Datenbank Implementationen

MongoDB

at the Cooperative State University Baden-Württemberg Stuttgart

by

Sedi Oben-Torks, Alexander Schmalz, Sascha Traub, Christop Walter

April 2017

Contents

1 MongoDB

This chapter guides you through the features of MongoDB. Whereby we go through what is MongoDB in general, Data model and CRUD Principles. Thereafter we check through the Use-Cases of MongoDB and performance and limitations. Additionally, we get a short Insight how MongoDB is used in a Big Data context. After all, we come up with a conclusion.

1.1 What is MongoDB ?

MongoDB is a part of NoSQL family and belongs to the document-oriented databases. Therefore, it doesn't have the concepts of tables, rows and columns. Instead, MongoDB is built on an architecture of collections and documents. Documents contain sets of key-value pairs like JSON and are the basic unit of data in MongoDB. A collection includes a set of documents and offers the same functionality as relational database tables(Banker, 2016)

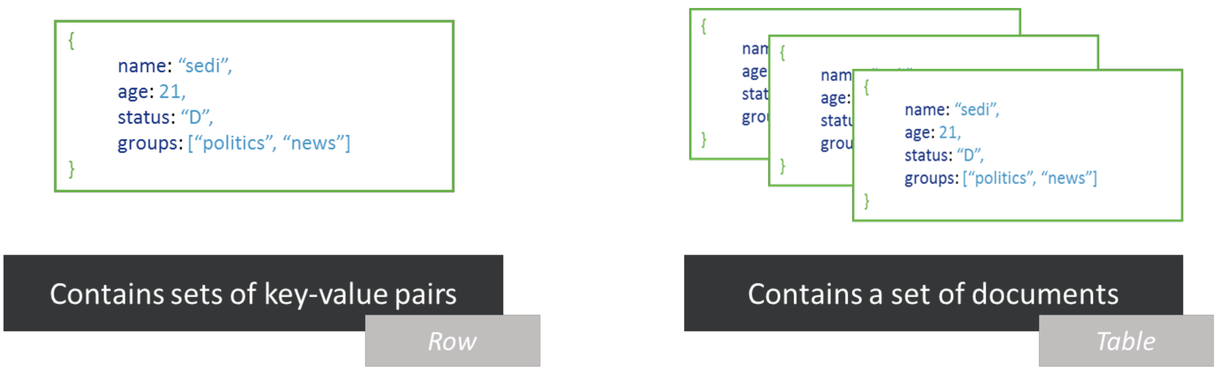

Figure 1.1: Documents and collections

MongoDB stores data as Binary JSON documents also known as BSON. The documents can have different schemas, which means that the schema can change dynamically as the application evolves. Automatic sharding enables data in a collection to be distributed across multiple systems for horizontal scalability as data volumes increase(Sabharwal & Edward, 2015). Additional, MongoDB doesn't only support Key-Value operations. It

allows complex queries, aggregations and secondary indexes that unlock the value in structured, semi-structured and unstructured data. One of MongoDB's major features is the support for many types of queries like text search, range queries, geospatial queries over to MapReduce queries(a. M. W. P. MongoDB Inc., 2013b).
MongoDB was created by Dwight Merriman and Eliot Horowitz, who had encountered development and scalability issues with traditional relational database approaches while building Web applications at DoubleClick, an Internet advertising company that is now owned by Google Inc. The database was released to open source in 2009 and is available under the terms of the Free Software Foundation's GNU AGPL Version 3.0 commercial license. However, the database is one of the most popular NoSQL databases and is used by several company like Bosch, Facebook, Expedia and so on(Plugge, Hows, Membrey, & Hawkins, 2015).

1.2 Use Cases: What is MongoDB for?

This section will give you an insight between the features and potential of MongoDB and some problems that is suited to solve.
Beginning with online and mobile Apps. Nowadays Companies want their business on their smartphones or access it from everywhere through the web. In comparison to RDMBS MongoDB addresses the upcoming challenges of these plans. Furthermore, MongoDB promises to make it easier than other alternatives. Requirements for going mobile or online are hard to manage. For example, different types of device like smartphone or wearables are creating new types of unstructured and semi-structured data. Another reason is the number of devices and users. Meanwhile, Response times must keep and provide the same User experience. So, Scalability has now a high priority. MongoDB tries to decrease the degree of difficulty for these Requirements. That is why MongoDB offers a flexible data model and rich query functionality. Therefore, MongoDB can manage any kind of data, no matter how dynamically the data changes. In addition to that, MongoDB's development concentrates on scalability and can handle a lot of Users and data sets(a. M. W. P. MongoDB Inc., 2013b).
Another Example for a suited use case is a catalog. Mention that almost anybody knows some requirements a Catalog must meet. Deleting, creating and changing items or their features or their attributes – only to mention few - are a standard set of Requirement of a catalog. Behind the scenes, we see a lot of challenges for a RDMS. There will be a lot of changes in the data, like new data and new metadata to your catalogs. We already talked about the untrusted and semi-structured in the first Use case. Again, we got the same problems with a RDBMS. How does MongoDB make it easier for developers? First, it is how the data is structured in MongoDB. With MongoDB's JSON document model

makes it easy to store different assets with different attributes in a single place. It also makes it simple to represent complex, hierarchical relationships. Schemas in MongoDB are self-describing. You can add new products and features and evolve the schema instantly, without taking the database down or impacting performance. Lastly an expressive query language, indexing, including text search and geospatial, and analytics provide flexible access to the data, no matter how the application, business or developer needs to find it(a. M. W. P. MongoDB Inc., 2013b).
All in all, these are only few examples for use cases and for what MongoDB is. In general MongoDB claims to be suited for high flexible data schemas to provide the ability for data changes, structed, unstructured and semi structured data. Additionally, MongoDB eco system let developer spend less time for the design of models, entities, relationships and tables, and more time on the application(a. M. W. P. MongoDB Inc., 2013b).

