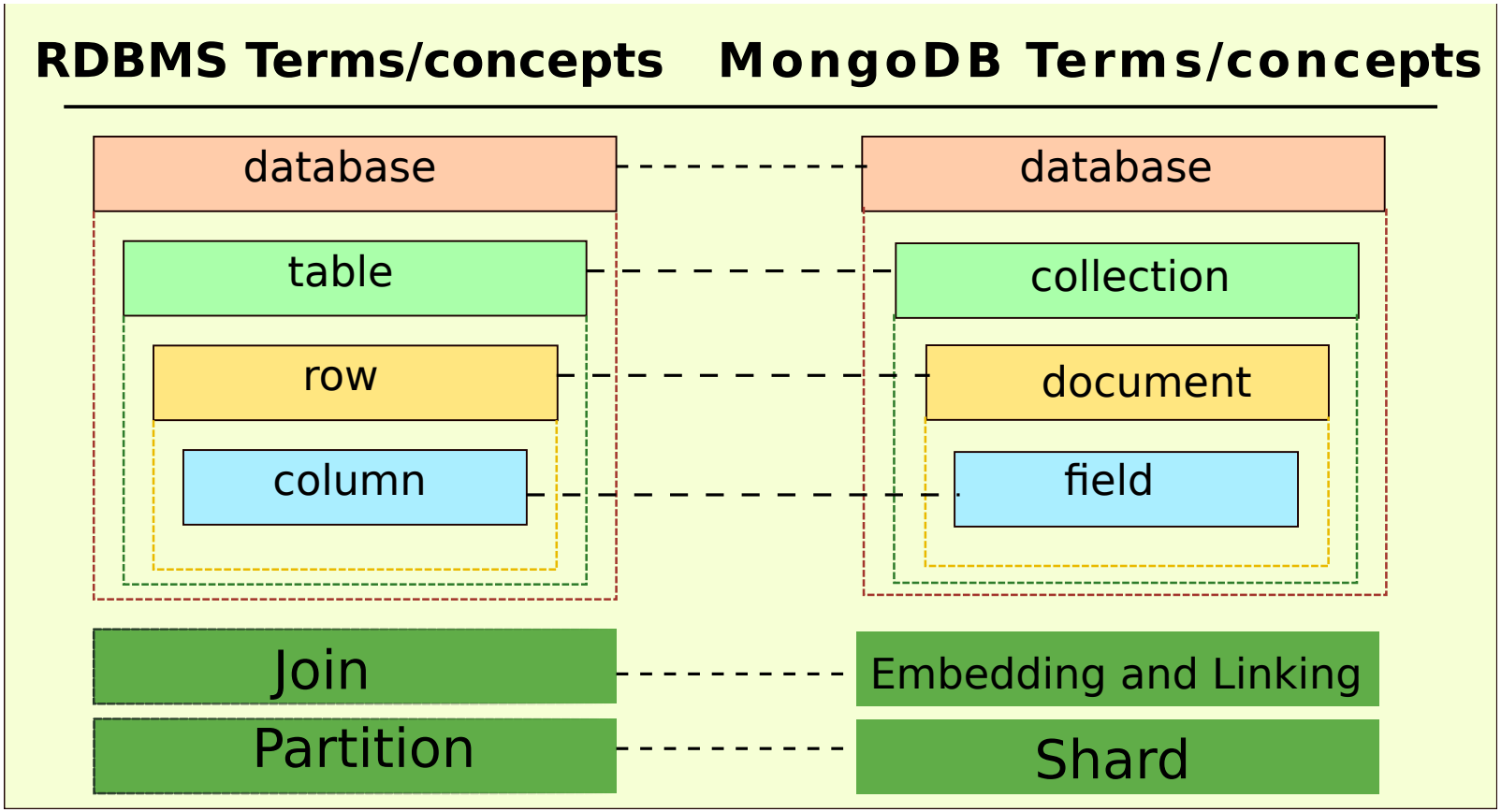
MongoDB: Overview

- Document-oriented database system:
  - schema-less
  - key-value store
  - JSON documents instead of rows of a table:

```
{
  mykey: myvalue, answer: 42,
  myarray: [ "red", "green", "blue" ]
}
```

- Storage in BSON (short for Binary JSON):
  - a serialization format used to store documents and make remote procedure calls in MongoDB
- Excellent documentation:
  - `https://www.mongodb.org/`
  - Web pages, videos, workshops, training, newsletters, ...

Terminology



Installation and Configuration

Karlsruhe Institute of Technology

To install:

- get repositories, packages or source code from the Web site
- packages from standard repositories, *e.g.*:
  - Red Hat/Fedora/CentOS:
    `yum install mongodb mongodb-server` (from `EPEL`)
  - Ubuntu/Debian:
    `apt-get install mongodb` (from `universe`)
  - Mac OS X (with Homebrew):
    `brew install mongodb`

