



# Document Stores

MongoDB

Fabrício Ferreira, Ledia Isaj

December 2019

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>NoSQL Databases</b>	<b>2</b>
2.1	NoSQL databases compared to RDBMS . . . . .	3
2.2	Types of NoSQL databses . . . . .	5
<b>3</b>	<b>Document Stores</b>	<b>6</b>
3.1	Popular Document Stores . . . . .	7
3.2	MongoDB . . . . .	7
<b>4</b>	<b>NoSQL Performance Benchmark</b>	<b>8</b>
4.1	Describing the dataset . . . . .	9
4.2	Experiment and results . . . . .	9
4.2.1	Single Read . . . . .	10
4.2.2	Single Write . . . . .	11
4.2.3	Single Write Synchronized . . . . .	11
4.2.4	Aggregation . . . . .	11
4.2.5	Neighbors . . . . .	11
4.2.6	Results . . . . .	12
<b>5</b>	<b>TPC-H</b>	<b>12</b>
5.1	Creating relational database in MySQL . . . . .	13
5.2	Adjusting the relational database for MongoDB . . . . .	13
5.3	Performance comparison . . . . .	19
5.4	Queries . . . . .	19
<b>6</b>	<b>Results</b>	<b>22</b>
<b>7</b>	<b>Conclusion</b>	<b>23</b>

# 1 Introduction

For many years, relational databases have been a standard to store data. The data is structured into tables, columns, and rows. The columns have relationships defined among the tables. [18]

Nowadays, the volume of data has increased significantly with the emergence of social networks, cloud, web and smartphone applications, etc. Moreover, the environment is changing fast and frequent and it is up to the business to keep up with its pace. Due to relational databases' strict structure and rules, this is not a good option for many companies. NoSQL is an alternative, schemaless and scalable solution.

For that reason, we were motivated to explore NoSQL databases and Document Stores in particular, in order to have a deeper look at what they have to offer. Moreover, we decided to compare in terms of performance one of the most popular document-oriented engine, MongoDB to other popular relational solutions, such as PostgreSQL and MySQL.

Firstly, we are going to quickly describe NoSQL databases and provide a quick overview of the differences with relational databases. Secondly, we are going to focus on Document Stores and its features. Thirdly, we will introduce MongoDB as one of the most popular engines in document-oriented solutions. Then, we will run a NoSQL benchmark introduced by ArangoDB to compare MongoDB with PostgreSQL. TPC-H benchmark was also adapted to compare the performance of MongoDB to MySQL, in a decision support application.

# 2 NoSQL Databases

NoSQL is a term that refers to a class of databases that do not follow the principles of relational management systems.

Classical database systems guarantee the integrity of the data relying on ACID properties (Atomicity, Consistency, Isolation, and Durability). However, scaling out of these systems has shown to be challenging. Conflicts are arising between different aspects of high availability in distributed systems, known as the CAP theorem.

- Strong Consistency: all the clients see the same version of the data
- High Availability: all clients can always find at least one copy of the requested data, even if some of the machines in a cluster are down.
- Partition-tolerance: the total system keeps its characteristic even when being deployed on different servers, transparent to the client.

The CAP-Theorem states that only two of the three different aspects of scaling out can be achieved fully at the same time. Many of the NoSQL databases have loosened up the requirements on Consistency to achieve better Availability and Partitioning. This resulted in systems known as BASE (Basically Available,

Soft-state, Eventually consistent). However, different approaches have been taken by NoSQL databases regarding the CAP theorem.

Primary Uses of NoSQL Database:

1. Large-scale data processing (parallel processing over distributed systems)
2. Embedded IR (basic machine-to-machine information look-up and retrieval)
3. Exploratory analytics on semi-structured data (expert level)
4. Large volume data storage (unstructured, semi-structured, small-packet structured) [10]

## 2.1 NoSQL databases compared to RDBMS

The main question is: What are the key features of NoSQL Databases compared to relational ones?

1. NoSQL databases feature dynamic schema, and allow you to use what's known as "unstructured data." This means you can build your application without having to first define the schema. No predefined schema makes NoSQL databases much easier to update as data and requirements change. Changing the schema structure in a relational database can be extremely expensive, time-consuming, and often involve downtime or service interruptions.
2. Relational databases are table-based. They were built during a time that the data was structured. Nowadays data is more complex. NoSQL databases can handle unstructured or semi-structured data, which can be document based, graph databases, key-value pairs, or wide-column stores.
3. Relational databases are vertically scalable but typically expensive. Since they require a single server to host the entire database, in order to scale, you need to buy a bigger, more expensive server. Scaling a NoSQL database is much cheaper, compared to a relational database, because you can add capacity by scaling horizontally over cheap, commodity servers.
4. NoSQL databases tend to be more a part of the open-source community. [15]

In the table below, you can find the main differences between NoSQL and SQL databases. [12]

NoSQL vs Relational		
	SQL Databases	NoSQL
Types	One type (SQL database) with minor variations	Many different types including key-value stores, document databases, wide-column stores, and graph databases
Data Storage Model	Individual records are stored as rows in tables, with each column storing a specific piece of data about that record, much like a spreadsheet. Related data is stored in separate tables and then joined together when more complex queries are executed.	Varies based on database type. For example, key-value stores function similarly to SQL databases, but have only two columns ('key' and 'value'), with more complex information sometimes stored as BLOBs within the 'value' columns. Document databases store all relevant data together in single 'document' in JSON, XML, or another format, which can nest values hierarchically.
Schemas	Fixed schema. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Dynamic, with some enforcing data validation rules. New fields can be added on the fly. Dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.

	SQL Databases	NoSQL
Scaling	Vertically, meaning a single server must be made increasingly powerful to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required, and core relational features such as JOINS, referential integrity and transactions are typically lost.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary.
Supports multi-record ACID transactions	Yes	Mostly no. MongoDB 4.0 and beyond support multi-document ACID transactions.
Data Manipulation	standard query language	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB, with tunable consistency for reads) whereas others offer eventual consistency (e.g., Cassandra).