1.3 Datamodel

Before designing a database, it is crucial to analyse the data and the requirements of the application. In contrast to relational databases there is no strongly recommended way to structure your data, like for example a normalized data model to avoid inconsistencies on updates. However, there are well established patterns which help developers to create their data model (Banker, 2016). Later in this chapter, there will bw some examples for common design patterns. But at first, the following paragraphs will concentrate on the data concepts of MongoDB in detail.

1.3.1 Databases and Collections

As mentioned in the introduction, MongoDB is structured in databases, collections and documents. MongoDB provides a mongo shell to communicate with the database. The code snippets provided in this chaptered can be performed on the mongo shell (MongoDB Inc., 2016a).

A Database can hold collections of documents. The creation of a database occurs automatically when the first document is pushed to a collection of this database. Thus, there is no command to create an empty database (MongoDB Inc., 2016a). The following commands show how simply it is to select and create a new database by inserting its first document.

```
use newDatabase
db.newCollection.insertOne( { value: 2 } )
```

Listing 1.1: Create Database

This example also shows the process of creating a new collection. As the database, the collection is created as soon as the first document got inserted. Certainly, there is also a function to explicitly create a new collection with an option to pass parameters affecting the collections behaviour. In the example in Listing 1.2, a collection with limited number of documents is created. All options can be found in the MongoDB documentation (MongoDB Inc., 2016a).

```
db.createCollection( 'newCollection', { max: 1000 } )
```

Listing 1.2: Create Collection

1.3.2 Documents

MongoDB stores documents in the BSON format. This format supports multiple datatypes. Many of them are common in the information technology, such as Double, String or Boolean. A full list would go beyond the scope of this book, but can be found in the BSON specification. In general, there are all types needed for object oriented programming (bsonspec.org, 2017). The chapter Queries and CRUD operations will treat the way BSON types can be used to query through documents. But before that, the concept of documents will be explained.

1.3.3 Embedded Documents and Referencing

The below example in Listing 1.3 shows how a document is structured. The field name _id is reserved to be used as primary key. It has some special characteristics like being unique in the collection and immutable. To make sure this field is unique, it is recommended to use a unique ObjectId. If there is no value submitted with the document, an ObjectId is generated automatically (MongoDB Inc., 2016a).

```
var newDocument = {
  _id: <ObjectId>,
  dateOfBirth: new Date('Jan 01, 1990'),
  name: {
        last:  'Doe',
        first: 'John'
    },
  contact: {
    phone: '12345678',
    email: 'john@doe.com'
  }
}
```

Listing 1.3: Embedded Documents

Other fields can be added as needed. For example, a field containing the date of birth of a person. Or an object containing the first and last name and another object with contact details. This concept of containing objects inside a document is called embedded sub-documents and has its own strength and weaknesses (MongoDB Inc., 2016a). It may be resulting in a performance growth if the application always needs the document with its whole embedded data. But one common pattern for MongoDB says to consider whether embedded data or referencing is more suitable for the given situation (Banker, 2016). By referencing, the embedded data is stored in a separate document with a reference to its related document. This concept is the foundation for creating normalised data models. How this can be implemented for the document with John Doe can be seen in the code snippet in Listing 1.4.

```
var newPerson = {
  _id: <ObjectId1>,
  dateOfBirth: new Date('Jan 01, 1990'),
}

var newName = {
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
    last: 'Doe',
    first: 'John'
}

var newContact = {
    _id: <ObjectId3>,
    user_id: <ObjectId1>,
    phone: '12345678',
  email: 'john@doe.com'

}
```

Listing 1.4: Referencing Documents

When deciding whether referencing is reasonable, the atomicity of write operations also influences this decision. In MongoDB atomicity is only guaranteed on document level. As no single write operation can affect more documents, a normalised data model cannot be updated with one atomic operation. At this point denormalised data models have benefits over normalised ones. However, another design pattern for MongoDB is to overthink your overall database selection if you need atomic, consistent, isolated and durable operations, also known as ACID principles (Banker, 2016).

1.3.4 Validation

MongoDB comes with a way to validate documents by itself. There are different ways to handle documents which do not match its validation criteria. By default, invalid documents are rejected with an error. A collection, which would check if the new document contains a date of birth, would look like the following in Listing 1.5. Furthermore, it is also possible the check the fields for a certain data type (MongoDB Inc., 2016a).

```
db.createCollection( 'users', {
  validator: {
      $or: [ {
          dateOfBirth: { $exists: true }
      }
        ]
    }
}
```

Listing 1.5: Validation for Collections

1.4 Queries and CRUD operations

MongoDB has several queries and operations to manage its data. This chapter gives a short overview of the most important operations.

1.4.1 Create (Insert)

Before searching for or working with data, the database needs information that can be worked with. How documents can be created was introduced while treating how to create a new database. This command inserts one document into a collection and can be found again in the Listing 1.6. But there are also commands to insert an array of documents at once (Membrey, Hows, & Plugge, 2014).

```
db.newCollection.insertOne( {name: 'John', age: 30} )
db.newCollection.insertMany( [
    {name: 'Max', age: 20},
    {name: 'Marie', age: 25}
] )
```

Listing 1.6: Create (Insert)

1.4.2 Read (Find)

To find data stored in a MongoDB, operations to either find one or multiple documents can be used. This example in Listing 1.7 shows both operations. The difference is that the second operations stops after the first result and it is generally advised if only one result is excepted. Of course, the results can also be filtered by field values as shown in line three, or sorted by values as shown in line four (Membrey et al., 2014).

```
1 db.newCollection.find()
2 db.newCollection.findOne()
3 db.newCollection.find( { 'name.last': 'Doe' } )
4 db.newCollection.find().sort( { 'name.first': 1 } )
```

Listing 1.7: Read (Find)