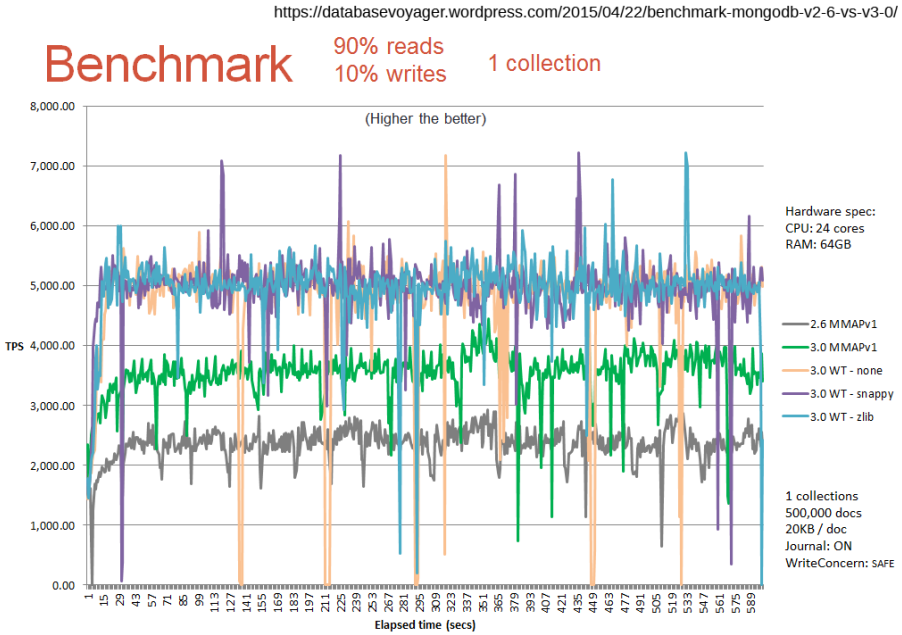
To configure: edit /etc/mongod.conf

- 2.6, 2.4 and older: old key–value format
- 2.6 and newer: new YAML format

Start the MongoDB service with *e.g.*:

- `service mongod start` or `systemctl start mongodb.service`

What Is New in Version 3.0?



- New storage engine: WiredTiger
  - on 64-bit systems only
  - much faster than the old engine (MMAPv1)
  - integrated compression
- Improved MMAPv1 performance
- Data locking:
  - WiredTiger: document-level
  - MMAPv1: collection-level
- New query-introspection system: `db.collection.explain()`
- Enhanced indexing, replica sets and sharding
- ...and many others!

Configuration Syntax

Karlsruhe Institute of Technology

- Old format: flat list of `key=value` pairs
  - keys identical to command-line options
  - somewhat inconsistent key casing:
    - sometimeslikethis,
    - sometimes_like_that,
    - andSometimesLikeThat
- YAML: hierarchical structure, `key:value` notation
  - all keys in camelCase
- Cores of key names *generally* the same
- Comment lines always preceded by a hash

Configuration Syntax

Example

Old syntax

```
# Example MongoDB config file
# (these two lines are comments)
fork = true
dbpath = /var/lib/mongodb
logpath = /var/log/mongod.log
bind_ip = 127.0.0.1
sslMode = preferSSL
auth = true
```

YAML

```
# Example MongoDB config file
# (these two lines are comments)
processManagement:
  fork: true
storage:
  dbPath: /var/lib/mongodb
systemLog:
  destination: file
  path: /var/log/mongod.log
net:
  bindIp: 127.0.0.1
  ssl:
    mode: preferSSL
security:
  authorization: true
```

ObjectID

ObjectID is a 12-byte BSON type, constructed using:

- a 4-byte timestamp,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

MongoDB uses ObjectId values as the default values for `_id` fields:

```
{
  "_id" : ObjectId("52308273355b531d2aeb768b"),
  "loc" : {
    "lat" : 68.91,
    "lon" : 46.98
  },
  "id" : "MLS-Aura_L2GP-O3_v03-30-c01_2005d001.0",
  "time" : ISODate("2004-12-31T23:59:41Z")
}
```

CreateRUD

1. `db.collection.insert()`
   - insert one or more documents into a collection
   - The primary key_`id` is automatically added if `_id` field is not specified
2. `db.collection.save()`
   - depending on the document parameters, insert a new document or update an existing one
3. `mongoimport / mongoexport`
   - import/export content as JSON, CSV, or TSV
   - a single-threaded method
   - inserts one document at a time
4. `mongodump / mongorestore`
   - create a binary export of the database content / use a binary dump to import to database
   - a backup method

Create in Perspective

Karlsruhe Institute of Technology

MongoDB

```
db.authors.insert({
  id: 2,
  first_name: "Arthur",
  last_name:  "Miller",
  age: 25,
  })
```

SQL

```
CREATE TABLE authors(
 id MEDIUMINT NOT NULL
     AUTO_INCREMENT,
 user_id Varchar(30),
 first_name char(15),
 last_name char(15),
 age Integer,
 PRIMARY KEY (id)
)
```

CReadUD

- `db.collection.find(criteria, projection)`
  - select documents in a collection
  - return a cursor (a pointer to the result set of a query)
  - `db.collection.findone()` is its limited version

- Cursor methods:
  - `count()`, `limit()`, `skip()`, `snapshot()`, `sort()`,
  - `batchSize()`, `explain()`, `hint()`, `forEach()`, …
  - *e.g.* `db.collection.find().count()`

e.g.

```
db.cities.find({"city": "KARLSRUHE"})
```

Conditional Operators

- Comparison Query Operators
  - `$gt, $gte, $lt, $lte, $ne, $all, $in, $nin`
- Logical Query Operators
  - `$and, $or, $nor, $not`
- Element Query Operators
  - `$exists, $type`
- Geospatial Query Operators
  - `$geoWithin, $centerSphere`

e.g.

```
db.cities.find({"loc": {
  $geoWithin: { $center: [[8.4, 49.017], 0.1] }
}})
```

Read in Perspective

MongoDB

```
db.authors.find({last_name: "Miller"})
```

SQL

```
SELECT * FROM authors WHERE last_name = "Miller"
```

CR**Update**D

1. `db.collection.update(query, update, options)`:
   - modify existing documents
   - based on query specifications can modify a field in or the whole document
   - options are:
     - `upsert`: *boolean* — insert new document if no match exists
     - `multi`: *boolean*
     - `writeConcerns`: *document* — concern levels: (unacknowledged, acknowledged, journaled, ...)
2. `db.collection.save()`
   - if the document contains `_id` field
   - save will call update, with upsert option