## 2.2 Types of NoSQL databases

There are four widely used NoSQL databases:

- Key-value pair databases
- Document-oriented databases
- Column family databases
- Graph databases

Key-value pair databases store information as matched pairs of key (hashed) and value. Values can be simple text or complex data types, such as sets of data. Data searches can usually only be performed against keys, not values, and are limited to exact matches. These kinds of databases are well suited to a lightning-fast, highly scalable retrieval of the values needed for application tasks like managing user profiles or sessions or retrieving product names.

Document-oriented databases store information in the form of a document. Data is denormalized, semi-structured and stored hierarchically. These documents are encoded in a standard data exchange format such as XML, JSON (Javascript Option Notation) or BSON (Binary JSON). Both keys and values are fully searchable in document databases. Document stores are good for storing and managing big collections of literal documents, for instance, text documents, email messages, and XML documents, as well as conceptual documents like de-normalized representations of a database entity. They are also good for storing sparse data in general, that is to say, irregular (semi-structured) data that would require extensive use of nulls in an RDBMS.

Column Family databases: The database structure of this NoSQL type is similar to the standard Relational Database Management System (RDMS) since all the data is stored as sets of columns and rows. However, RDBMS tend to have simple data types and a predefined schema. Column-oriented databases provide much more flexibility and support complex data types, unstructured text and graphics. This type of DMS is great for distributed data storage, especially versioned data because of WC/CF time-stamping functions; large-scale, batch-oriented data processing: sorting, parsing, conversion, algorithmic crunching; exploratory and predictive analytics performed by expert statisticians and programmers.

Graph databases are mostly used when the stored data may be represented as a graph with interlinked elements such as social networking, road maps or transport routes. Graph databases are useful when you are more interested in relationships between data than in the data itself: for example, in representing and traversing social networks, generating recommendations or conducting forensic investigations (e.g., pattern detection). [8] [10] [3]

### 3 Document Stores

A document-oriented database, or document store, is a computer program designed for storing, retrieving and managing document-oriented information, also known as semi-structured data. Document-oriented databases are one of the main categories of NoSQL databases.

As mentioned earlier, document stores inherit all the features of NoSQL databases, like the ability to load unstructured complex data, no schema, scalability, etc.

Document stores are a type of key-value store: each document has a unique identifier (it's key) and the document itself serves as the value. The difference between these two models is that, in a key-value database, the data is treated as opaque and the database doesn't know or care about the data held within it. It's up to the application to understand what data is stored. In a document store, however, each document contains some kind of metadata that provides a degree of structure to the data. Conceptually the document-store is designed to offer a richer experience with modern programming techniques.

Relational databases generally store data in separate tables that are defined by the programmer, and a single object may be spread across several tables. Document databases store all information for a given object in a single instance in the database, and every stored object can be different from every other. This eliminates the need for object-relational mapping while loading data into the database. Object-relational mapping (ORM) is a programming technique in which a metadata descriptor is used to connect object code to a relational database. ORM converts data between type systems that are unable to coexist within relational databases and OOP languages. [2] [11] [16]

### 3.1 Popular Document Stores

According to the DB-Engines ranking of Document Stores, MongoDB, Amazon DynamoDB, and Couchbase are the top 3 popular ones. [7]

MongoDB is one of the most popular document stores available both as a fully managed cloud service and for deployment on self-managed infrastructure. This platform can be used by developers building OLTP and analytical apps. MongoDB maintains the most valuable features of relational databases: strong consistency, ACID transactions, expressive query language, and secondary indexes. MongoDB provides the data model flexibility, elastic scalability, along with the performance and resilience of NoSQL databases. As a result, developers can continuously enhance applications and deliver them at an almost unlimited scale wherever they choose to run them. [9]

Amazon DynamoDB is a key-value and document database system. It is a hosted, scalable database service by Amazon with the data stored in Amazon's cloud. It's a fully managed, multi-region, multi-master, durable database with built-in security, backup and restores, and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and can support peaks of more than 20 million requests per second. [4]

Couchbase is a JSON-based document store derived from CouchDB with a Memcached-compatible interface. Couchbase architected a SQL-compatible NoSQL database for today's data-intensive web, mobile, and IoT applications. Couchbase is built on open standards, combining the best of NoSQL with the power and familiarity of SQL, to simplify the transition from mainframe and relational databases. Couchbase has a shared-nothing, asynchronous, elastic architecture and has a consistent performance at any scale. It is globally distributed, edge-to-cloud. It has workload isolation with multi-dimensional scaling. [6]

### 3.2 MongoDB

MongoDB is a document-oriented database. It is a powerful, flexible, and scalable general-purpose database.

Due to the fact that allows embedded documents and arrays, it is possible to represent complex hierarchical relationships with a single record in MongoDB.



This fits naturally into the object-oriented way of thinking of developers about their data. Adding/removing fields is easier and faster. This gives the opportunity to try different data models and choose which is best for your scenario.

MongoDB was designed to scale out. MongoDB automatically takes care of balancing data and load across a cluster, redistributing documents automatically and routing user requests to the correct machines. This allows developers to focus on programming the application, not scaling it.

MongoDB offers a wide range of features aside from its general-purpose ones, like creating, reading, updating, and deleting data. MongoDB supports generic secondary indexes, allowing a variety of fast queries, and provides unique, compound, geospatial, and full-text indexing capabilities as well. Moreover, MongoDB supports an “aggregation pipeline” that allows you to build complex aggregations from simple pieces and allow the database to optimize it. There are also special collection types, for instance, time-to-live collections for data that should expire at a certain time (such as sessions), fixed-size collections, which are useful for holding recent data (such as logs). MongoDB supports an easy-to-use protocol for storing large files and file metadata.

