
# Datenbank Implementationen

## Das E-Book.

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

**IT14C**

April 2017

# Inhaltsverzeichnis

# 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(?, ?)

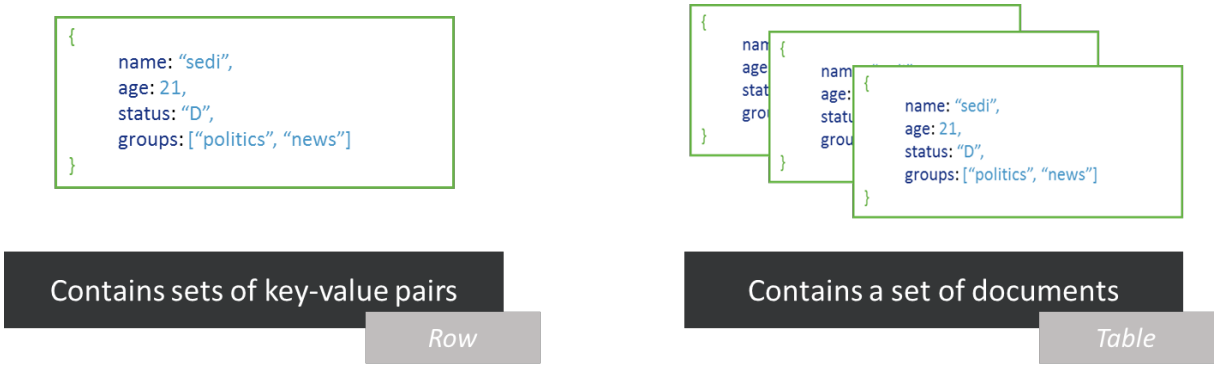


Abbildung 1.1: Documents and collections

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(?, ?). 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(?, ?). 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(?, ?).

## 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(?, ?).
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(?, ?).
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(?, ?).

## 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 (?, ?). 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 (?, ?).

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 (?, ?). The following commands show how simply it is to select and create a new database by inserting its first document.

```
1 use newDatabase
2 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 **??**, a collection with limited number of documents is created. All options can be found in the MongoDB documentation (?, ?).

```
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 (?, ?). 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 **??** 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 (?, ?).

```
var newDocument = {
  _id: <ObjectId>,
  birthdate: 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 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 (?, ?). 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 (?, ?). 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 **??**.

```
var newPerson = {
  _id: <ObjectId1>,
  birthdate: 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 (?, ?).

### 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 birthdate, would look like the following in Listing **??**. Furthermore, it is also possible the check the fields for a certain data type (?, ?).

```
db.createCollection( 'users', {
  validator: {
      $or: [ {
          birthdate: { $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 **??**. But there are also commands to insert an array of documents at once (?, ?).

```
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 **??** 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 (?, ?).

```
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 (?, ?). An example for a basic update operation can be found in Listing **??**.