Atomic Modifiers

Field operators:

- `$inc, $mul, $set, $unset, $rename`

Array operators:

- `$, $push, $pull`

```
db.books.update(
  item: "Divine Comedy" ,
  {
    $set: { price: 18 },
    $inc: { stock: 5 }
  }
)
```

CRUDelete

1. `db.collection.remove(query, options)`:
   - removes all documents from a collection, w/o indexes
   - options are:
     - `justOne` *boolean*
     - `writeConcerns`
2. `db.collection.drop()`
   - drops the entire collection, including the indexes

Delete in Perspective

Karlsruhe Institute of Technology

MongoDB

```
db.authors.remove({_id: 1})
```

SQL

```
DELETE FROM authors WHERE id = "1"
```

Authentication, Authorisation and Encryption

Overview

Default MongoDB configuration:

- no user authorisation — full access for all connections
- full trust between cluster members
- plain-text network communication
- certain versions / packages: listen on all network interfaces

Unsuitable for production systems!

August 2015: **39,134** unsecured MongoDB instances found on the Internet, exposing up to almost **620 TB** of data:

`http://blog.binaryedge.io/2015/08/10/data-technologies-and-security-part-1/`

Encryption

- Data transfer: supports SSL
  - `--sslMode` on the command line, `net.ssl.mode` in configuration
  - different degrees of enforcement
  - valid certificate(s) required
  - not available in old binary builds from `mongodb.orb`
- Data at rest: not encrypted
  - use third-party solutions, *e.g.* LUKS or BitLocker

Authorisation

Access to Instance

- First and foremost: protect your daemons
  - only listen on required network interfaces
    - `net.bindIp` in configuration
    - best practice: only 127.0.0.1 until needed and properly secured
  - filter network traffic *etc.*

Authorisation

Inside the Instance

- Role-Based Access Control
  - an user assigned to one or more *roles*
  - a role contains a set of *privileges*
  - a privilege: a *resource* plus a set of allowed *actions*
  - a resource: collection, set of collections, database or cluster
- Some built-in roles:
  - user access: `read`, `readWrite`
  - database administration: `dbAdmin`, `userAdmin`
  - cluster administration: `clusterManager`
  - `backup`, `restore`
  - `root`
- Custom roles possible
- Users are added per-database but can have roles spanning other databases

Authorisation

Inside the Instance

- To enable:
  - pass `--auth` on the command line,
  - set `security.authorization` in configuration, or
  - create a key file, set `security.keyFile` in configuration
- Disables anonymous access
  - *localhost exception*: local access and **no users defined**
    - 3.0: only create first user in `admin`
    - older versions: full instance access
  - also applies to shards

Authentication

- Two modes
  - user access
  - cluster membership
- Several mechanisms available
  - standard: challenge-response, X.509 certificates
  - Enterprise: Kerberos, LDAP
- One user per connection

Challenge-Response Authentication

- Users: validate password against user database
- Clusters: same *key file* on different machines
  - contains a (preferably random) key 6–1024 bytes long, Base64 character set
- Transmits modified MD5 password hash
- SSL recommended — hash protected against replay attacks but could be brute-forced

X.509 Authentication

- As seen on the Web and elsewhere
- SSL required
- Users: certificate subject registered in user database
- Clusters: *unique* certificate and key for each machine, in a PEM file
- Constraints on certificates
  - users: appropriate key-usage data
  - clusters: same CA for all members, subject must match host name
  - mustn't re-use subject for user and cluster authentication

Accounting

- monitor database use, performance studies *etc.*
- standard: basic logging, journal monitor, profiling
- Enterprise: full event audit system

Accounting

Journal Monitor

- Journalling — a mechanism for ensuring data consistency
- Provides basic monitoring features: `serverStatus`, `journalLatencyTest`
- Useful for assessing overall performance

Profiling

- estimate performance/cost of database operations
- enabled per-instance or per-database
- stores results in the `system.profile` collection

Index Introduction

- Index is a special data structure at **collection** level.
- Using indexes ensures that MongoDB only scans the smallest possible number of documents.
- Any field or sub-field of documents in a collection can be indexed.
- No more than 64 indexes per collection are possible. (MS SQL server supports 1,000 per table)
- MongoDB does not support clustered indexes.

Index Introduction

- Index is a special data structure at **collection** level.
- Using indexes ensures that MongoDB only scans the smallest possible number of documents.
- Any field or sub-field of documents in a collection can be indexed.
- No more than 64 indexes per collection are possible. (MS SQL server supports 1,000 per table)
- MongoDB does not support clustered indexes.

Index Introduction

- Index is a special data structure at **collection** level.
- Using indexes ensures that MongoDB only scans the smallest possible number of documents.
- Any field or sub-field of documents in a collection can be indexed.
- No more than 64 indexes per collection are possible. (MS SQL server supports 1,000 per table)
- MongoDB does not support clustered indexes.

Index Introduction

- Index is a special data structure at **collection** level.
- Using indexes ensures that MongoDB only scans the smallest possible number of documents.
- Any field or sub-field of documents in a collection can be indexed.
- No more than 64 indexes per collection are possible. (MS SQL server supports 1,000 per table)
- MongoDB does not support clustered indexes.

Index Introduction

- Index is a special data structure at **collection** level.
- Using indexes ensures that MongoDB only scans the smallest possible number of documents.
- Any field or sub-field of documents in a collection can be indexed.
- No more than 64 indexes per collection are possible. (MS SQL server supports 1,000 per table)
- MongoDB does not support clustered indexes.