1.4.3 Update

MongoDB also provides a function to update documents with three input parameters: the search criteria, the new object and options. It is important to understand that any fields that are not part of the new object are also removed from the old object, as it were completely rewritten. The option upsert implies to add any fields that do not exist yet. It is also possible to update multiple options that match the search criteria with one command (Banker, 2016). An example for a basic update operation can be found in Listing 1.8.

```
db.newCollection.update(
    { name: 'John' },
    { age: 21, name: 'John', lastname: 'Doe'  },
    { upsert: true } )
```

Listing 1.8: Update

1.4.4 Delete

The last operation of CRUD examines the deletion of documents, collections or whole databases. The remove operation, as shown in line one of the code snippet in Listing 1.9, removes all documents which match the criteria. The _id field can help to be sure to only delete the right document. It can lead to conflicts if a documents is deleted without considering its references, because references from other documents will not change automatically (Membrey et al., 2014). Line 2 shows how to delete a collection and line 3 shows how to delete an entire database.

```
db.newCollection.remove( { age: 20 } )
db.newCollection.drop()
db.dropDatabase()
```

Listing 1.9: Delete

1.5 Architecture

1.5.1 Core Processes

The MongoDB server package comes up with three main processes:

- mongod
- mongo
- mongos

Mongod is the core database server. Once started, the *mongod* listens by default on port 27017 for requests. It also manages the data format and performs all background operations. For administration the *mondod* provides an HTTP interface, witch can be reached at the localhost on port 28017 (1000 higher than the default port). The data directory *mongod* connects to is by default C:\data\db (or /data/db). This directory definitely has to exist and the default ports have to be free, otherwise the process fails to start (Sabharwal & Edward, 2015).
The *mongo* process is an interactive MongoDB shell. It gives the user the possibility to communicate with a running *mongod* process via a JavaScript like query language. If running on the same host, it automatically connects to the *mongod* process and a preinstalled test database (Sabharwal & Edward, 2015).
The *mongos* process is working like a routing service and is the basis for MongoDBs sharding ability that will be described later (1.5.3). It holds the information about where the requested data is located and forwards the request from an application server to the right destination (Sabharwal & Edward, 2015).
By running a *mongod* process you already have a standalone deployment of MonogDB that can be accessed by a client. But in case of failure there is no redundancy or recovery, that prevents data loss, so its not recommended to use this in a production environment. To avoid this, replication is used to guard against hardware failure or database corruption. It also gives the possibility to perform normally high-impact maintenance with little or no impact (Plugge et al., 2015).

1.5.2 Replication

„MongoDB supports the replication of a database's contents to another server in real time or near real time“(Plugge et al., 2015, p. 285). For that MongoDB provides two different replication methods. The traditional *Master/Slave Replication* and *Replication Sets*.

Master/Slave Replication

In a Master/Slave setup one *mongod* instance acts as a master, the others declared as slaves. All write and read operations are requested to the master and the slaves replicate the data of the master, but can't be requested by a client. If a failure occurs that forces the master to go down, the hole system can't be reached anymore. The data, till the last replication to the slaves, is saved, but can't be accessed until the master comes up again. In MongoDB the master holds a capped collection called *oplog*. The *oplog* is an ordered history of all logical writes, that are executed within a defined time period. The operations stored in the *oplog* in an idempotent way, so they can be performed multiple times without changing the result (Sabharwal & Edward, 2015). That's useful when a slave runs into failure while executing the operations onto its data. In that case the replication process can be simply restarted. If the slaves syncing process last to long or the slave was down for a longer time, the oplog data could be deleted before the slave was able to synchronize (Sabharwal & Edward, 2015). In that case the slave has to start a resync process. To avoid such a situation, the oplog length should be chosen in consideration of the slaves performance.
The configuration of MongoDB with a *Master/Slave Replication* is only recommended for more than 50 nodes (Sabharwal & Edward, 2015). At that point the next described replication method, the *Replica Set*, is reaching its limits, because the communication overhead becomes to big.

Replica sets

In contrast to the *Master/Slave Replication*, in a *Replica Set* no fixed master is defined. Instead the nodes are declared as primary or secondary. Each node in a *Replica Set* can become primary, but there is only one primary at a time, the others are secondaries. All write operations going through the primary, but read operations can also be performed by secondaries (Sabharwal & Edward, 2015). The replication process works just like it does in a *Master/Salve Replaction*, but if a primary goes down a new primary is elected out of the secondaries.

Communication

All nodes in a *Replica Set* communicate with each other. As life sign they are sending a heartbeat signal to each node and getting back status replies of each node. Those replies contain information about the node, such as is he primary or secondary and what type of node he is. Each node can be assigned a certain number of votes and a priority.
This results in a various types of nodes:

- **normal secondaries:** hold a copy of the primaries data, accept read requests and are primary candidates
- **priority-0-members:** secondaries that will never become primary
- **hidden-members:** priority-0-members that can't serve read request, because they are hidden for the client
- **delayed-members:** have a delay in synchronization to prevent human failure
- **arbiters:** don't hold data, they just solve ties in a election process
- **non-voting-members:** normal secondaries, but they can't vote

If the primary recognizes that the heartbeat of a secondary has stopped, he has to check if he still can reach the majority of the set. If it can't he demotes itself to secondary and starts a election process. Also the election process is started if a secondary recognizes the primary is down. All voting nodes now collect the for the election required information form the primary candidates. The election of a primary depends on various parameters. Important is, that the elected node has the most recent data of all nodes. The candidate with the most votes is promoted to primary. When the old primary comes up again, he will be a new secondary (Sabharwal & Edward, 2015; Plugge et al., 2015).

Write Concerns and Read Preferences

MongoDB provides two important configurations to regulate consistency and availability. With *write concerns* it's possible to configure a minimum number of secondaries, that have to replicate the data, before the client gets the success response. Figure 1.2 shows how a write process would run, by configuring a minimum of one secondaries.