However, there are some features of relational databases that are not present in MongoDB, such as joins and complex multirow transactions. These features were sacrificed to gain greater scalability. [5]

## 4 NoSQL Performance Benchmark

This benchmark is based on the benchmark provided by ArangoDB to compare different NoSQL solutions performance. We are going to run this benchmark in MongoDB for document stores and PostgreSQL for a relational database. [14] The same data and the same hardware is used to test each database system. The tests for this benchmark:

- single-read: these are single document reads of profiles (100,000 different documents).
- single-write: these are single document writes of profiles (100,000 different documents).
- single-write sync: these are the same as single-writes, but we waited for fsync on every request.
- aggregation: these are ad-hoc aggregation over a single collection (1,632,803 documents).
- neighbors second: we searched for distinct, direct neighbors, plus the neighbors of the neighbors, returning ID's for 1,000 vertices. [13]

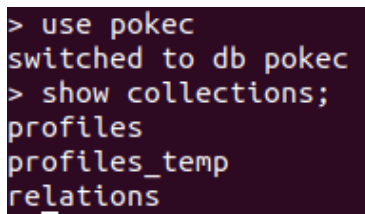
## 4.1 Describing the dataset

Pokec is the most popular on-line social network in Slovakia. Pokec has been provided for more than 10 years and connects more than 1.6 million people. Datasets contain anonymized data of the whole network. A snapshot of its data provided by the Stanford University SNAP was used for this benchmark. It contains profile data from 1,632,803 people. The corresponding friendship graph has 30,622,564 edges. The profile data contain gender, age, hobbies, interest, education, etc. However, the individual JSON documents are very diverse because many fields are empty for many people. Profile data are in the Slovak language. Friendships in Pokec are directed. The uncompressed JSON data for the vertices need around 600 MB and the uncompressed JSON data for the edges require around 1.832 GB. [17] [13]

## 4.2 Experiment and results

The experiment was conducted in Google Cloud Platform in Ubuntu 18.04 LTS virtual machine, 2 vCPUs, 7.5 GB memory, and 100 GB disk size. The goal of this experiment is to compare how document stores in MongoDB performs comparing to a regular relational database in PostgreSQL. First, MongoDB (3.6.1) and PostgreSQL(10.1.1) were downloaded. Then the data was imported in MongoDB and PostgreSQL. Below, we are going to explore the data loaded in MongoDB. The same information is loaded in PostgreSQL in a tabular form.

Pokec database in MongoDB consists of 3 collections: profiles, profiles\_temp, and relations, as presented in figure 1. To show the list of collections, *show collections* command is used.

A screenshot of a terminal window with a dark background and light-colored text. The text shows a MongoDB command prompt session. The first line is a prompt character followed by 'use pokec'. The second line is the response 'switched to db pokec'. The third line is a prompt character followed by 'show collections;'. The fourth line is the response listing three collections: 'profiles', 'profiles\_temp', and 'relations' on separate lines.

```
> use pokec
switched to db pokec
> show collections;
profiles
profiles_temp
relations
```

Figure 1: MongoDB collections

In figure 2, we can see what kind of documents, profiles collection holds. The command: *db.collection.find(query, projection)* selects documents in a collection or view and returns a cursor to the selected documents. Query and projection are optional parameters to specify selection filtering or specify the fields to return.

As we can see, profiles collection holds information regarding the user profiles, such as gender, region, timestamp of the last login, timestamp of registration, age, body properties like height and weight, spoken languages, hobbies, favorite color, etc.

```

> db.profiles.find()
{ "_id" : "P2", "public" : 1, "completion.percentage" : 62, "gender" : 0, "region" : "zilinsky kraj, lyseucke nove me
sto", "last_login" : "2012-05-25 23:08:00.0", "registration" : "2007-11-30 00:00:00.0", "AGE" : 0, "body" : "166 cm,
58 kg, "i am working in field" : "", "spoken languages" : "nemecky", "hobbies" : "turistika, prace okolo domu, pra
ca s pc, poznavani hudby, poznavani filmu, tanovani, diskoteky, kupalisko, varenie, party, priatelis, spanie, maš
spovanie, stanovanie", "i most enjoy good food" : "pri sviečkach s partnerom", "pets" : "mačka", "body type" : "prie
sna", "my eyesight" : "vyborny", "eye color" : "zelene", "hair color" : "cierna", "hair type" : "dlha", "completed
level of education" : "zakladne, ale som uz na strednej škole dufam ze ju spreviam", "favourite color" : "cierna, no
dra, ružova", "relation to smoking" : "nefajcia", "relation to alcohol" : "pijem prilezitostne, iba ked sa nieco kom
u a to napr. na zabavy, na chate, na stanovackach a pod.", "sign in zodiac" : "byk", "do poker i am looking for" : "do
brego priateľa, priateľku, možno aj viac", "flow is for me" : "nie je nič lepšie, ako byť zamilovaný(a)", "relations
to casual sex" : "iba s mojou láskou", "my partner should be" : "láskou mojej života", "marital status" : "alobodný(
á)", "children" : "ne budu a tak ked budeme vladat tak bude aj viac co ja viem co ma v zivote
ne potrebujem", "relation to children" : "v buducnosti chcem mat deti", "i like movies" : "komedio, romanticke", "i
like watching movie" : "doma z gauča", "i like music" : "disko, pop, rap a jam eto co teraz leti najviac najlepšie
je fun-radio", "i mostly like listening to music" : "na diskoteky, pri chodži", "the idea of good evening" : "pri s
viečkach s partnerom", "i like specialties from kitchen" : "slovenskej", "fun" : "<div> <a title='vstup do klubu'>
href='\\klub\\profesionalni'>profesionalni\\</a> </div>", "i am going to concerts" : "", "my active sports" : "", "
my passive sports" : "", "profession" : "", "i like books" : "", "life style" : "", "music" : "", "cars" : "", "politi
tics" : "", "relationships" : "", "art culture" : "", "hobbies interests" : "", "science technologies" : "", "comput
ers internet" : "", "education" : "", "sport" : "", "movies" : "", "travelling" : "", "health" : "", "companies bran
ch" : "", "mars" : "" }