Index Behaviour

- All indexes are B-trees in MongoDB – support both equality and range-based queries.
- Index stores items internally in order, stored by the value of field.
- Order of indexes can be ascending or descending.
- MongoDB may traverse an index in either direction.
- No scanning of any documents when the query and its projection are included in index.

Index Behaviour

- All indexes are B-trees in MongoDB – support both equality and range-based queries.
- Index stores items internally in order, stored by the value of field.
- Order of indexes can be ascending or descending.
- MongoDB may traverse an index in either direction.
- No scanning of any documents when the query and its projection are included in index.

Index Behaviour

- All indexes are B-trees in MongoDB – support both equality and range-based queries.
- Index stores items internally in order, stored by the value of field.
- Order of indexes can be ascending or descending.
- MongoDB may traverse an index in either direction.
- No scanning of any documents when the query and its projection are included in index.

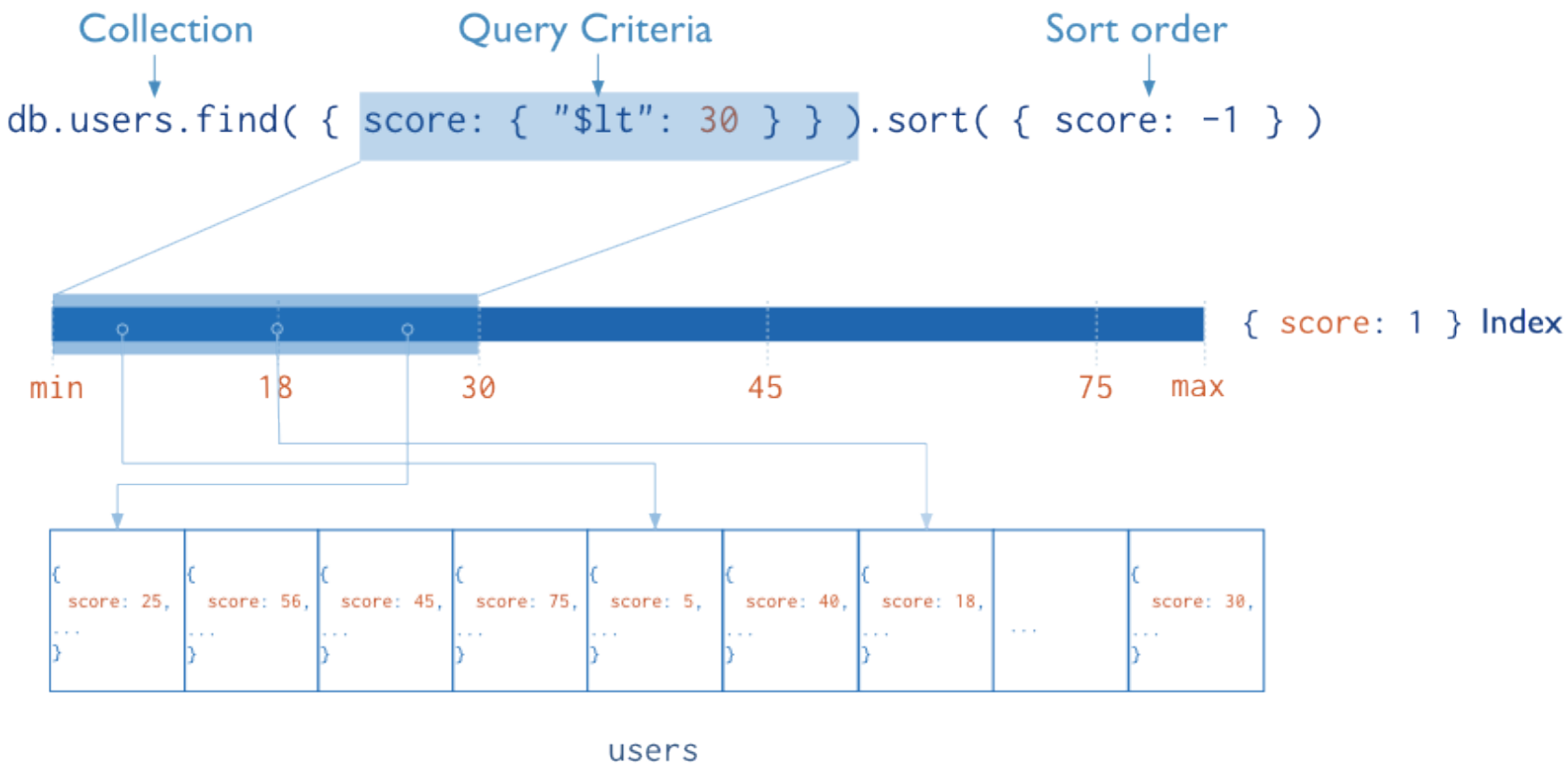
Index Behaviour

- All indexes are B-trees in MongoDB – support both equality and range-based queries.
- Index stores items internally in order, stored by the value of field.
- Order of indexes can be ascending or descending.
- MongoDB may traverse an index in either direction.
- No scanning of any documents when the query and its projection are included in index.

Index Behaviour

- All indexes are B-trees in MongoDB – support both equality and range-based queries.
- Index stores items internally in order, stored by the value of field.
- Order of indexes can be ascending or descending.
- MongoDB may traverse an index in either direction.
- No scanning of any documents when the query and its projection are included in index.