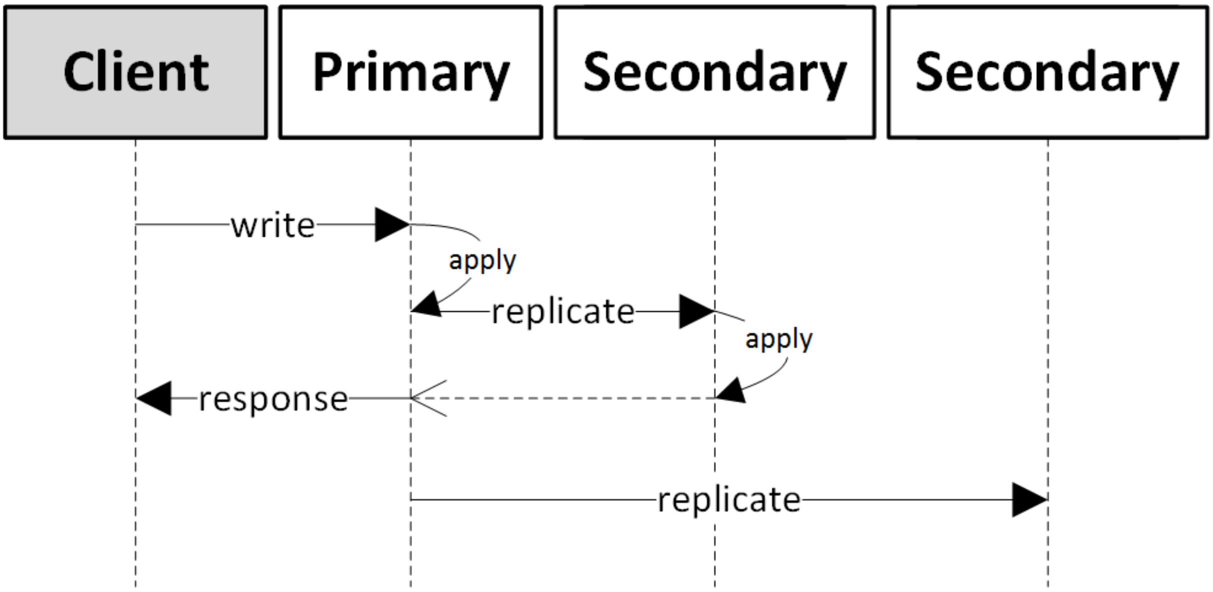

Figure 1.2: Write process with write concerns

Read preferences allow the administrator to route read operations. They determine from which node a client is allowed to read. MongoDB supports five read preferences:

- **primary:** all read operations are requested to the primary node
- **primaryPreferred:** read operations are requested to secondaries, if the primary is unavailable
- **secondary:** read operations are requested to secondaries
- **secondaryPreferred:** read operation are just requested to primary, if no secondary is available
- **nearest:** read operations are requested to the node with the lowest network latency

(MongoDB Inc., 2016a)

1.5.3 Sharding

If the amount of data exceeds the capacity of a single database server, partitioning is needed to distribute the data on multiple servers. For MongoDB this ability is even more important, because it uses memory mapped file I/O to access its underlining data storage (Plugge et al., 2015). MongoDB uses a horizontal partitioning mechanism called *sharding*. The data collection gets divided and distributed onto multiple servers called shards. Every shard is an independent database managed by multiple *mongod* processes. All the shards are combined to one logical database. The partitioning and routing are managed by the earlier mentioned *mongos* (1.5.1) process. All write and read requests of an application are send to a *mongos* process, that holds the information where the requested data is stored and forwards the requests to the responsible *mongod* processes. The data is distributed based on a configured *shard key* and chunk size. The metadata of a sharded cluster is stored on special config servers, where the *mongos* processes can obtain the routing information (Sabharwal & Edward, 2015; Plugge et al., 2015).

1.5.4 Summary

Figure 1.3 describes one possible deployment architecture, that contains all in this section mentioned artifacts. Clients can connect to a *mongos* process, running on an application server. This process forwards the requests, based on the information stored on the *config servers*, to the right shard. A shard is an replica set, containing several *mongod* processes. The *shard key* in this example is the year. So Shard-1 contains all data from 1999 til 2009 and Shard-2 contains the data from 2009 til now.