```
1 db.newCollection.update(
2     { name: 'John' },
3     { age: 21, name: 'John', lastname: 'Doe'  },
4     { 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 **??**, 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 (?, ?). Line 2 shows how to delete a collection and line 3 shows how to delete an entire database.

```
1 db.newCollection.remove( { age: 20 } )
2 db.newCollection.drop()
3 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. 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, Hows, Membrey & Hawkins, 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). For that MongoDB provides two different replication methods. The traditional *Master/Slave Replication* and *Replication Sets*.

### Master/Slave Replication

In a Master/Slave set up 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 and 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).

### Consitency

The replication in MongoDB works asynchronously. Writes are always routed to the primary, but reads can also requested to a secondary. „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). It is possible to configure the preferred reading node by primary, primaryPrefferd, secondary, secondaryPreffered or nearest (Plugge et al., 2015). The only way to ensure that a client always requests the up to date data is to set the read preference to primary. But in this situation we only get the traditional *Master/Slave* behaviour and we loose all availability features. Normally the read preference would be set to primaryPreffered, so reads were just send to the secondaries if the primary is unavailable. Or if the system is widely distributed nearest would be chosen. To ensure a minimum number of secondaries is up to date it is possible to specify write concerns. With write concerns the client will get the success response only after the write operation was replicated to the specified number of secondaries (Sabharwal & Edward, 2015).

### 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). For that 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 a *mongod* process. All the shards are combined to one logical database. The partitioning and routing are managed by the earlier mentioned *mongos* 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 request to responsible *mongod* process. Also the partitioning decision is taken by the *mongos* process. 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.

### 1.5.4 Summary

Figure 1.1 describes one possible deployment architecture that contains all in this chapter 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.

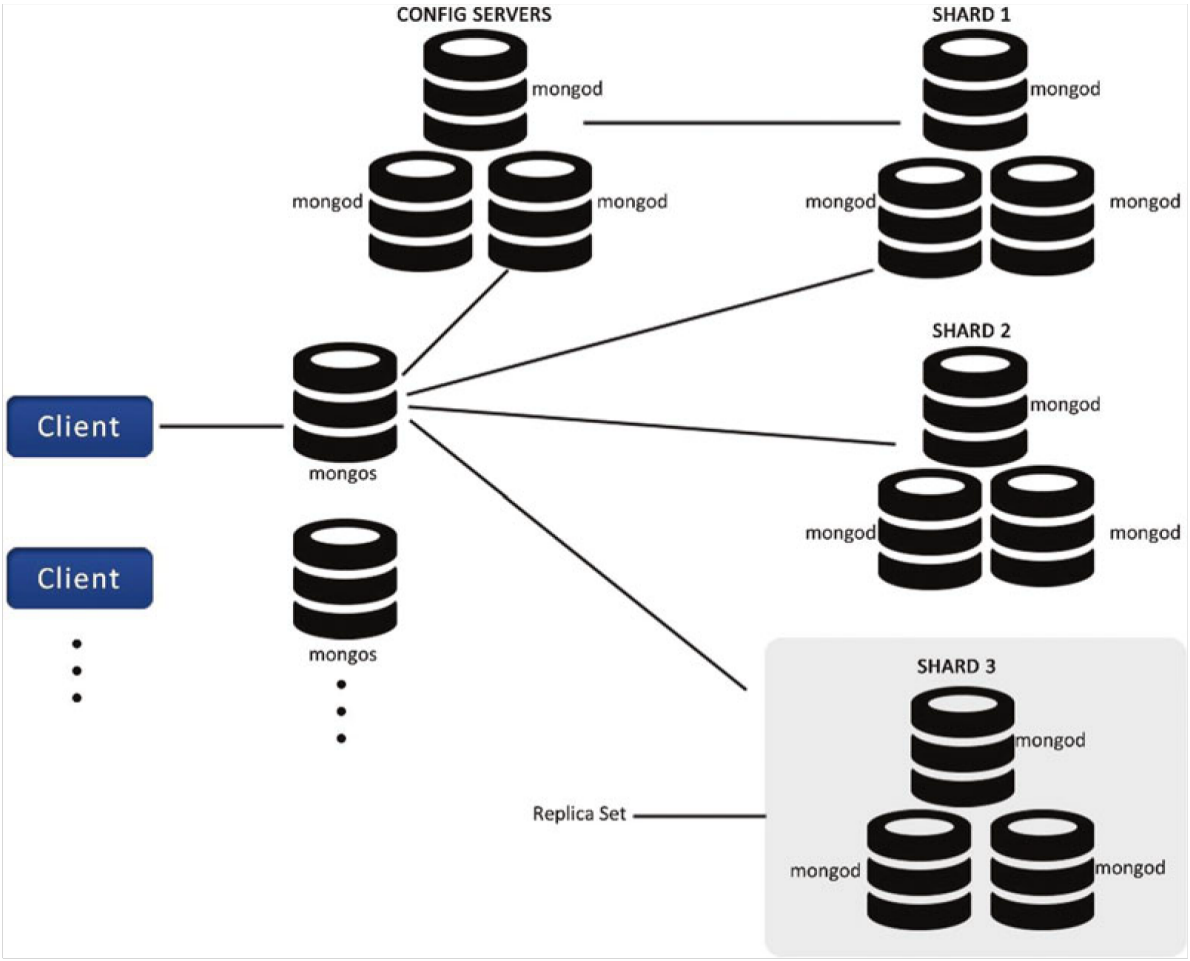


Abbildung 1.2: MongoDB Architecture Example

## 1.6 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(?, ?).

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(?, ?).

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(?, ?).
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(?, ?).
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:

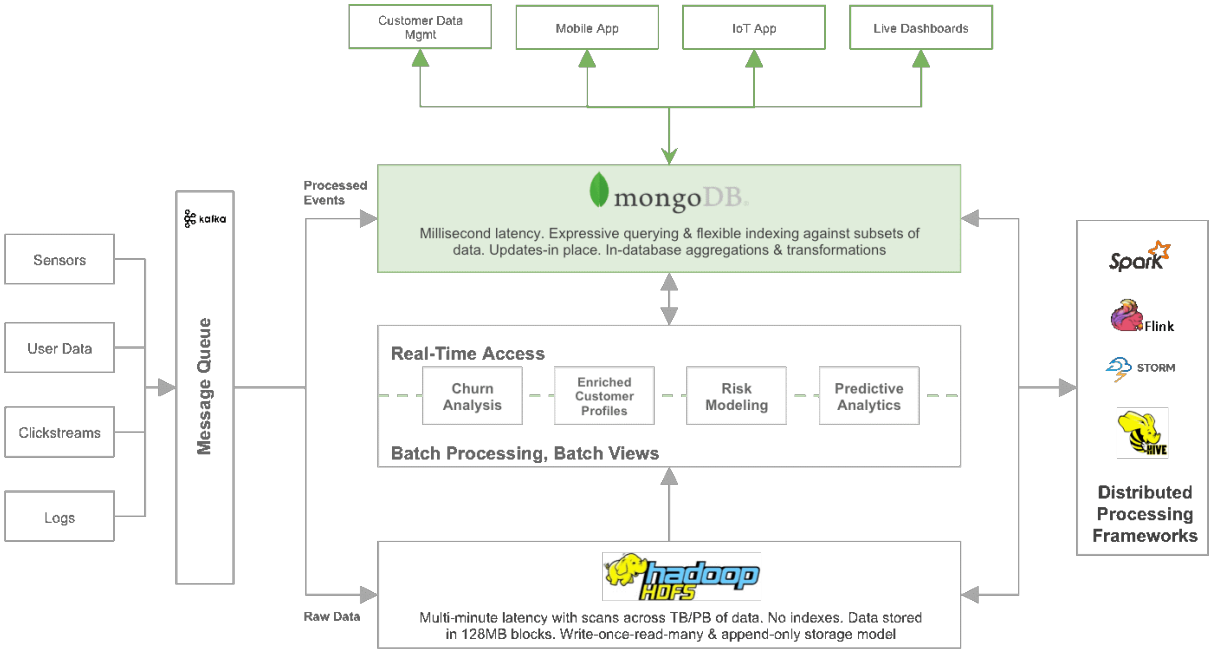


Abbildung 1.3: Design pattern for integrating MongoDB with Data Lake

# Literaturverzeichnis

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 Aufl.). Berkely, CA, USA: Apress.

Sabharwal, N. & Edward, S. G. (2015). *Practical mongodb: Architecting, developing, and administering mongodb.* Berkely, CA, USA: Apress.