Query with the Help of an Index



Index Handling in MongoDB

Karlsruhe Institute of Technology

Building an index on a collection:

- `db.collection.createIndex()`

Creating an index if it does not currently exists:

- `db.collection.ensureIndex()`

Getting description of the existing indexes:

- `db.germancities.getIndexes()`

Having a report on the query execution plan, including index use:

- `cursor.explain()`

Forcing MongoDB to use a specific index for a query:

- `cursor.hint()`

Index Types (1)

1. Default `_id`:
   - All document have a generated index with ObjectId value for this field
   - `_id` unique index prevents users to generate two documents with same values
2. Single-Key index:
   - Supports user-defined index on a single key
   - `db.books.ensureIndex({ "author": 1 })`

Index Types (2)

Karlsruhe Institute of Technology

3. Compound Indexes
   - To support several different queries
   - If some queries use *x* key and some queries use *x* key combined with *y* key
   - `db.books.ensureIndex({ "author": 1, "price": 1 })`
   - A single compound index on multiple fields can support all the queries that search a "prefix" subset of those fields.
   - Maximum 31 fields can be combined in one compound index.
   - Two important factors play their role:
     - list order (*i.e.* the order in which the keys are listed in the index)
     - sort order (*i.e.* ascending or descending)

Compound Indexes

If the collection *orders* has the following compound index:

```
{ shell: 1, ord_date: -1 }
```

The compound index can support the following queries:

```
db.orders.find({
  shell: {
    $in: [ "B", "S" ]
  }
})
```

```
db.orders.find({
  shell: { $in:[ "S", "B" ] },
  ord_date: {
    $gt: new Date("2014-02-01")
  }
})
```

But not the following two queries:

```
db.orders.find({ }).sort({
  ord_date: 1
})
```

```
db.orders.find({
  ord_date: {
    $gt: new Date("2014-09-07")
  }
})
```

Index Intersection

- Before version 2.6, only a single index to fulfill most queries.
- The exception was just for `$or` clauses, which could use a single index for each `$or` clause.
- Intersection can employ multiple/nested index intersections to resolve a query.
- Note: Index intersection does not apply when the `sort()` operation requires an index completely separate from the query predicate.

Intersection and Sort

If the *orders* collection has the following indexes:

```
{ qty: 1 }  { shell: 1, ord_date: -1 }  { shell: 1 }
{ ord_date: -1 }
```

MongoDB cannot use index intersection for the following query with sort:

```
db.orders.find({
  qty: { $gt: 10 }
}).sort({
  shell: 1
})
```

MongoDB can use index intersection for the following query and fulfill part of the query predicate:

```
db.orders.find({
  qty: { $gt: 10 },
  shell: "B"
}).sort({
  ord_date: -1
})
```

Index Types (3)

4. Multikey Index:
   - index contents of arrays.
   - separate index entries for every element of the array.
   - Automatically generated for array fields.
5. Geospatial Index:
   - To support efficient queries of geospatial coordinate data.
   - `2d` index support two dimensional geometry queries
   - `2dspher` index support there dimensional geometry queries
6. Text Indexes:
   - supports searching for string content in a collection.
7. Hash Indexes:
   - more random distribution of values along their range.
   - just equality matches, not range-based queries

Index Properties

- Unique Indexes:
  - causes MongoDB to reject all documents that contain a duplicate value for the indexed field
  - `db.members.ensureIndex({ "usr_id": 1 }, { unique: true })`

- Sparse Indexes:
  - Only contains documents containing the indexed filed. (even if it contains a null value)
  - `db.addresses.ensureIndex({ "usr_tel": 1 }, { sparse: true })`

Covered Query

Covered Query:

- all the fields in the query are part of an index, and
- all the fields returned in the results are in the same index.

Note: Try to Create Indexes that Support Covered Queries

Covered-Query Benefits

Querying the index can be much faster than documents that are not in index, because:

- Index keys are smaller than the documents they catalog,
- Indexes are usually either available in RAM or located sequentially on disk.

Covered-Query Checking

Karlsruhe Institute of Technology

To determine covered query:

- use `cursor.explain()`
- `indexOnly: true` — covered query

Covered query cannot support:

- a multi-key index (if the indexed field is an array),
- any of the indexed fields are fields in subdocuments.

Index Strategy

Depends on:

- kinds of queries you expect
- the ratio of reads to writes
- amount of free memory on the system

Best Index Design Strategy:

Profile variety of index configurations

Traditional Schema Design

- Put the data into First Normal Form (NF1)
  - Putting data in fixed tables
  - Each cell should have just a single scalar value
  - Remove redundancy from a set of tables

Normalization Example (1)

id	name	phone_number	zip_code
1	Rick	555-111-1234	30062
2	Lisa	555-222-1234	30074
3	Sam	555-333-1234	30006

Normalization Example (2)

id	name	phone_numbers	zip_code
1	Rick	555-111-1234	30062
2	Lisa	555-222-1234; 555-345-1234	30074
3	Sam	555-333-1234; 555-324-234	30006

Query a name for a given number:

```
SELECT name FROM contacts WHERE
  phone_numbers LIKE '%555-222-1234%'
```

Multiple Columns

id	name	phone_number0	phone_number1	zip_code
1	Rick	555-111-1234	NULL	30062
2	Lisa	555-222-1234	555-345-1234	30074
3	Sam	555-333-1234	555-324-234	30006

Query a name for a given number:

```
SELECT name FROM contacts WHERE
  phone_number0 LIKE='555-222-1234' OR
  phone_number1='555-222-1234'
```

Normalization and Join

contacts

id	name	zip_code
1	Rick	30062
2	Lisa	30074
3	Sam	30006

numbers

id	phone_number
1	555-111-1234
2	555-222-1234
2	555-345-1234
3	555-333-1234
3	555-324-234

Query a name for a given number:

```
SELECT name, phone_number FROM contacts LEFT JOIN numbers
  ON contacts.contact_id=numbers.contact_id WHERE
  contacts.contact_id=3
```

Denormalization and Performance

What is the problem?