```

Figure 2: Profiles

Collection relations contains the relationships between profiles.

```

> db.relations.find()
{ "_id" : ObjectId("5df45930605a41558cd572fd"), "from" : "P1", "to" : "P13" }
{ "_id" : ObjectId("5df45930605a41558cd572fe"), "from" : "P1", "to" : "P11" }
{ "_id" : ObjectId("5df45930605a41558cd572ff"), "from" : "P1", "to" : "P3" }
{ "_id" : ObjectId("5df45930605a41558cd57300"), "from" : "P1", "to" : "P4" }
{ "_id" : ObjectId("5df45930605a41558cd57301"), "from" : "P1", "to" : "P5" }
{ "_id" : ObjectId("5df45930605a41558cd57302"), "from" : "P1", "to" : "P15" }
{ "_id" : ObjectId("5df45930605a41558cd57303"), "from" : "P1", "to" : "P14" }
{ "_id" : ObjectId("5df45930605a41558cd57304"), "from" : "P1", "to" : "P7" }
{ "_id" : ObjectId("5df45930605a41558cd57305"), "from" : "P1", "to" : "P8" }
{ "_id" : ObjectId("5df45930605a41558cd57306"), "from" : "P1", "to" : "P12" }
{ "_id" : ObjectId("5df45930605a41558cd57307"), "from" : "P1", "to" : "P9" }
{ "_id" : ObjectId("5df45930605a41558cd57308"), "from" : "P1", "to" : "P10" }
{ "_id" : ObjectId("5df45930605a41558cd57309"), "from" : "P1", "to" : "P16" }
{ "_id" : ObjectId("5df45930605a41558cd5730a"), "from" : "P2", "to" : "P1" }
{ "_id" : ObjectId("5df45930605a41558cd5730b"), "from" : "P2", "to" : "P18" }
{ "_id" : ObjectId("5df45930605a41558cd5730c"), "from" : "P2", "to" : "P19" }
{ "_id" : ObjectId("5df45930605a41558cd5730d"), "from" : "P2", "to" : "P20" }
{ "_id" : ObjectId("5df45930605a41558cd5730e"), "from" : "P2", "to" : "P21" }
{ "_id" : ObjectId("5df45930605a41558cd5730f"), "from" : "P2", "to" : "P22" }
{ "_id" : ObjectId("5df45930605a41558cd57310"), "from" : "P2", "to" : "P23" }

```

Figure 3: Relations

In figure 4, we can see that profiles\_temp collection holds temporary information about profiles. As we can see, collections can have different fields, which makes MongoDB dynamic. Moreover, if the information is missing for a field, that field can be omitted from the document, which is a good solution for sparse tables with a lot of null values in relational databases.

```

{ "_id" : ObjectId("5df45930605a41558cd57311"), "public" : 1, "completion.percentage" : 62, "gender" : 0, "region" : "zilinsky kraj, lyseucke nove me
sto", "last_login" : "2012-05-25 23:08:00.0", "registration" : "2007-11-30 00:00:00.0", "spoken languages" : "nemecky", "hobbies" : "turistika, prace okolo domu, pra
ca s pc, poznavani hudby, poznavani filmu, tanovani, diskoteky, kupalisko, varenie, party, priatelis, spanie, maš
spovanie, stanovanie", "i most enjoy good food" : "pri sviečkach s partnerom", "pets" : "mačka", "body type" : "prie
sna", "my eyesight" : "vyborny", "eye color" : "zelene", "hair color" : "cierna", "hair type" : "dlha", "completed
level of education" : "zakladne, ale som uz na strednej škole dufam ze ju spreviam", "favourite color" : "cierna, no
dra, ružova", "relation to smoking" : "nefajcia", "relation to alcohol" : "pijem prilezitostne, iba ked sa nieco kom
u a to napr. na zabavy, na chate, na stanovackach a pod.", "sign in zodiac" : "byk", "do poker i am looking for" : "do
brego priateľa, priateľku, možno aj viac", "flow is for me" : "nie je nič lepšie, ako byť zamilovaný(a)", "relations
to casual sex" : "iba s mojou láskou", "my partner should be" : "láskou mojej života", "marital status" : "alobodný(
á)", "children" : "ne budu a tak ked budeme vladat tak bude aj viac co ja viem co ma v zivote
ne potrebujem", "relation to children" : "v buducnosti chcem mat deti", "i like movies" : "komedio, romanticke", "i
like watching movie" : "doma z gauča", "i like music" : "disko, pop, rap a jam eto co teraz leti najviac najlepšie
je fun-radio", "i mostly like listening to music" : "na diskoteky, pri chodži", "the idea of good evening" : "pri s
viečkach s partnerom", "i like specialties from kitchen" : "slovenskej", "fun" : "<div> <a title='vstup do klubu'>
href='\\klub\\profesionalni'>profesionalni\\</a> </div>", "i am going to concerts" : "", "my active sports" : "", "
my passive sports" : "", "profession" : "", "i like books" : "", "life style" : "", "music" : "", "cars" : "", "politi
tics" : "", "relationships" : "", "art culture" : "", "hobbies interests" : "", "science technologies" : "", "comput
ers internet" : "", "education" : "", "sport" : "", "movies" : "", "travelling" : "", "health" : "", "companies bran
ch" : "", "mars" : "" }

```

Figure 4: Profiles\_temp

After loading the data, queries described in the following section were run to measure the performance of both technologies.

#### 4.2.1 Single Read

The goal is to read single documents of profile collection For this test, multiple single reads(100,000) were done on MongoDB using the following command: *db.collection.findOne(query, projection)* This command returns one document

that satisfies the specified query criteria on the collection or view. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. In capped collections, the natural order is the same as the insertion order. If no document satisfies the query, the method returns null. The selection criteria were the `_id` of the document. In PostgreSQL, we simply read from table profiles, where `_key` is equal to a given id.

#### 4.2.2 Single Write

The goal is to write single documents in `profile_temp` collection. For this test multiple single writes (100,000) were executed using the following command: `db.collection.insert()`, which inserts a document or documents into a collection. In PostgreSQL, insert statement is used.