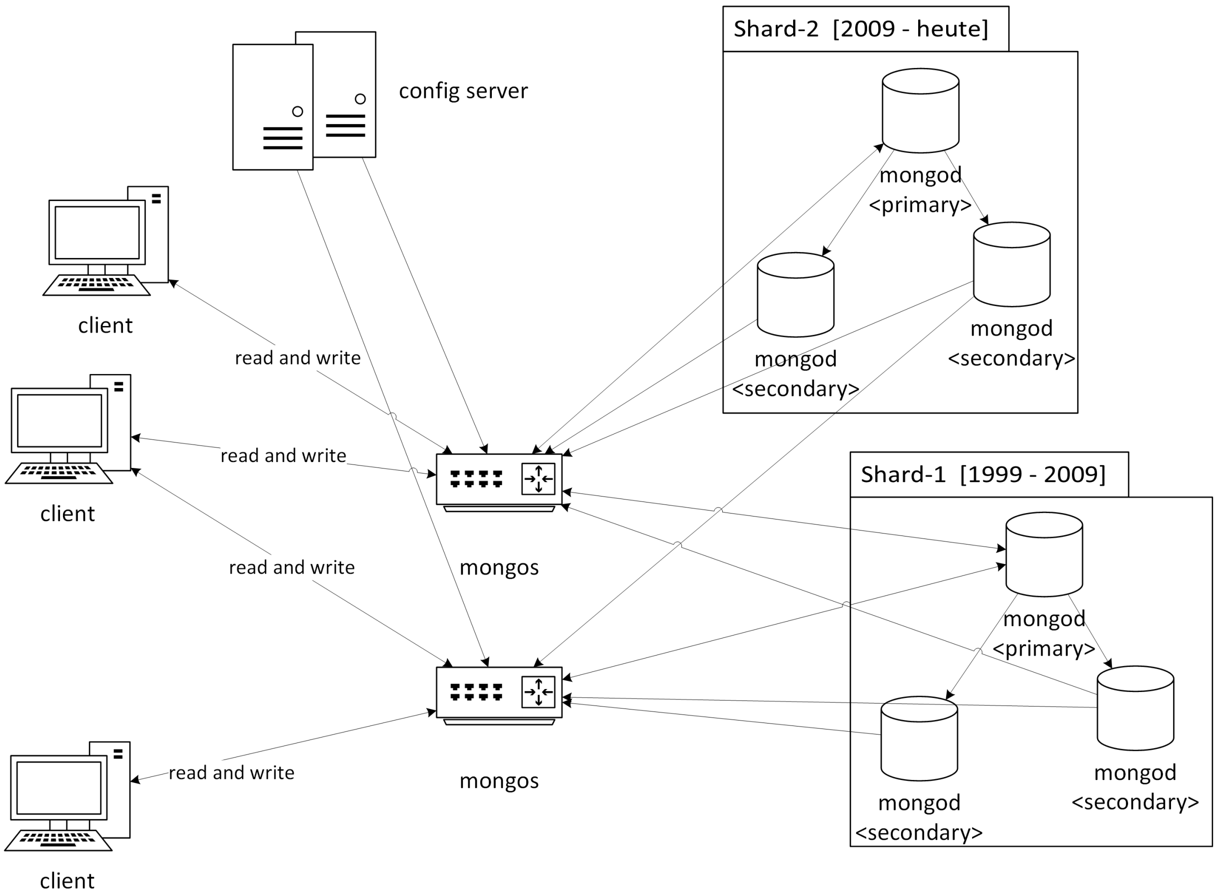

Figure 1.3: MongoDB Architecture Example

1.6 Performance

1.6.1 Performance measurement introduction

Before a performance measurement or comparison between database systems can be made, it is necessary to define the performance indicators. Depending on the use-case of the measured databases the results can be completely different. For example a real-time system is a lot more dependent on access times, latency and fast updates for concurrent users. A scientific database has to be fast at processing complex queries with joints and preprocessing routines like aggregation (O'Neil, 1997). Of course the benchmark should be implemented to test the performance of a database system in a way that reflects your usage in the future.

For database systems there is a council called the TPC . This organization tests different database systems on physical and virtual machines and scores them by performance. The performance indicators can be seen on their website and there are different benchmarks depending on the use-case of the system (Tpc.org, n.d.).

Another standardized benchmark for database implementations is the Wisconsin Benchmark. This was one of the first standardized benchmarks and was made for relational databases. The test contains of inserts, selections, joints and projections. For further details on this benchmark, see the paper (DeWitt, 1991).

This paper will summarize MonogDB performance compared to other popular NoSQL and SQL databases.

1.6.2 Influencing factors to database performance

All the results provided by this paper are dependent on the underlying hardware used. Depending on the host system for the database application, the performance can vary a lot (Lee, 2009). Examples for big performance factors are available memory, processor speed and the used storage.

For evaluation which database system should be used, it is important to know on what kind of hardware the production system will run. This is due to the fact, that different database systems were developed to meet certain performance goals on different hardware. Therefore some databases scale well with more memory and memory bandwidth since they try to cache a lot of the data in system memory. But there are also databases which try to be very lightweight on memory for low end systems or massive I/O operations (Boncz, 1999).

Nearly all databases are reliant on the used storage. This storage is the only way to keep the data, even when the system is turned off. Even In-memory databases use the persistent storage to save their current data (Wang, 2001). As transactions ultimately have to be written to the storage, this becomes the biggest bottleneck. With traditional HDDs having a high latency when accessing random data, which happens frequently on a database system, the introduction of SSDs eliminated those problems. Benchmarks done on traditional HDD storage can be translated most of the time to SSDs since all databases will perform better but should stay at the same relative performance.

1.6.3 NOSQL compared to SQL databases

It is often one of the first question, when talking about database performance. SQL or NOSQL, which one is faster? Comparing those two types of databases generally, is not really useful. The way how data is stored is completely different and the use-case defines if SQL or NOSQL fits better.

A good example for the difference of these two types is Twitter. They use a NOSQL database for all the tweets. Although Twitter is using MySQL heavily. The reason for this is easy to explain. A tweet contains some sort of content like text or images and a lot of additional information like hashtags, user and topics. Storing that data in a relational, normalized way, it would take a lot of tables and joins to represent a tweet . All hashtags would be referenced over multiple tables. These joins take time and since Twitter is a real time platform, latency should be low (Weil, 2017).

In contrast the NOSQL implementation is way easier. The complete Tweet is stored as a document. The hashtags and links are stored in the same document in a key value manner. Now when you want to read 100 Tweets, in a NOSQL context, this can be done by one simple query. In a relational model, complex joins for each Tweet would be necessary. In such a scenario performance is obviously on the NOSQL side, but only because the use-case is well suited for non-relational models.

1.6.4 MongoDB performance

This paper will be limited to benchmarks of low level functions for databases. These functions are: reads, writes and deletions. Performance is measured between MongoDB and MariaDB. MariaDB is a SQL database and as previously mentioned, general comparisons between NOSQL and SQL are not useful. In this case two implementations of SQL and

NOSQL are compared which is relevant when the use-case can be implemented by both designs without drawbacks. In such a scenario, raw performance is a valid factor. In addition to the benchmark implemented by the author of this section, another one is used for validity and a broader overview. The referenced paper contains additional databases.

The following performance test were performed, using NodeJS. For MongoDB connectivity the official MongoDB driver was used. The same applies for the MariaDB driver . The driver selection can cause huge performance differences. There are several MariaDB drivers for NodeJS and of course other programming languages. Therefore comparisons of database performance should be done, using the same programming language (Mscdex, n.d.). The inserted data contains just an id represented by an integer.