Seek takes over 99% of the the time spent reading a row.
JOINs typically require random seeks.

MongoDB and Schema Design

- Has performance benefits of redundant denormalized format
- Complicated schema design process
- Decide if you have to Embed or Reference
- General answer to schema-design problem with MongoDB:

It depends!!!

Embedding Benefits

- Read Performance:
  - data locality: documents are stored continuously on disk
  - one seek to load the entire document
  - one round trip to the database
- Atomicity and Isolation:
  - with Mongo not supporting multidocument transactions guarantee consistency

Embedding Benefits

- Read Performance:
  - data locality: documents are stored continuously on disk
  - one seek to load the entire document
  - one round trip to the database
- Atomicity and Isolation:
  - with Mongo not supporting multidocument transactions guarantee consistency

Embedding Penalties

- Write operations may take long.
- The larger a document is, the more RAM it uses.
- Growing documents must eventually get copied to larger spaces.
- MongoDB documents have a hard size limit of 16 MB.
  - but: GridFS

Embedding Penalties

- Write operations may take long.
- The larger a document is, the more RAM it uses.
- Growing documents must eventually get copied to larger spaces.
- MongoDB documents have a hard size limit of 16 MB.
  - but: GridFS

Embedding Penalties

- Write operations may take long.
- The larger a document is, the more RAM it uses.
- Growing documents must eventually get copied to larger spaces.
- MongoDB documents have a hard size limit of 16 MB.
  - but: GridFS

Embedding Penalties

- Write operations may take long.
- The larger a document is, the more RAM it uses.
- Growing documents must eventually get copied to larger spaces.
- MongoDB documents have a hard size limit of 16 MB.
  - but: GridFS

Embedding Penalties

- Write operations may take long.
- The larger a document is, the more RAM it uses.
- Growing documents must eventually get copied to larger spaces.
- MongoDB documents have a hard size limit of 16 MB.
  - but: GridFS

Where to Use Linking?

- Referencing for flexibility:
  - if your application may query data in many different ways
  - if you are not able to anticipate the patterns in which data may be queried

Embedding *vs.* Linking

Decide based on more READs or more WRITEs that your application may have.

Polymorphic Schema

- **Schemaless**: not enforcing a particular structure on documents in a collection
- **Polymorphic schema**: all documents in a collection has similar, but not identical structure

Polymorphism

- Support Object-Oriented Programming
- Enable Schema Evolution
- Its major drawback is storage efficiency

Databases in Web Applications

A Bit of History

The LAMP stack:

- **L**inux, **A**pache, **M**ySQL and **P**HP
- Established work horse of Web applications worldwide
- Free and Open Source
- Reasonably fast
- Simple to deploy and maintain

but

- Three different languages: JavaScript, PHP, SQL
- Data always as tables
- Problems scaling to big-data applications

Databases in Web Applications

Enter MongoDB

- Alternative solution: the MEAN stack
  - **M**ongoDB
  - **E**xpress — Node.js Web-application framework
  - **A**ngularJS — JavaScript Model-View-Whatever framework
  - **N**ode.js — high-performance server platform for Web applications
- Flexible data structure
- Optimised for big data and high load
- *One* language throughout the stack
- Free and Open Source

MongoDB and Node

Karlsruhe Institute of Technology

- Official native Node.js driver since 2012
- Several higher-level wrappers. Examples:
  - Mongoose — official, extensive Object Document Mapper
  - **Mongoskin** — lightweight, schema-less, similar to *pymongo*
    - usage similar to the `mongo` shell
    - asynchronous: methods take callback functions

Example: MongoDB REST API

- Problem: cannot talk to MongoDB from Web browser
- Solution: provide a *REpresentational State-Transfer* API
- Intermediate Node.js Web server
- Mongoskin to talk to the database
- Express to provide REST plumbing
- Client I/O with JSON

Example: MongoDB REST API

- Goal: map HTTP requests for URIs like `http://myMongo/collections/myColl/docId` to MongoDB CRUD
- With Express, just a few lines of code:
  - initialisation
  - include JSON body parser
  - define handlers for appropriate URIs and HTTP verbs
  - start listening for requests!

Example code: `http://wiki.scc.kit.edu/gridkaschool/upload/8/85/MongoREST-exampleCode.zip`

Replication

Karlsruhe Institute of Technology

Database Replication ensures:

- redundancy
- backup
- automatic failover

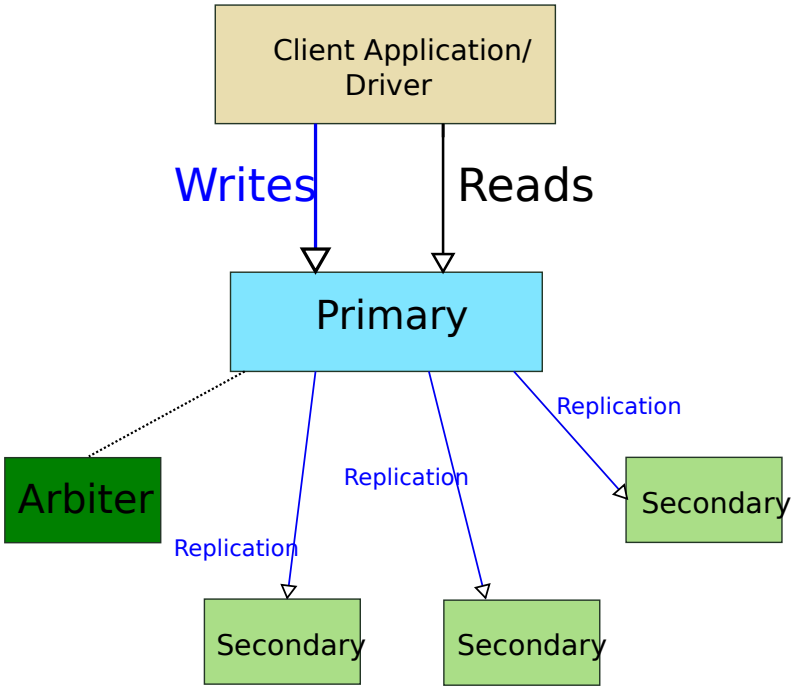
Replication occurs through groups of servers known as replica sets.