#### 4.2.3 Single Write Synchronized

This test is similar to the previous one, except for the fact that we wait for `fsync` on every request. Function `fsync()` transfers all modified in-core data of the file referred to by the file descriptor `fd` to the disk device so that all changed information can be retrieved even after the system crashed or was rebooted. This includes writing through or flushing a disk cache if present. The call blocks until the device report that the transfer has completed. It also flushes metadata information associated with the file. To flush the journal to disk can be simply implemented in MongoDB by setting `j=true`, as a write concern parameter in the function described previously. In PostgreSQL this is not implemented

#### 4.2.4 Aggregation

In this test, we did an ad-hoc aggregation over 1,632,803 profile documents and counted how often each value of the AGE attribute occurred. We didn't use a secondary index for this attribute on any of the databases. As a result, they all had to perform a full collection scan and do counting statistics. Command `db.collection.aggregate(pipeline, options)` calculates aggregate values for the data in a collection or a view. Pipeline is a sequence of data aggregation operations or stages. In our case we group by age and count the profiles for each age group. `coll.aggregate([{$group: _id: '$AGE', count: $sum: 1}])`. In PostgreSQL, the following command is used: `select AGE, count(*) from profiles group by AGE`.

#### 4.2.5 Neighbors

During this test, we searched for distinct, direct neighbors, plus the neighbors of the neighbors, returning ID's for 1,000 vertices. This is a typical graph matching problem, considering paths of length one or two. For the non-graph database MongoDB, we used the aggregation framework to compute the result.

In PostgreSQL, we used a relational table with id from and id to, each backed by an index. In the Pokec dataset, we found 18,972 neighbors and 852,824 neighbors of neighbors.

#### 4.2.6 Results

The tests were run five times and the average of the results was calculated. In figure 5, the average time to read, write and find neighbors for 1 document/entity is presented for MongoDB and PostgreSQL. Similarly, the total time to read and write 100000 documents/entities, to do aggregation queries over all the instances in profiles and finding the neighbors for all instances is shown in figure 6. It is noticeable that PostgreSQL outperforms MongoDB. Particularly, time to read a single instance in PostgreSQL is 0.74 of the time to read it in MongoDB, 0.6 for writing a single instance, and much faster in more complex queries like finding the neighbors and aggregation.

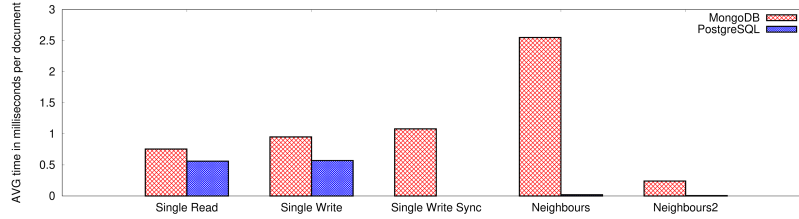


Figure 5: Average time

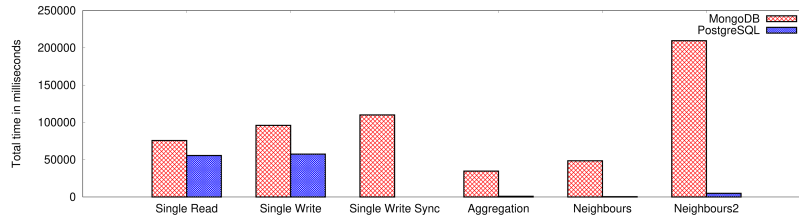


Figure 6: Total time

## 5 TPC-H

The TPC-H is a decision support benchmark. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. This benchmark illustrates decision support systems that:

- Examine large volumes of data;

- Execute queries with a high degree of complexity;
- Give answers to critical business questions.

The goal of the TPC-H queries is to give an answer to real-world business questions, simulate generated ad-hoc queries, show a higher level of complexity compared to most OLTP transactions, and include a wide variety of operators and selectivity constraints. Moreover, they are executed against a database complying to a specific population and scaling requirements and implemented with constraints derived from staying closely synchronized with an on-line production database.

The metric considered is the performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric.[1]

This experiment was run in a Microsoft Windows 10 Enterprise machine, x64 bit OS, i7 CPU, 16.0 GR. The latest versions of MongoDB(4.2) and MySQL(8.0.18) were downloaded.

## 5.1 Creating relational database in MySQL

To generate the data, the dbgen tool offered by TPC-H benchmark was used. The dataset generated in this particular case was of 10 MB. After the dataset was generated, respective tables were created and the data were loaded from cvs files.

As we can see in the logical model provided in Figure 7, the database consists of 8 tables: part, supplier, partsupp, customer, nation, lineitem, region, and order. The number of rows for each table is 2000, 100, 8000, 1500, 25,60175, 4, 15000 respectively.

## 5.2 Adjusting the relational database for MongoDB

In order to load the dataset generated by dbgen tool into MongoDB, a few adjustments should be made.

Firstly, the csv files should be converted to json. We used a python code that iterated all the files line by line and converts to json.

```
def convert_csv_to_json(csv_line , csv_headings):
    json_elements = []
    for index,heading in enumerate(csv_headings):
        json_elements.append(heading + ": \"\" +

        unicode(csv_line[index], 'UTF-8') + "\"")

    line = "{ " + ', '.join(json_elements) + " }"
    return line
```