Figure 1.4: Insert MongoDB vs MariaDB

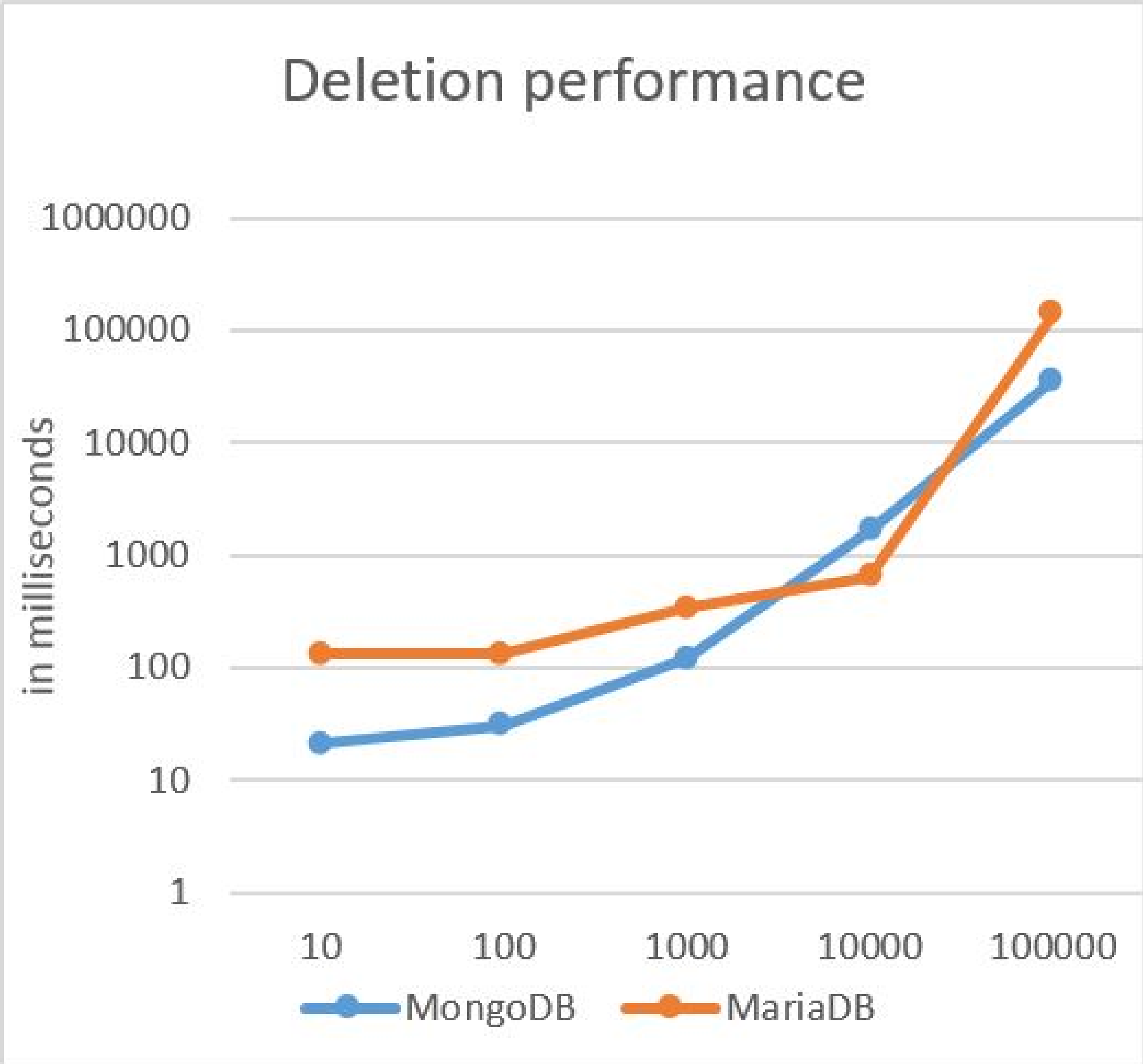

Figure 1.5: Deletion performance

The numbers show an interesting result. In the insert test MongoDB performs worse than MariaDB just slightly up to the point of 1000 inserts. After that point MariaDB falls behind. With increasing number of inserts the performance leap of MongoDB increases a lot. A reason for this could be some kind of overhead for inserts on MongoDB, further investigation is needed to evaluate the results and the cause. A similar behavior can be seen on deletion test.

The following performance results are provided by the paper (Li & IEEE, 2013)

Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	8	14	23	138	1085	10201
RavenDB	140	351	539	4730	47459	426505
CouchDB	23	101	196	1819	19508	176098
Cassandra	115	230	354	2385	19758	228096
Hypertable	60	83	103	420	3427	63036
Couchbase	15	22	23	86	811	7244
MS SQL Express	13	23	46	277	1968	17214

Figure 1.6: time for read operations in milliseconds

Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	61	75	84	387	2693	23354
RavenDB	570	898	1213	6939	71343	740450
CouchDB	90	374	616	6211	67216	932038
Cassandra	117	160	212	1200	9801	88197
Hypertable	55	90	184	1035	10938	114872
Couchbase	60	76	63	142	936	8492
MS SQL Express	30	94	129	1790	15588	216479

Figure 1.7: time for write operation in milliseconds

Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	4	15	29	235	2115	18688
RavenDB	90	499	809	8342	87562	799409
CouchDB	71	260	597	5945	67952	705684
Cassandra	33	95	130	1061	9230	83694
Hypertable	19	63	110	1001	10324	130858
Couchbase	6	12	14	81	805	7634
MS SQL Express	11	32	57	360	3571	32741

Figure 1.8: time for delete operation in milliseconds

Interestingly the benchmarks from the paper show similar results on write performance when comparing the SQL implementation and MongoDB. After 100 writes MongoDB performs better. When comparing to other NoSQL databases, MongoDB is always in the upper half of the field. So it can be said, that this database should be suited well for demanding applications and performance shouldn't be a major problem.