Replica Set Members

- **Primary/Master:**
  - The only node that accepts writes
  - The default read node
- **Secondary/Slave**:
  - The read-only members
  - Replicate from primary
- **Arbiter**:
  - Doesn't replicate a copy of data set
  - Cannot become Primary
  - May be used to add votes for reelection process

Primary and Oplog

- Primary saves all write operations in its Oplog
- Oplog is a capped collection which records all of the modifications
- Oplog by default usually 5% of disk space
- Secondaries copy the Oplog from Primary asynchronously



Secondary and Read Preferences

Karlsruhe Institute of Technology

Read Preferences:

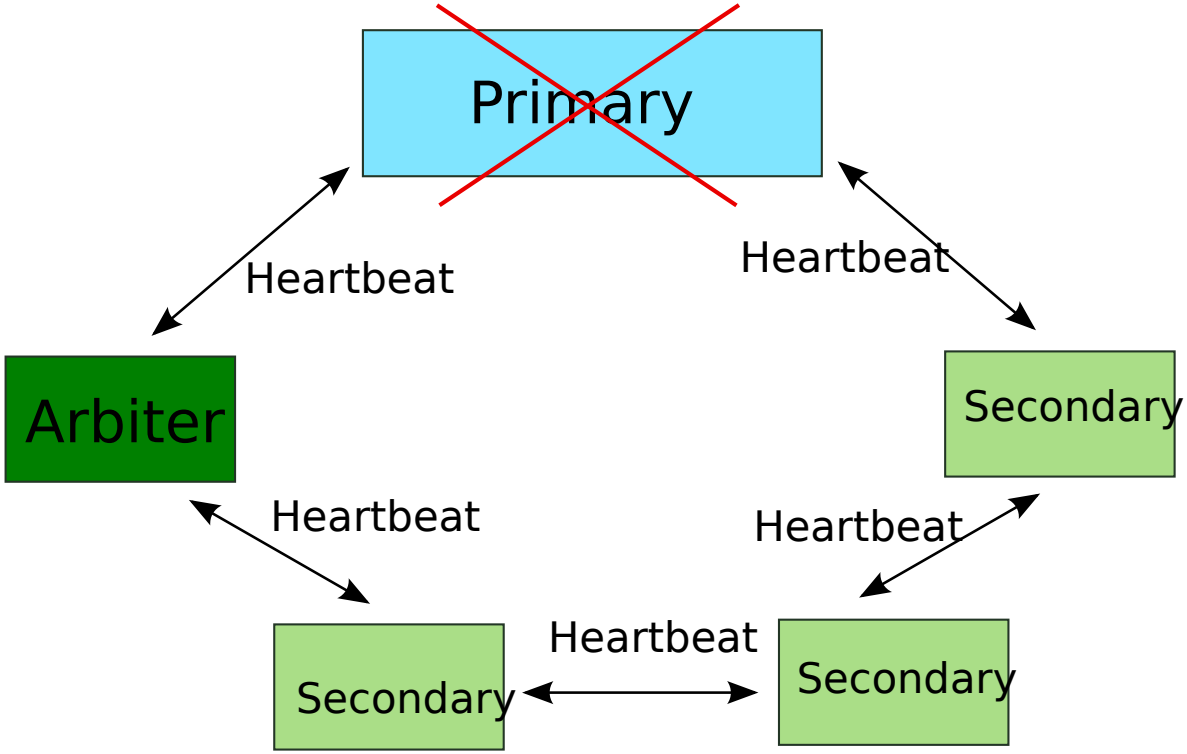
- Primary
- PrimaryPreferred
- Secondary
- SecondaryPreferred
- Nearest

Client applications can be configured with a Read Preference, on a:

- per-connection,
- per-collection,
- or per-operation basis.

Election Process

Election starts every time there isn't any Primary.