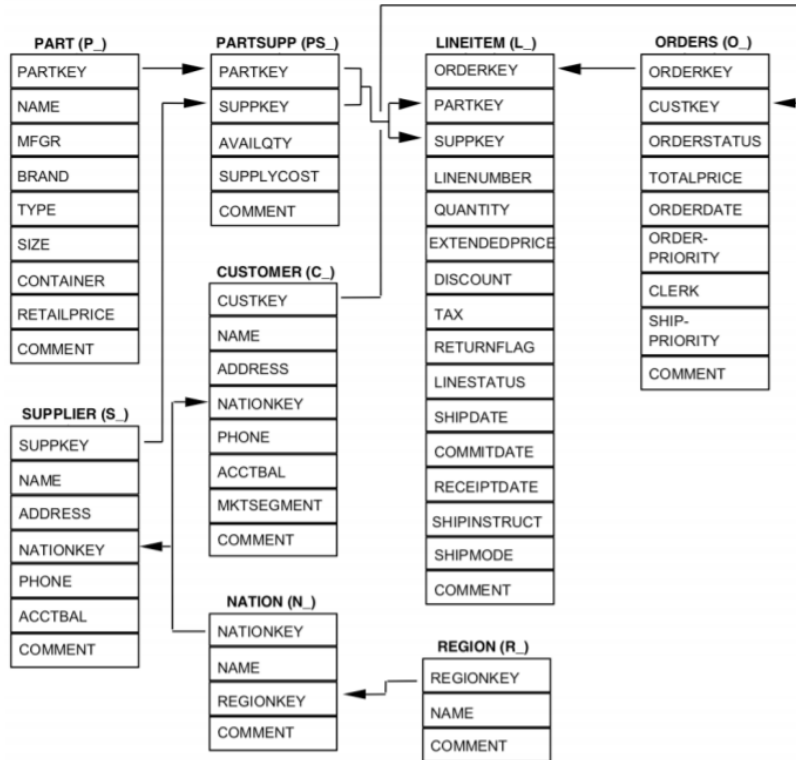


Figure 7: Logical Model

After that, we created the tpch database in MongoDB and loaded the collections into tpch database. The command to load the collections is described below for loading the nation collection. Similarly, all the collections were loaded into MongoDB.

```
C:\Program Files\MongoDB\Server\4.2\bin>mongoimport --db tpch --collection nation --file C:\Users\Ledia\Desktop\projectADB\nation.json
2019-12-11T00:48:25.960+0100 connected to: mongod://localhost/
2019-12-11T00:48:26.033+0100 25 document(s) imported successfully. 0 document(s) failed to import.
```

Figure 8: Loading nation collection

When all the collections were loaded, we did some data parsing. All the data that we loaded were string, but in order to go manipulate the data and do aggregations, we needed to convert some of the fields from string to float numbers. In the code below it is presented how we converted custkey to float in the customer collection.

```

db.customer.find( { 'custkey' : { "$type" : 2 } } ).forEach(
function (x) {
    x.custkey = parseFloat(x.custkey);
    db.customer.save(x);
});

```

Similarly, we parsed for the following fields:

- collection customer: nationkey and acctbal
- collection lineitem: orderkey, suppkey, partkey, linenumber, quantity, extendedprice, discount and tax
- collection order: orderkey, custkey and totalprice
- collection supplier: suppkey, nationkey, and acctbal
- collection partsupp: suppkey, partkey, supplycost, availqty
- collection part: partkey, size, retailprice
- collection region: regionkey
- collection nation: regionkey, nationkey

As we seek for performance, we denormalized the data by embedding. Below we present the code used to denormalize region into nation.

```

var i=0;
var temp;
var temp2;

db.nation.update({},{$set : {"region":1}},false,true)
db.supplier.update({},{$set : {"nation":1}},false,true)
db.customer.update({},{$set : {"nation":1}},false,true);

while (i<5){
    temp2 = db.region.findOne({"regionkey": i });
    db.nation.update({"regionkey": i },
    {$set:{"region": temp2}},false,true);
    printjson(i);
    i++;
}

```



```

    _id: ObjectId("5ded4454bbdf1f72b9614fbe")
    nationkey: 3
    name: "CANADA"
    regionkey: 1
    comment: "eas hang ironic, silent packages. slyly regular packages are furiously..."
  ✓ region: Object
    _id: ObjectId("5debee7cfb4c40d0c89a88e9")
    regionkey: 1
    name: "AMERICA"
    comment: "hs use ironic, even requests. s"

```

Figure 9: Denormalizing region into nation

Similarly, we continue denormalizing other collections.  
Nation is embedded into supplier.

```

    _id: ObjectId("5ded44c48c689cb575f1ae84")
    supplekey: 3
    name: "Supplier#000000003"
    address: "q1,G3Pj60jIuUYfUoH18BFTKP5aU9bEV3"
    nationkey: 1
    phone: "11-383-516-1199"
    acctbal: 4192.4
    comment: "blithely silent requests after the express dependencies are sl"
  ✓ nation: Object
    _id: ObjectId("5ded4454bbdf1f72b9614fc0")
    nationkey: 1
    name: "ARGENTINA"
    regionkey: 1
    comment: "al foxes promise slyly according to the regular accounts. bold request..."
  ✓ region: Object
    _id: ObjectId("5debee7cfb4c40d0c89a88e9")
    regionkey: 1
    name: "AMERICA"
    comment: "hs use ironic, even requests. s"

```

Figure 10: Denormalizing nation into supplier

Nation is embedded into customer.

```

_id: ObjectId("5ded4408de1dd928b4cb255a")
custkey: 1
name: "Customer#000000001"
address: "IVhziApeRb ot,c,E"
nationkey: 15
phone: "25-989-741-2988"
acctbal: 711.56
mktsegment: "BUILDING"
comment: "to the even, regular platelets. regular, ironic epitaphs nag e"
nation: Object
  _id: ObjectId("5ded4454bbdf1f72b9614fca")
  nationkey: 15
  name: "MOROCCO"
  regionkey: 0
  comment: "rns. blithely bold courts among the closely regular packages use furio..."
  region: Object

```

Figure 11: Denormalizing nation into customer

Part and supplier are embedded into partsupp.

```

_id: ObjectId("5ded44b2990d4d2e23ad8256")
partkey: 1
suppkey: 2
availqty: 3325
supplycost: 771.64
comment: ", even theodolites. regular, final theodolites eat after the carefully..."
part: Object
  _id: ObjectId("5ded449c0bf8a721ee3fd7be")
  partkey: 1
  name: "goldenrod lavender spring chocolate lace"
  mfg: "Manufacturer#1"
  brand: "Brand#13"
  type: "PROMO BURNISHED COPPER"
  size: 7
  container: "JUMBO PKG"
  retailprice: 901
  comment: "ly. slyly ironi"
  supplier: 1
supplier: Object
  _id: ObjectId("5ded44c48c689cb575f1ae85")
  suppkey: 2
  name: "Supplier#000000002"
  address: "89eJ5ksX3ImxJQBvxObC,"
  nationkey: 5
  phone: "15-679-861-2259"

```

Figure 12: Denormalizing part and supplier into partsupp

Customer is embedded into orders.

```

    _id: ObjectId("5ded4483b066245edb7df9ac")
    orderkey: 4
    custkey: 1369
    orderstatus: "0"
    totalprice: 56000.91
    orderdate: "1995-10-11"
    orderpriority: "5-LOW"
    clerk: "Clerk#000000124"
    shippriority: "0"
    comment: "sits. slyly regular warthogs cajole. regular, regular theodolites acro"
  > customer: Object
    _id: ObjectId("5ded4408de1dd928b4cb2ab1")
    custkey: 1369
    name: "Customer#000001369"
    address: "rXTwOzU0a2ak4Nj5L5b1aLij"
    nationkey: 10
    phone: "20-232-617-7418"
    acctbal: 498.77
    mktsegment: "AUTOMOBILE"
    comment: "ong the ironic ideas haggle slyly above the courts. packages engage bl..."
  > nation: Object

```

Figure 13: Denormalizing customer into orders

Orders and partsupp are embedded into lineitem.

```

    _id: ObjectId("5ded443a99137cb34dcfc953")
    orderkey: 1
    partkey: 637
    supkey: 38
    linenum: 3
    quantity: 8
    extendedprice: 12301.04
    discount: 0.1
    tax: 0.02
    returnflag: "N"
    linestatus: "0"
    shipdate: "1996-01-29"
    commitdate: "1996-03-05"
    receiptdate: "1996-01-31"
    shipinstruct: "TAKE BACK RETURN"
    shipmode: "REG AIR"
    comment: "riously. regular, express dep"
  > order: Object
  > partsupp: Object
    _id: ObjectId("5ded44b2990d4d2e23ad8c45")
    partkey: 637
    supkey: 38
    availqty: 3173
    supplycost: 224.93
    comment: "... hold requests make b13+433... and 433... 6223"

```

Figure 14: Denormalizing orders and partsupp into lineitem

The advantage of this schema is that one object represents one "deal". How-

ever, it has a drawback on higher memory usage.

### 5.3 Performance comparison

To compare the performance of the relational model built in MySQL and the equivalent document-oriented version built-in MongoDB, we ran a batch of queries selected from the auto-generated ones by TPC-H benchmark. The equivalent queries to run on MongoDB were written. In the subsection below we present some of these queries.

### 5.4 Queries

In this subsection, we show some of the SQL queries and their equivalent versions for MongoDB. Since 18 queries were used, not all of them are explained and presented here. For more information, you can find all the queries in Appendix A. [19]

**Pricing Summary Report Query (Q1)** The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The query lists totals for the extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included.

In figure 15 we can see the query generated by the tpch tool. In figure 16 we can see the equivalent query for MongoDB. To execute SQL query in MongoDB, we will use the pipeline aggregation framework. We will divide the above query into different stages and execute each of them in the pipeline. The stages that we will need here are a Match (where) stage, a Project(select) stage, a Group stage, and a Sort stage.

In the Match stage, we eliminate records that do not match the specification. All records where lshipdate is less than or equal 1996-01-10 will be kept and pass to the next stage of the pipeline. In the project stage, we keep only the required attributes needed for aggregation. Then group by aggregation is performed. The \_id attribute holds a distinct group by key which is used to do a group by on a data set. After all other aggregations are done we can proceed to sort the finished dataset in increasing order.

```

select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= '1996-01-10'
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus;

```

Figure 15: Query 1 in MySQL

```

db.lineitem.aggregate(
  [
    {
      "$match": { "shipdate": { "$lte": "1996-01-10" } }
    },
    {
      "$project": {
        "returnflag": 1, "linestatus": 1, "quantity": 1, "extendedprice": 1, "discount": 1,
        "l_dis_min_1": { "$subtract": [ 1, "$discount" ] }, "l_tax_plus_1": { "$add": [ 1, "$tax" ] }
      },
      {
        "$group": { "_id": { "RETURNFLAG": "$returnflag", "LINESTATUS": "$linestatus" },
          "sum_qty": { "$sum": "$quantity" },
          "sum_base_price": { "$sum": "$extendedprice" },
          "sum_disc_price": { "$sum": { "$multiply": [ "$extendedprice", "$l_dis_min_1" ] } },
          "sum_charge": { "$sum": { "$multiply": [ "$extendedprice", "$l_dis_min_1", "$l_tax_plus_1" ] } },
          "avg_qty": { "$avg": "$quantity" },
          "avg_price": { "$avg": "$extendedprice" },
          "avg_disc": { "$avg": "$discount" },
          "count_order": { "$sum": 1 }
        }
      },
    ],
    {
      "$sort": {
        "returnflag": 1,
        "linestatus": 1
      }
    }
  ]
).pretty();

```

Figure 16: Query 1 in MongoDB

**Order Priority Checking Query (Q4)** This query determines how well the order priority system is working and gives an assessment of customer satisfaction. The Order Priority Checking Query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order. In figure 17 we can see the query generated for MySQL.

```

select
  o_orderpriority,
  count(*) as order_count
from
  orders
where
  o_orderdate >= '1993-07-01'
  and o_orderdate < '1993-10-01'
  and exists (
    select
      *
    from
      lineitem
    where
      l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate --remove this and get the exact print o
  )
group by
  o_orderpriority
order by
  o_orderpriority;

```

Figure 17: Query 4 in MySQL

MongoDB can not run multiple queries at the same time due to the pipeline framework so to counter this flaw we can run the subqueries and store it's output in temp storage to later use in the main query. In this subquery, we essentially find the orders where their lineitem's commitdate is less than receiptdate and store all those orderkey in temporary storage. We can then use these orderkeys in the main query by placing them in the match stage which will find all document that matches the orderkey in the temporary array.

```

--
db.lineitem.aggregate([
  {
    "$project": {
      "orderkey": 1,
      "eq": {
        "$cond": [{
          "$lt": ["$commitdate", "$receiptdate"]
        },
        1,
        0
        ]
      }
    }
  }, {
    "$match": {
      "eq": {
        "$eq": 1
      }
    }
  }
]).forEach(function(u) {
  exist.push(u.orderkey)
})
db.orders.aggregate(

```

Figure 18: Query 4 in MongoDB

In figure 19, we can see the rest of the query. First, orderpriority, orderdate

and orderkey are projected. Then we match to select the documents that have o\_orderdate greater or equal to 1993-07-01 and have o\_orderdate less than 1993-10-01. We group the remaining dataset by o\_orderpriority which is in the \_id and compute an order count to see how many o\_orderpriority of each type occurs. In the end, we sort the results based on o\_orderpriority in ASC order.

```
db.orders.aggregate([
  {
    "$project": {
      "orderpriority": 1,
      "orderdate": 1,
      "orderkey": 1,
    }
  }, {
    "$match": {
      "orderdate": {
        "$gte": "1993-07-01",
        "$lt": "1993-10-01"
      },
      "orderkey": {
        "$in": exist
      }
    }
  }, {
    "$group": {
      "_id": {
        "orderpriority": "$orderpriority",
      },
      "order_count": {
        "$sum": 1
      }
    }
  }, {
    "$sort": {
      "orderpriority": 1
    }
  }
]).pretty()
```

Figure 19: Query 4 in MongoDB

## 6 Results

As we can see, for different types of queries, sometimes MongoDB outperforms MySQL, but usually, MySQL is faster. For example, query 1 is an aggregation

query on one table, filtering the data on where clause, grouping by specific columns and aggregating and ordering the results. We can see that MongoDB performs better than MySQL in this case. Time to execute query 1 in MongoDB 0.9 of the time needed for MySQL.

However, when the queries become more complex, like in query 4 that sub-query is needed, MongoDB is slower. Particularly, the time to run this query in MongoDB is 3.7 times higher compared to MySQL.

In query 3, MongoDB outperforms significantly MySQL. The reason may be that MySQL needs to join 3 tables, but in MongoDB, they are denormalized.

As we can see, generally, for this batch of queries MySQL outperforms MongoDB.

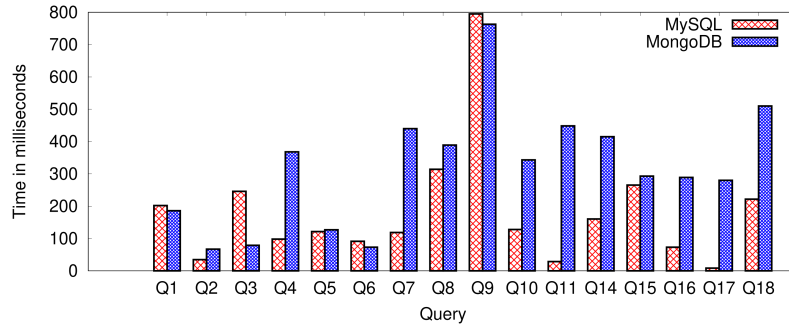


Figure 20: Performance of MongoDB vs MySQL

## 7 Conclusion

NoSQL gives an alternative solution to relational databases. They are unstructured or semi-structured and no schema required. These two key features make them desirable in the nowadays fast-paced environment, where the data is unstructured and the volume is too big. Relational databases are vertically scalable but typically expensive. Scaling a NoSQL database is much cheaper, compared to a relational database, because you can add capacity by scaling horizontally over cheap, commodity servers. Moreover, document stores resolve the issue of impedance mismatch and no object-relational mapping is needed.

MongoDB is one of the most popular document stores nowadays. It offers a wide variety of features and has embraced the ease and philosophy of NoSQL solutions and document stores. However, as the results of these benchmark present, more work is needed to be done regarding its performance. Although in specific cases, it might outperform relational databases, in general, the relational solutions on the market (particularly PostgreSQL and MongoDB) are faster.

Moreover, I would like to point out the fact that choosing the best design for your document store application is a difficult task that is not standardized



like in the case of relational databases. Choosing the best design is carried out in an ad-hoc manner with a trial and error approach. This task might have a significant impact on performance.

As far as I can tell, their application will become popular hence the promise of horizontal scaling. Moreover, is a good match for the web world. There is no way to predict if an application will be popular or not. So built it as quickly as possible, expose it to the public. If it gains in popularity, quickly rebuild it so it is robust. In that world, RDMBS requires more up-front engineering work, so more engineering work is thrown away.

## References

- [1] Transaction Processing Performance Council (TPC). “TPC BENCHMARK H”. In: (2014).
- [2] *A Comparison of NoSQL Database Management Systems and Models*. <https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models>.
- [3] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. “Experimental evaluation of NoSQL databases”. In: *International Journal of Database Management Systems* 6.3 (2014), p. 1.
- [4] *Amazon DynamoDB*. <https://aws.amazon.com/dynamodb/>.
- [5] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. ” O’Reilly Media, Inc.”, 2013.
- [6] *Couchbase System Properties*. <https://db-engines.com/en/system/Couchbase>.
- [7] *DB-Engines Ranking of Document Stores*. <https://db-engines.com/en/ranking/document+store>.
- [8] Robert T Mason. “NoSQL databases and data modeling techniques for a document-oriented NoSQL database”. In: *Proceedings of informing science and IT education conference (InSITE)*. 2015, pp. 259–268.
- [9] *MongoDB System Properties*. <https://db-engines.com/en/system/MongoDB>.
- [10] ABM Moniruzzaman and Syed Akhter Hossain. “Nosql database: New era of databases for big data analytics-classification, characteristics and comparison”. In: *arXiv preprint arXiv:1307.0191* (2013).

- [11] *NDocument-oriented database*. [https://en.wikipedia.org/wiki/Document-oriented\\_databaseRelationship\\_or\\_relational\\_databases](https://en.wikipedia.org/wiki/Document-oriented_databaseRelationship_or_relational_databases).
- [12] *NoSQL Databases Explained*. <https://www.mongodb.com/nosql-explained>.
- [13] *NoSQL Performance Benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB*. <https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/>.
- [14] *NoSQL Performance Benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB*. <https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/>.
- [15] *NoSQL vs SQL Databases*. <https://www.mongodb.com/scale/nosql-vs-relational-databases>.
- [16] *Object-Relational Mapping (ORM)*. <https://www.techopedia.com/definition/24200/object-relational-mapping-orm>.
- [17] *Pokec social network*. <https://snap.stanford.edu/data/soc-pokec.html>.
- [18] Dan Sullivan. *NoSQL for mere mortals*. Addison-Wesley Professional, 2015.
- [19] *TPCH IN MONGODB*. <https://mongodb2017.wordpress.com/>.