These statements only apply for single instance usage. Since MongoDB uses a single master architecture for multiple instances, throughput will be less than database systems that trade consistency for performance. Using sharded servers can increase performance for horizontal scaling. With this addition, MonngoDB is also very usable for scientific applications with high amounts of I/O and data (Dede, 2013).

1.7 MongoDB and Big Data

In terms of Mobile Applications, IoT, Industry 4.0 and cloud-computing, data is vast, unstructured, sometimes unwieldy and complicated. In this context, Big Data is identified by its velocity, variety and volume. Therefore, requirement and expectations has changed how to store, process and analyze data. It has led to the development of NoSQL databases such as MongoDB(a. M. W. P. MongoDB Inc., 2013a).
However, in the era of Big Data, there a 2 kind of database solutions for facing Big Data. We distinguish operational Big Data Systems and analytical Big Data solutions. Features of Operational Big Data Systems provides real-time, interactive, dynamic workloads that ingest and store data. MongoDB belongs to this category and is a popular technology for operational Big Data applications(a. M. W. P. MongoDB Inc., 2013a).
On the other hand, Analytical Big Data technologies are useful for retrospective, sophisticated analytics of your data. A most-known example of an Analytical Big Data technology is Apache Hadoop. Hadoop is designed for storing and processing large sets of data on a distributed environment based on commodity servers and storage. It is an open-source Apache project, which consists of a distributed file system called HDFS (Hadoop Distributed File System) and a data processing and execution model called MapReduce(Wadkar, Siddalingaiah, & Venner, 2014).
Choosing between operational and analytical Big Data solution isn't the right way of thinking about facing this Decision. Many organizations are harnessing the power of Hadoop and MongoDB together to create complete big data applications. At the one hand, MongoDB powers the online, real time operational application, serving business processes and end-users, exposing analytics models created by Hadoop to operational processes. At the other hand, Hadoop consumes data from MongoDB, blending it with data from other sources to generate sophisticated analytics and machine learning models. Results are loaded back to MongoDB to serve smarter and contextually-aware operational processes – i.e., delivering more relevant offers, faster identification of fraud, better prediction of failure rates from manufacturing processes(a. M. W. P. MongoDB Inc., 2013a).

In the following, you see a Figure which shows a Design pattern how to combine these two technologies to be ready for a Big Data environment:

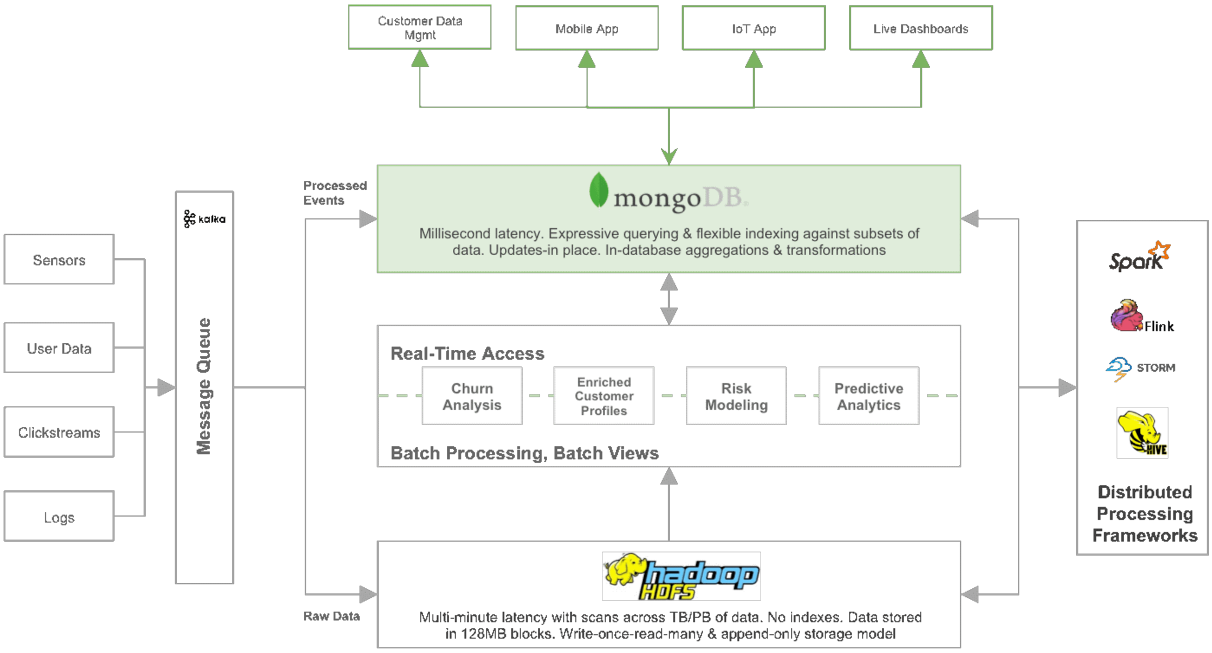

Figure 1.9: Design pattern for integrating MongoDB with Data Lake (MongoDB Inc., 2016b)

1.8 Conclusion

All in all, MongoDB is a powerful and popular documented-oriented database. It comes with a lot of features, which meets the modern-day requirements and challenges for business applications, platforms and the web. The examples about the MongoDB data model, queries and CRUD operations showed how easy it is for developers to setup and work with this database. MongoDB is also very flexible in terms of extending the structure of documents. However, it is strongly recommended to consider the usage of common patterns to create a strong and future-proof data model. The usage of patterns will also help developers to understand a given database structure and work together on larger projects. Even in a big data context MongoDB is a good choice for big data applications. That is why you will find a lot of companies and startups using MongoDB in their development.