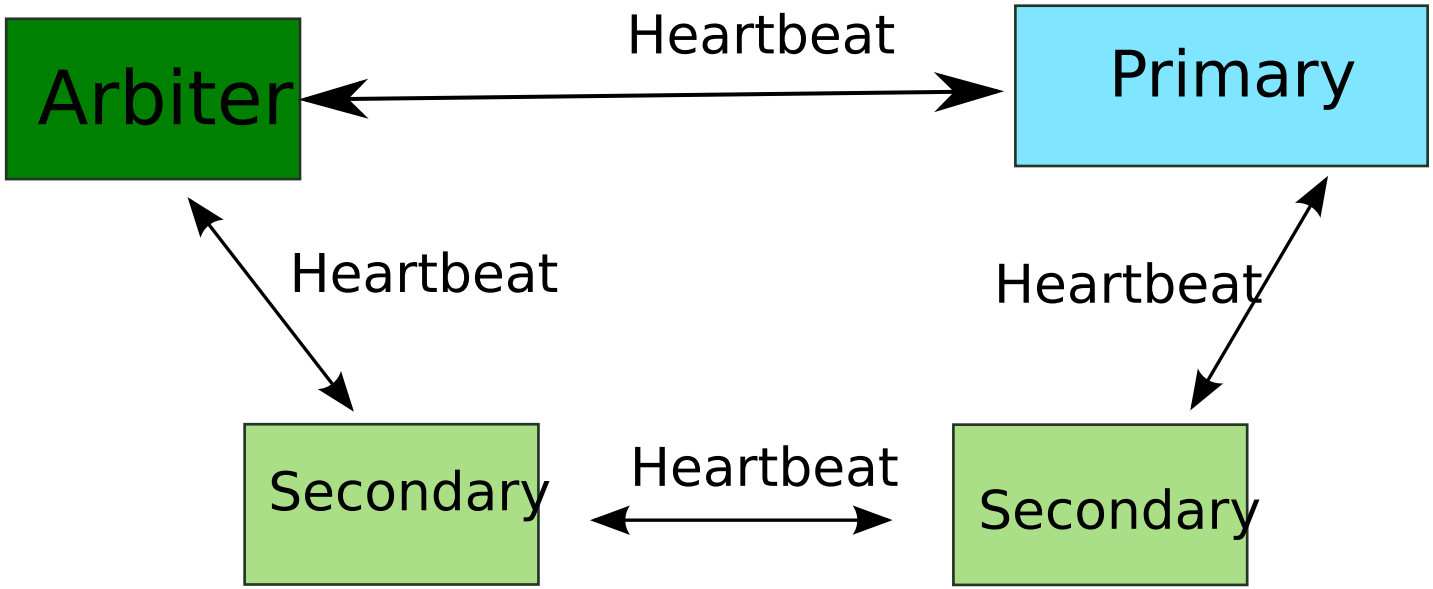


Effective Factors

- **Priority**:
  - Members will vote for the node with highest priority
  - Set priority to 0 to prevent Secondary to ever become Primary
- **Optime**:
  - Timestamp of last operation applied from Oplog for each member
  - The member with the most recent optime from all visible members will get elected
- **Connections**:
  - Primary candidate must be able to connect with majority of the members
  - Majority: total number of accessible voting members
  - Choose number of Replica set members by considering fault tolerance.

Arbiter and Election

- Maximum seven voting Members.
- Use Arbiter to get odd number of members.



Primary/Secondary *vs.* Master/Slave

Karlsruhe Institute of Technology

Primary/Secondary is the preferred MongoDB replication method.

- Primary/Secondary cannot support more than 12 nodes.
- Master/Slave doesn't support automatic failover.

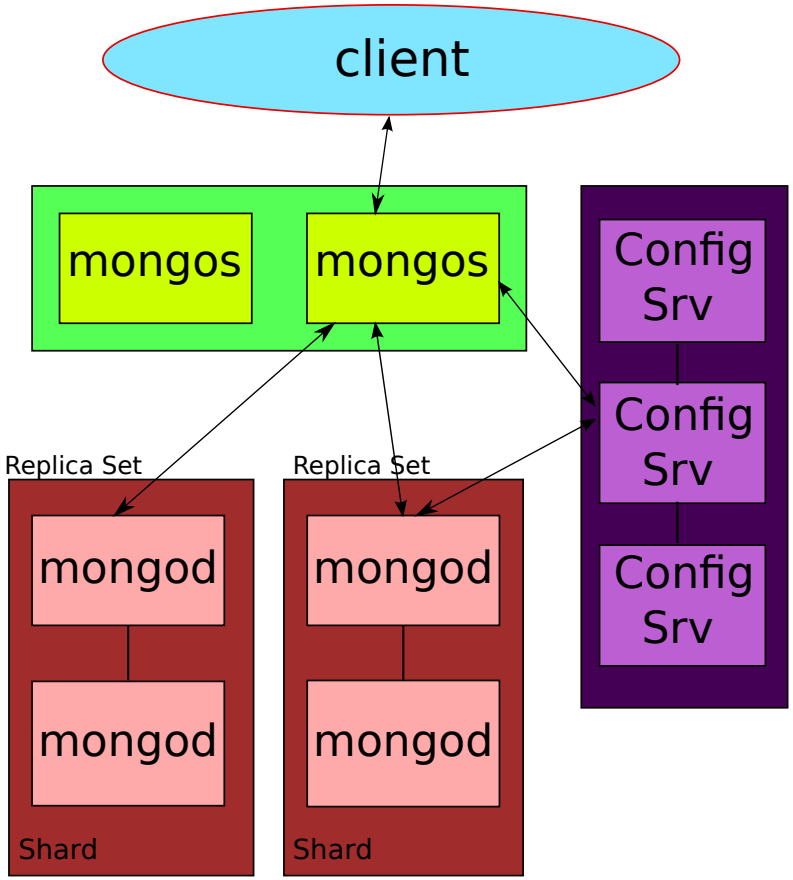
Introduction to Sharding

- Automatic horizontal partitioning and management
- No downtime is needed to convert to a sharded system
- Fully consistent
- No code changes required
- At collection level

Sharding Components

- **Config server**: holds a complete copy of cluster metadata
- `mongos`: a lightweight routing service
- **Shards**:
  - `mongod`: the primary daemon process for MongoDB system
  - **replica set**

Sharding


client
mongos
mongos
Config Srv
Config Srv
Config Srv
Replica Set
Replica Set
mongod
mongod
mongod
mongod
Shard
Shard

References

1. https://www.mongodb.org/
2. MongoDB Applied Design Patterns, Rick Copeland, 2013
3. Making Sense of NoSQL, Dan McCreary and Ann Kelly, 2013