1.8.1 MongoDB in CAP-Theorem

In a single server or Master/Slave configuration MongoDB prioritize consistency over availability. That might be the reason why in most literature MongoDB is positioned at CP-side of the CAP-Theorem. But in *Replica Sets* it is possible to trade some of the consistency for a higher availability, by configuring *read preferences* (1.5.2). By allow reading from secondaries, there is no way to ensure the client is reading consistent data. „This behavior is characterized as eventual consistency, witch means that although the secondary's state is not consistent with the primary node state, it will become consistent over time“(Sabharwal & Edward, 2015, p. 108). With *write concerns* (1.5.2) it is possible to obviate inconsistent reads happen to often, by ensure that a minimum number of secondaries is consistent.

Figure 1.10 describes how the three values change depending on the configuration. The partition tolerance is always fulfilled, because MongoDBs architecture is designed that way. By allow reading from secondaries availability will be increased at the expense of consistency and some consistency can be recovered using *write concerns*. But availability is always limited to reading, so it can never be fulfilled for all aspects of an application.

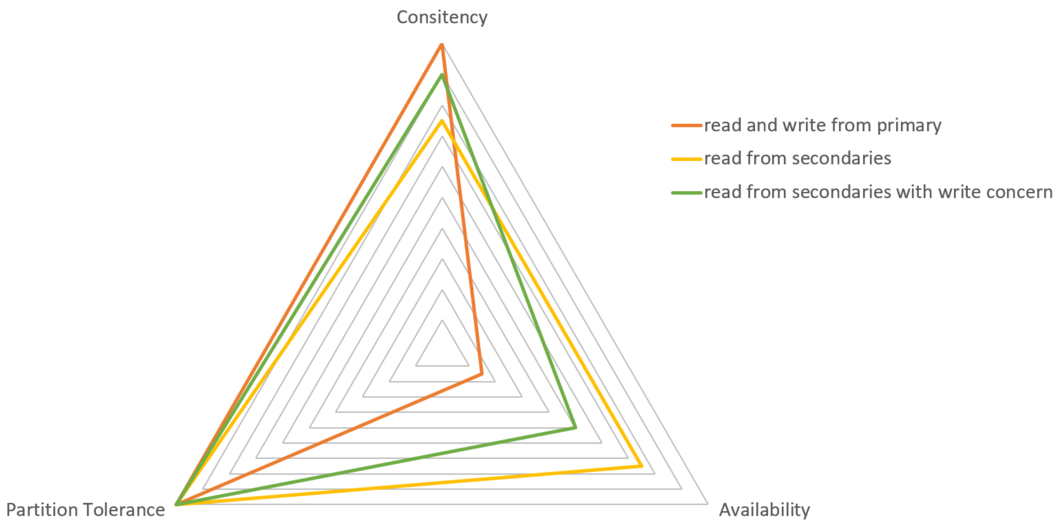

Figure 1.10: Write process with write concerns

References

Banker, K. (2016). *MongoDB in action : covers MongoDB version 3.0.* Retrieved from https://www.safaribooksonline.com/library/view/mongodb-in-action/9781617291609/

Boncz, P. A. (1999). Database architecture optimized for the new bottleneck: Memory access.

bsonspec.org. (2017). *Specification version 1.1.* (Retrieved March 23, 2017, from http://bsonspec.org/spec.html;)

Dede, E. (2013). Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. *Science Cloud 13*.

DeWitt, D. J. (1991). The Wisconsin Benchmark: Past, Present, and Future. *Computer Sciences Department*.

Lee, S. (2009). W. . Advances in flash memory SSD technology for enterprise database applications. *SIGMOD 09*.

Li, Y. S., & IEEE. (2013). A performance comparison of SQL and NoSQL databases. *Rim Conference*.

Membrey, P., Hows, D., & Plugge, E. (2014). *Mongodb basics.* Apress.

MongoDB Inc. (2016a). *The mongodb 3.4 manual.* (Retrieved March 23, 2017, from https://docs.mongodb.com/manual/;)

MongoDB Inc. (2016b). Unlocking Operational Intelligence from the Data Lake. (June). Retrieved from https://webassets.mongodb.com/mongodb_operational_data_lake.pdf

MongoDB Inc., a. M. W. P. (2013a). *Big Data: Examples and Guidelines for the Enterprise Decision Maker* (Tech. Rep.). Retrieved from http://s3.amazonaws.com/info-mongodb-com/10gen_Big_Data_White_Paper.pdf

MongoDB Inc., a. M. W. P. (2013b). *MongoDB Architecture Guide* (Tech. Rep. No. August). Retrieved from https://webassets.mongodb.com/_com_assets/collateral/MongoDB_Architecture_Guide.pdf

Mscdex. (n.d.). *Node.JS MySQL client library benchmarks* (Vol. 2017). Retrieved 23.03.2017, from http://mscdex.github.io/node-mysql-benchmarks/

O'Neil, P. (1997). Database Performance Measurement. *CRC Handbook for Computer Science and Engineering*.

Plugge, E., Hows, D., Membrey, P., & Hawkins, T. (2015). *The definitive guide to mongodb: A complete guide to dealing with big data using mongodb* (3rd ed.). Berkely, CA, USA: Apress.

Sabharwal, N., & Edward, S. G. (2015). *Practical MongoDB: Architecting, Developing, and Administering MongoDB.* Retrieved from https://www.safaribooksonline.com/library/view/practical-mongodb-architecting/9781484206478/

Tpc.org. (n.d.). *About the TPC* (Vol. 2017). Retrieved 2017-03-21, from http://www.tpc.org/information/about/abouttpc.asp

Wadkar, S., Siddalingaiah, M., & Venner, J. (2014). Pro Apache Hadoop. , 405. Retrieved from https://www.safaribooksonline.com/library/view/pro-apache-hadoop/9781430248644/ doi: 10.1007/978-1-4302-4864-4

Wang, M. (2001). Main Memory Databases.

Weil, K. (2017). Rainbird: Realtime Analytics at Twitter (Strata 2011). Retrieved 21.03.2017, from https://www.slideshare.net/kevinweil?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview