TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics
Chair of Information Systems

# Performance Comparison of MongoDB and PostgreSQL with JSON types

Master's Thesis

Student: Dmitri Maksimov
Student's code: 121839IAPM

Supervisor: Erki Eessaar
Associate Professor

Tallinn
2015

**Autorideklaratsioon**

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

…….  ……

(kuupäev)  (allkiri)

# Annotation

*Performance Comparison of MongoDB and PostgreSQL with JSON types*

Traditional SQL database management systems (DBMSs) and NoSQL systems are different. SQL DBMSs on the one hand feature a data model that requires explicit definition of database schema. In addition, they allow its users to ensure consistency of data as well as support transactions that span multiple statements and have strict properties (ACID). On the other hand, this focus on strict consistency makes them hard to scale horizontally and the system do not perform well under the high load of large data volumes or/and concurrent access. Whereas NoSQL systems offer great capabilities of horizontal scaling, flexible data models that do not require explicit schema definition at the database level, and good performance under the high load of large data volumes or/and concurrent access. On the other hand, NoSQL systems generally place less strict requirements to transactions (BASE), making the systems built on top of these systems prone to the data inconsistencies. Many of them treat each single operations as an atomic transactional unit but do not allow developers to group multiple statements (operations) to one atomic unit (transaction). Consequently, software developers must continuously choose between data consistency and performance when developing a new product.

The introduction of JSON data types in PostgreSQL, one of the most widely used traditional SQL DBMS, has given rise to debates as to whether one can use SQL systems in a manner that offers the data model's flexibility and performance of NoSQL systems, while still being able to use many advantages and advanced features that the SQL systems offer.

The goal of this work is to create a set of benchmarks and to measure the performance of PostgreSQL with JSON types and compare it with the performance of a NoSQL system (MongoDB in particular) from the perspective of a real world Java application. The use of PostgreSQL with JSON types means that database contain base tables (tables) with JSON or JSONB types. The performance measurement is conducted from the viewpoint of database usage by modern software systems meaning that connection with a DBMS is not established through some console application but instead by using some API. The target audience of this performance measurement is software developers who write programs in Java. The measurements will be made by using PostgreSQL 9.4.1 and MongoDB 3.0.2.

## Annotatsioon

*MongoDB ja JSON tüüpe kasutava PostgreSQLi jõudluse võrdlemine*

Traditsioonilised SQL-andmebaasisüsteemid ja NoSQL süsteemid on erinevad. SQL-andmebaasisüsteemid pakuvad andmebaasi skeemi ilmutatud kujul defineerimist nõudva andmemudeli. Samuti võimaldavad need oma kasutajatel tagada andmete terviklikkust ning aitavad saavutada transaktsioonidele kehtestatud rangete nõudmiste täidetuse (ACID). Teisest küljest muudab andmete terviklikkuse range tagamine süsteemide horisontaalselt skaleerimise keerulisemaks ning suurte andmemahtude ja samaaegsete kasutajate arvu korral väheneb oluliselt süsteemi jõudlus. NoSQL süsteemid pakuvad horisontaalse skaleeruvuse võimekust, paindlikke andmemudeleid, mis ei nõua andmebaasi tasemel andmebaasi skeemi ilmutatud kujul defineerimist ning head töövõimet suurte andmemahtude või/ja samaaegsete kasutajate arvu korral. Samas esitavad NoSQL süsteemid transaktsioonidele enamasti leebemad nõudeid, mis muudab nende abil realiseeritud süsteemid altiks andmete terviklikkuse rikkumise suhtes (BASE). Samuti käsitlevad paljud NoSQL süsteemid iga andmemuudatuse operatsiooni atomaarsena, kuid ei võimalda koondada mitut operatsiooni kokku üheks atomaarseks tervikuks (transaktsiooniks). Sellest lähtuvalt peavad tarkvaraarendajad tegema pidevalt valikuid andmete terviklikkuse ja süsteemi jõudluse vahel.

JSON andmetüüpide kasutuselevõtt PostgreSQL-is, kui ühes kõige laialdasemalt kasutatavas traditsioonilises SQL-andmebaasisüsteemis, on tekitanud arutelu, kas SQL süsteeme saaks kasutada nii, et pannakse kokku NoSQL süsteemide andmemudelite paindlikkus ning nende süsteemide jõudlus, saades samal ajal endiselt osa SQL süsteemide paljudest eelistest.

Käesoleva töö eesmärk on luua mõõtlusaluste kogum ning mõõta JSON tüüpe pakkuva PostgreSQL jõudlust ja võrrelda seda NoSQL süsteemide jõudlusega. Jõudluse mõõtmine toimub ühe reaalse Java rakenduse näitel. PostgreSQL-i koos JSON tüüpidega kasutamine tähendab, et andmebaasis on baastabelid (tabelid), kus on JSON või JSONB tüüpi veerud. Mõõtmisi teostatakse viisil, mis arvestab andmebaaside kasutamisega tänapäevaste tarkvarasüsteemide poolt. See tähendab, et andmebaasisüsteemiga ühenduse loomiseks ei kasutata mitte konsoolirakendusi, vaid mõnda programmiliidest. Selle töökiiruse uuringu sihtrühmaks on tarkvara arendajad, kes kirjutavad rakendusi Javas. Mõõtmisi tehakse PostgreSQL 9.4 ja MongoDB 3.0.2 põhjal.

## Abbreviations and a Glossary of Terms

| | |
|---|---|
| **ACID** | The model of transactions support in DBMS featuring **A**tomicity, **C**onsistency, **I**solation, and **D**urability. |
| **API** | Application Programming Interface – "collection of invocation methods and associated parameters used by one piece of software to request actions from another piece of software" [1] |
| **BASE** | The model of transactions support in DBMS featuring **B**asic **A**vailability, **S**oft state, and **E**ventual consistency. |
| **Benchmark** | "A procedure, problem, or test that can be used to compare systems or components to each other or to a standard" [2] |
| **BSON** | Binary JSON – "a binary-encoded serialization of JSON-like documents." [3] |
| **CMS** | Content Management System – "a piece of software that is used to organize, manage or change the content of a website." [4] |
| **DAO** | Data Access Object – a software design pattern that provides an abstract interface to persistence mechanism. |
| **DBMS** | DataBase Management System – "a collection of integrated services which support database management and together support and control the creation, use and maintenance of a database" [5] |
| **EnterpriseDB** | The leading worldwide provider of PostgreSQL software. [6] |
| **EntityManager** | Interface used to interact with the persistence context in JPA specification. |
| **GitHub** | Web based source code repository used to publish open-sourced projects. |
| **Hibernate** | Object-relational mapping framework for the Java language. |
| **Horizontal Scalability** | Ability to add more machines into the pool of resources, thus increasing system's overall performance. |
| **Java** | Object-oriented programming language supported by Oracle Corporation. |
| **JDBC** | Java DataBase Connectivity – a Java API that can access any kind of tabular data, especially data stored in a Relational/SQL Database. [7] |
| **JIT** | Just-In-Time compiler – a compiler that performs source code compilation during the execution of the program. |
| **JMH** | Java Microbenchmarking Harness framework – a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java [8] |

| | |
|---|---|
| **JPA** | Java Persistence API – "the Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications." [9] |
| **JSON** | JavaScript Object Notation – data interchange/representation format. |
| **JSONB** | Data type introduced by PostgreSQL. Differs from JSON by that the data is stored in a decomposed binary format. |
| **JVM** | Java Virtual Machine –n abstract computing machine, used to execute and run Java applications. [10] |
| **MMAPv1** | MongoDB's default storage engine. |
| **MongoDB** | An open-source document database and the leading NoSQL database. [11] |
| **ORM** | Object-Relational Mapping – the technique of bridging the gap between the object model and the relational/SQL model. [12] |
| **POJO** | Plain Old Java Object – a regular Java object. |
| **Polyglot Persistence** | The idea of using different data storage technologies for different kinds of data in the scope of one application. [13] |
| **PostgreSQL** | One of the most widely used SQL database management systems that is also very sophisticated and is open-source. |
| **PostgreSQL operator '->'** | Operator for accessing JSON object field or JSON array element through the SQL query. |
| **SQL** | Structured Query Language – standardized database language for managing databases. ISO/IEC 9075 [14]. |
| **SQL Dialect** | An implementation of SQL standard that is specific to a particular SQL DBMS. |
| **SQL Join** | A generic operator that combines rows from two or more tables in a SQL database based on a search condition (join condition; Boolean value expression). |
| **WiredTiger** | Storage engine that was included into release of MongoDB 3.0. It offers an improved performance and data compression as opposed to MMAPv1. |

## List of Figures

## List of Tables

# Table of Contents

## Introduction

Enterprise level software development nowadays, at the era of the Web applications, is impossible without making them to use databases. Every e-shop, blog, entertainment portal, CMS (Content Management System) based sites, etc. is using the services of a database management system (DBMS) (either directly or through cloud) for data management. In each product, there could be parts that have to process different amounts of stored data, that have different amounts of simultaneous data access, and that have different requirements for data processing. These aspects of the data management immensely influence on the application's overall performance. Moreover, application's database layer often becomes a performance bottleneck. Therefore, when building an application, it is extremely important to provide the best possible performance at the database layer.

The rapid growth of the Web, the size of stored data, and the number of users has revealed that current mainstream (traditional) SQL DBMSs are not capable to provide a suitable performance and scalability. This makes industry-leading companies, such as Google, Facebook, Amazon, and EBay, to search new solutions for the modern data management. One line of research has led to the NoSQL systems. These systems are developed with good scalability, data model flexibility, and performance of data management operations in mind. However, some of the habitual features of DBMSs were dropped in order to achieve the goal. Support to strict form of transactions that satisfy ACID properties are one of these. Consequently, nowadays when it is necessary to choose a DBMS for an application, one is choosing between SQL and NoSQL systems, making tradeoffs between performance, scalability, and ease of development.

PostgreSQL tried to offer some capabilities of the NoSQL systems with introduction of JSON data type in year 2012 and more sophisticated JSONB data type in 2014. In this sense PostgreSQL has become a multi-model system meaning that it makes it possible to build up databases by using building blocks of different data models (tables and their surrounding ecosystem of SQL; documents of the document-oriented model).

One may think of this as a revolutionary development, because one of the oldest SQL systems now offers the flexibility of a documents-based NoSQL data model. Furthermore, the leading worldwide provider of PostgreSQL software presented a performance comparison of PostgreSQL (with JSON data types) with a leading NoSQL system MongoDB. It shows that PostgreSQL can outperform NoSQL system in case of both data reading and data writing.

The goal of this thesis is to create a set of benchmarks and to test what performance PostgreSQL with JSON data types can offer for a real application written in Java. Moreover, the goal is to verify as to whether the PostgreSQL can actually outperform MongoDB as some recent benchmarking results claim. The use of PostgreSQL with JSON types means that database contains base tables (tables) with JSON or JSONB types. The performance measurement is conducted from the viewpoint of database usage by modern software systems meaning that connection with a DBMS is not established through some console application but instead by using some API. The target audience of this' performance measurement is software developers who have to create Java programs that interact with a DBMS in this way. PostgreSQL 9.4.1 and MongoDB 3.0.2 will be used for the testing. The general idea of creating benchmarks and testing the performance of DBMSs from the point of view of APIs should be useful to any developer regardless of a programming language or a DBMS.

The thesis is organized as follows. The first chapter will briefly cover the theoretical background that stands behind SQL and NoSQL systems and Java specific technologies for database communication. The second chapter will describe the performance comparison of PostgreSQL and MongoDB that has been done by EnterpriseDB (the leading worldwide provider of PostgreSQL software). Moreover, the analogous performance comparison of PostgreSQL and MongoDB most recent versions will be done with the same performance benchmarking method. The third chapter will give an overview of the Java application's database layer, which will serve as a platform for experimenting and benchmarking. It will define design of the possible database layer implementations using PostgreSQL with JSON data types and MongoDB. Additionally, the performance measurement method will be described as well as a brief description of how benchmarks were written by the author. The fourth chapter will thoroughly describe the experiments and benchmarks, which will be based on the specified designs. The fifth chapter will include the analysis of the benchmarking results, providing summary tables and charts. In this chapter, the author will explain which design has the best performance. Finally, the summary of the work will be presented and future work offered.

# 1 Theoretical Background

Comparison of different database management systems, their features, and applications assumes that concepts and ideas behind these systems are clear and understandable. This chapter will try to describe, without going deeply into details, primary properties, pros and cons of classic SQL Database Management Systems (DBMS) and NoSQL systems. As the present work will focus on performance measurements and usage of databases in real world application written with Java programming language, possibilities of interoperability between Java applications and databases are also covered.

## 1.1 NoSQL Systems

What is NoSQL? As it turned out, this term has no strong definition; it is just a commonly used name for a set of DBMSs that have different ideas compared to SQL and relational systems about how to organize, present, store, and retrieve data. Martin Fowler has outlined some common, but not definitional, characteristics of NoSQL systems [15]:

- Not using the relational data model (nor the SQL language)
- Open source
- Designed to run on large clusters
- Based on the needs of 21st century web properties
- No explicitly defined schema, allowing fields to be added to any record without controls. Of course, the data must have schema or otherwise the users would not know how to use the data but this knowledge of schema resides in the application code that accesses the database.

The most significant characteristic, in which NoSQL systems differ from SQL DBMSs, is their underlying data model that determines building blocks of databases, possible generic operations in these databases, and possible types of constraints that the system can enforce in case of the databases. Although there are dozens of NoSQL systems, their data models primarily fall into one of the following four categories [16] [17]:

- **Key-Value Models** – Every item in the database is stored as an attribute name or key, together with its value. The value, however, is entirely opaque to the system. It means that data can only be queried by the key.
    - Applications: Narrow set of applications that only query data by a single key value. The appeal of these systems is their performance and scalability, which can be highly optimized due to the simplicity of the data access patterns.
    - Examples: Riak, Redis, Memcached, Berkeley DB, Couchbase, etc.

- **Wide Column Models** – Wide column stores or column family stores, use a sparse, multi-dimensional sorted map to store data. Each record can vary in the number of columns that are stored and one can nest columns inside other columns called super columns. One can group columns together for access in column families or can spread columns across column families. Data is retrieved by primary key per column family.
    - Applications: Narrow set of applications that only query data by a single key value.
    - Examples: Cassandra, HBase, Hypertable.
- **Graph Models** – Graph-based systems use graph structures with nodes, edges and properties to represent data. In essence, data is modeled as a network of relationships between specific elements. Its main appeal is that it makes it easier to model relationships between entities in an application.
    - Applications: Graph databases are useful in cases where relationships are core to the application, like social networks.
    - Examples: Neo4j, HyperGraphDB, OrientDB, or FlockDB.
- **Document Models** – Documents-based systems store data in documents. These documents typically use a structure that is like JSON, XML, BSON, and so on. Documents contain one or more fields, where each field contains a typed value, such as a string, date, binary, or array. Rather than spreading out a record across multiple columns and tables, each record and its associated data are typically stored together in a single document. This simplifies data access and reduces or even eliminates the need for joins and complex transactions.
    - Applications: A wide variety of applications due to the flexibility of the data model, the ability to query on any field, and the natural mapping of the document data model to objects in modern programming languages.
    - Examples: MongoDB, CouchDB, Terrastore, OrientDB, RavenDB.

### 1.1.1 MongoDB

MongoDB was chosen in this thesis as a representative of the world of NoSQL systems for several reasons.

- MongoDB is the most popular NoSQL DBMS according to Knowledge Base of Relational and NoSQL Database Management Systems in April 2015 [18].
- MongoDB is an open-source document DBMS that provides high performance, high availability, and automatic scaling.
- It uses JSON structure for its documents.

- Learning curve: it has a very solid and informative documentation; it has a whole MongoDB University with free courses such as M101J: MongoDB for Java Developers (which author of the thesis successfully completed).

Key features of MongoDB are [19]:

**High Performance**

MongoDB provides high performance data persistence. In particular, it provides the following.

- Support for embedding related data into one document (embedding a document within a document) reduces I/O activity on DBMS. In case of consuming such documents, applications may need to issue fewer queries and updates to complete common operations. Generally, embedding provides better performance for read operations. This kind of organization of data makes it possible to update related data in a single atomic write operation [20].
- Indexes support faster queries and can include keys from embedded documents and arrays.

**High Availability**

To provide high availability, MongoDB's replication facility, called replica sets, provide the following.

- Automatic failover.
- Data redundancy.

A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

**Automatic Scaling**

MongoDB provides horizontal scalability as a part of its core functionality.

- Automatic sharding distributes data across a cluster of machines.
- Replica sets can provide eventually-consistent reads for low-latency high throughput deployments.

## 1.2 SQL Database Management Systems

Each SQL Database Management System (SQL DBMS) is a DBMS where one can use SQL database language and can build up databases based on the underlying data model of SQL (SQL model). This data model has been created based on the relational data model.

SQL/relational and NoSQL data models are very different. SQL model presents data in tables. Tables are organized into columns, and each column stores one type of data (integer, real number, character strings, date, etc.). Each row of a table represents a true proposition of some portion of the world and a table is collection of such propositions that all conform to the same predicate represented by the table heading that determines the meaning of the table. A difference from NoSQL systems is that one has to explicitly define the database schema.

Tables reference each other through foreign keys that values are stored in columns as well. The SQL/relational model lends it very well to building up databases and DBMSs as layered systems to separate concerns [21]. External level contains views to the data. Conceptual level specifies base data structures, constraints to these, and generic high-level operations to access data. Internal level deals with data storage and implementing algorithms for executing the high-level operations.

The SQL/relational model allows us to potentially minimize the amount of required storage space, because databases can be designed in a manner that minimizes data redundancy at the logical database level by applying the normalization theory and the principle of orthogonal design. Normalization theory deals with redundancy within one table and the orthogonal design theory across tables. How much their application influence the data size on disk depends on the implementation of the DBMS. It could be that at the logical level, redundancy is almost eliminated but at the internal database level there is a lot of redundancy to speed up certain database operations.

Reduction of a storage space at the internal database level was once very important, when disk storage was very costly. Reduction of redundancy at the logical database level is important because it helps us avoid inconsistencies, separate unrelated facts from each other, simplify enforcement of certain integrity constraints, and increase the speed of certain update operations. However, redundancy reduction comes at expense of increased complexity when looking up data. The desired information has to be collected from many tables and combined before it can be provided to the application. Similarly, when writing data, the write needs to be coordinated and performed on many tables. This complexity though can be resolved by implementing a virtual data layer that presents denormalized data to applications that need the data. This virtual

data layer belongs to the external database level. By making the views updatable, it also simplifies the update operations.

**Some advantages of a SQL DBMS** (at least those that are important in the current thesis):

- **Efficient storage** – Separation of conceptual and internal database level allows implementers of DBMSs to use advanced techniques like packing to reduce the footprint of data on disk.
- **Avoidance of update anomalies** – The data model makes it possible to design databases in a manner that reduces data duplication and thus also the number of update anomalies. The process is guided by the Database Normalization theory and the orthogonal database design principle [22]. The update anomalies occur if replacing the value in a field of a row requires replacing values in other fields as well or inserting/deleting a fact is only possible if some logically unrelated fact in the context of this operation is also inserted/deleted. Update anomalies could cause inconsistent data, burden the system, and complicate application development.
- **Transactions** – transaction is an atomic unit consisting of one or more operations. The results of all operations may be applied (committed) or refused (rolled back), if one of the operations fail or on any other reason.
- **Declarative data manipulation language with different generic relational operators like join or union** – a possibility to retrieve data from one more tables by declaring the expected outcome not the low-level algorithm how to achieve the desired result.
- **Tools** – due to a long-lasting domination of the SQL DBMSs there are a huge variety of different tools and utilities, which offer a great support to the users of these systems. Hibernate ORM is, probably, the most significant one for Java developers.

### 1.2.1 PostgreSQL JSON Data Types

Although most popular SQL DBMS's support a wide variety of system-defined data types such as text, numbers, Boolean, or even arrays of the values with these types, it was impossible to store a set of arbitrary values as a value of one column. The database schema ensures that each field must contain at most one value of the data type of the column. If the DBMS provides a limited set of system-defined types and does not allow us to define new types or makes it too complex by placing on it unnecessary restrictions, then it reduces the flexibility of the usage of the data model.

This was somewhat changed by PostgreSQL since the introduction of the JSON data type in release 9.2 in 2012 [23] and further improvement of this and introduction JSONB data type in

release 9.4 in 2014 [24]. Of course, one can assume that it is possible to store JSON data in columns of the text data type, but PostgreSQL's JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules.

There are two JSON data types: *json* and *jsonb* in PostgreSQL. Each type is a named set of values. These types contain almost identical sets of values. The major practical difference is one of efficiency in case of internal representation of the values. In case of *json* data type the system stores an exact copy of the input text, which processing functions must reparse on each execution. On the other hand, *jsonb* data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. *jsonb* also supports indexing, which can be a significant advantage [25].

These JSON types bring the advantage of database schema flexibility of NoSQL databases to the SQL database. Of course, they also bring with them a problem that data could be duplicated in different json values at the logical database level that makes it easier to register inconsistent data. To avoid that, programmers have to do more work and system has to do more work to replace multiple json values with one operation. One could create views that present to data consumers (applications) json values that are constructed on the fly based on values taken from the base tables that do not contain such values. Investigation of such views is not a task of this thesis.

## 1.3 Java and Databases

Every modern programming language has support of connectivity and communication with databases. Nevertheless, Java platform was selected for the current thesis because the author has a strong knowledge of Java language and Java-based technologies.

In this section, different possibilities of communication between Java application and databases are overviewed.

### 1.3.1 JDBC

JDBC or Java DataBase Connectivity is a Java built-in functionality for the communication with SQL DBMSs. It is a "low-level" API that makes it possible to use databases by applications.

Offering a simple and portable abstraction of the proprietary client programming interfaces offered by database vendors, JDBC allows java programs to fully interact with the DBMS (and by doing it use the databases). This interaction is heavily reliant on SQL, offering developers a

chance to write queries and data manipulation statements in the database language. These statements are executed and processed by using a simple Java programming model [12]. Caveat is that although JDBC's programming interfaces are portable, the SQL is not. Despite the ongoing attempts to standardize it, every SQL DBMS is using its own dialect of the SQL.

JDBC helps developers to conduct the following activities in the Java applications.

1) Connect to a database
2) Send queries and update statements to the database
3) Retrieve and process the results received from the database in answer to a query

Pros of using JDBC in applications.

- Performance – because JDBC is the lowest possible abstraction level for interacting with a DBMS, it should provide the best performance in means of communication with the DBMS.
- Control – due to the same reason, developer is in control of every detail of communication with DBMS, be this query composition, data processing, or a transaction management.

Cons of using JDBC in applications.

- Developer productivity and ease of development – developer should write a lot of code to perform a little. Data retrieved from a database should generally be manually transformed to the business objects. It becomes a tedious work, when there are many relationships/associations between objects, such as One-to-Many, Many-to-One, and/or Many-to-Many.
- Portability – using JDBC usually means that application is bound to a particular SQL DBMS.
- No cache.

However, the JDBC is not the only option.

### 1.3.2 ORM and JPA

"The domain model has a class. The database has a table. They look pretty similar. It should be simple to convert one to the other automatically." [12] The technique of bridging the gap between the object model and the relational/SQL model is known as object-relational mapping or simply ORM. The term comes from the idea that concepts from one model are mapped onto

another, with the goal of introducing a mediator to manage the automatic transformation of one to the other [12].

Following these concepts several commercial and open-source products were developed, such as Hibernate or TopLink.

Hibernate, for instance, takes care of the mapping from Java classes to database tables and provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling with SQL and JDBC. Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC [26].

As Hibernate, and other persistence APIs became ensconced in applications and met the needs of the application very well, the need of one standardized API arose. This API is the Java Persistence API or JPA.

The Java Persistence API deals with:

- the way SQL data is mapped to Java objects ("persistent entities"),
- the way that these objects are stored in a SQL database so that they can be accessed at a later time,
- the continued existence of an entity's state even after the application that uses it ends.

In addition to simplifying the entity persistence model, the Java Persistence API standardizes object-relational mapping [27].

Pros of using ORM and JPA in applications.

- Ease of development – object-relational mapping with Hibernate and JPA is completely metadata-driven and uses domain model's POJOs – Plain Old Java Objects. JPA automatically generates SQL queries and manages entity dependencies. Developer should not write conversions between domain objects and database tables as it is being done automatically by JPA.
- Portability – JPA generates queries according to the specified SQL dialect.
- Cache – JPA uses caching a lot. It takes place transparently for a developer.

Cons of using ORM and JPA in applications.

- Performance – a slightly worse performance comparing with pure JDBC. JPA is the ORM framework based on JDBC, thus it adds load of data processing, query generation,

and so on. In addition, automatically generated queries are often not the most optimized ones.

### 1.3.3 MongoDB Java Driver

Standard tools, such as JDBC, Hibernate, and JPA are not suitable for the usage with non-SQL databases (at least at the moment of writing this thesis in the spring of 2015). That is, vendors of NoSQL systems should provide their own facilities for programming languages to make possible communication with their products.

MongoDB Java Driver is a library provided by MongoDB team for Java platform. It is similar to JDBC, with the difference that it has different API, and allows to connect only to MongoDB DMBS.

MongoDB driver, as well as JDBC, facilitates management of the following three programming activities.

1) Connecting to a database.
2) Sending queries and update statements to the database.
3) Retrieving and processing the results received from the database in answer to a query.

## 1.4 ACID *vs*. BASE

In the context of using database management systems in software development, transactions are very important. The aspect of how transactions are managed by a DBMS influences the performance and code complexity of programs that use the services of the DBMS. Thus, it is important to understand the principles standing behind transactions in SQL and NoSQL systems.

### ACID

Mostly every SQL DBMS supports transactions with the ACID qualities. These qualities are as follows [28].

- **Atomicity** – Either the task (or all tasks) within a transaction are performed or none of them are from the point of view of invoker. If one element of a transaction fails, the entire transaction fails.
- **Consistency** – The transaction must meet all protocols or rules defined by the system at all times. These rules are expressed in terms of constraints implemented in the system in a declarative or procedural manner. All the transactions must not violate those protocols and the database must remain in a consistent state as the result of a transaction.

- **Isolation** – No transaction should see data changes in a database made by an incomplete transaction. This is required for consistency of data within a database. SQL defines four different isolation levels. The higher is isolation the better it ensure consistency but also the more it starts to hamper performance because of locking of data elements that is typically used to achieve isolation.
- **Durability** – Once the transaction is complete, it will persist as complete, and cannot be undone; it will survive system failure, power loss and other types of system breakdowns.

However, providing all these properties means that the system becomes incompatible with availability and performance in very large systems. The main reason is that support of ACID features means that the system is going to have limited capabilities of processing concurrent requests. Additionally, a horizontal scalability becomes a non-trivial task while still providing all ACID features.

**CAP Theorem**

The CAP Theorem was presented by Eric Brewer in 2000. "The central tenet of the theorem states that there are three essential system requirements necessary for the successful design, implementation and deployment of applications in distributed computing systems. They are Consistency, Availability and Partition Tolerance – or CAP" [28].

- **Consistency** refers to whether a system operates fully or not. Does the system reliably follow the established rules within its programming according to those defined rules? Do all nodes within a cluster see the most recently written data they are supposed to see?
- **Availability**. Is the given service or system available when requested? Does each request get a response as to whether it is failure or success within reasonable amount of time?
- **Partition Tolerance** represents the fact that a given system continues to operate even under circumstances of partial data loss or partial system failure. A single node failure should not cause the entire system to collapse.

"Theorem: You can have at most two of these properties for any shared-data system". The theorem says that if one is willing to have the Consistency and the Availability (ACID), the Partition Tolerance should be left out. This is the case in systems where data is in one node (server). However, in case of a distributed system, partition tolerance is usually expected and

thus the choice is between relaxing consistency or availability requirements. One should note that different parts of the same system might require different decisions in terms of CAP theorem. For instance, in case of some parts availability is more important in case of other consistency is more important.

**BASE**

The BASE is an alternative to the ACID providing the Availability and the Partition Tolerance. These are principles adopted by most of the NoSQL systems (MongoDB in particular). The idea is that it is enough for the database to eventually be in a consistent state, rather than requiring consistency after every transaction. BASE principles are characterized as follows [28].

- **Basic Availability** – the system does guarantee the availability of the data as regards CAP Theorem; there will be a response to any request. Nevertheless, that response could still fail to obtain the requested data or the data may be in an inconsistent or changing state.
- **Soft state** – the state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft'.
- **Eventual consistency**: The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

From the perspective of software developer, the main difference between ACID and BASE transactions is that in case of ACID it is possible control transactions manually. Thus, it is possible to issue several data operation requests in scope of one transaction. Whereas in case of BASE transactions, the one data operation request means the one transaction, and there is no possibility to obtain more control over it.

# 2 PostgreSQL JSON Data Type *vs.* MongoDB

## 2.1 Performance

The introduction of the JSON and JSONB data types allowed developers to use PostgreSQL as a NoSQL system. EnterpriseDB (the leading worldwide provider of PostgreSQL software) even carried out a series of performance tests showing that PostgreSQL as a NoSQL system can outperform MongoDB.

The initial set of tests by EnterpriseDB compared MongoDB v2.6 to PostgreSQL v9.4 beta, on single machine instances. Both systems were installed on Amazon Web Services M3.2XLARGE instances with 32GB of memory. EDB found that PostgreSQL outperforms MongoDB in selecting, loading, and inserting complex document data in key workloads involving 50 million records [29].

- Ingestion of high volumes of data was approximately 2.1 times faster in PostgreSQL
- MongoDB consumed 33% more the disk space
- Data inserts took almost 3 times longer in MongoDB
- Data selection took more than 2.5 times longer in MongoDB than in Postgres

Chart on Figure 1 shows relative performance comparison in terms of the amount of time spent on execution of corresponding operations. The higher value means that more time was spent. Therefore, the higher value is worse.

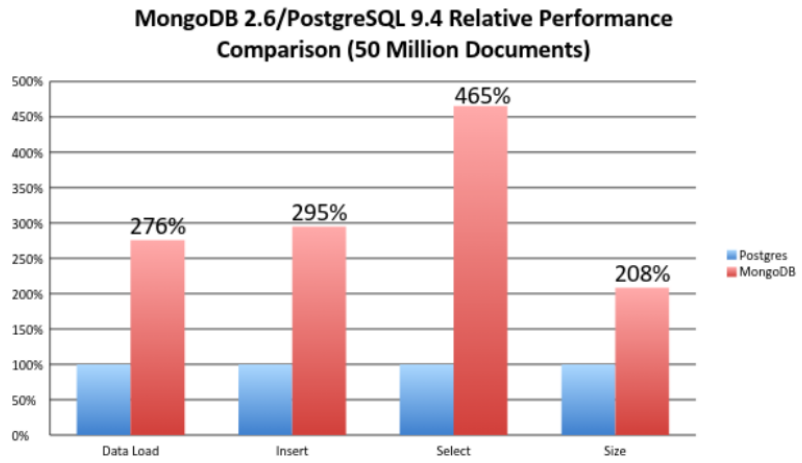


**Figure 1. MongoDB 2.6 and PostgreSQL 9.4 Relative Performance Comparison [29]**

These are amazing results for PostgreSQL. However, at the moment of writing the thesis (spring 2015) the current version of MongoDB is 3.0.2. According to the MongoDB 3.0 release announcement, it brings massive improvements to performance and scalability, enabled by

comprehensive improvements in the storage layer [30]. Moreover, support for different storage engines was added. Thus, the improved version of the default engine MMAPv1 and the new one WiredTiger are available out-of-box. The following was promised in the announcement:

- 7-10x better write performance,
- Up to 80% less storage with compression (only using WiredTiger).

The MongoDB 2.6 and PostgreSQL 9.4 Relative Performance Comparison was made with the open-sourced testing framework "*pg_nosql_benchmark*". The framework is developed by EnterpriseDB and publicly available at EnterpriseDB's GitHub repository [31].

The introduction abstract, describing the framework, from the GitHub repository's Web page is the following [31]:

"This is a benchmarking tool developed by EnterpriseDB to benchmark MongoDB 2.6 (BSON) and Postgres 9.4 (JSONB) database using JSON data. The current version focuses on data ingestion and simple select operations in single-instance environments - later versions will include a complete range of workloads (including deleting, updating, appending, and complex select operations) and they will also evaluate multi-server configurations.

This tool performs the following tasks to compare of MongoDB and PostgreSQL:

- The tool generates a large set of JSON documents (the number of documents is defined by the value of the variable *json_rows* in *pg_nosql_benchmark*)
- The data set is loaded into MongoDB and PostgreSQL using *mongoimport* and PostgreSQL's COPY command.
- The same data is loaded into MongoDB and PostgreSQL using the INSERT command.
- The tool executes 4 SELECT Queries in MongoDB & PostgreSQL."

The framework firstly generates the defined in configuration amount of JSON test data, then it generates PostgreSQL and MongoDB specific data for insertion based on JSON test data.

At the moment of writing this thesis (April 2015), there was no MongoDB 3.0 and PostgreSQL 9.4 performance comparisons found on the Web. Consequently, the author decided to perform such comparison as a part of this thesis. The same testing framework was used, although it required a couple of minor changes in order to work with the latest MongoDB.

What were the minor changes in the framework?

The "pg_nosql_benchmark" framework consists of one Linux Bash shell base script *"pg_nosql_benchmark"* and a set of functions in separate files *"common_func_lib.sh"*, *"mongo_func_lib.sh"*, and *"pg_func_lib.sh"*. It was necessary to change parameters (such as usernames, passwords, and network addresses of PostgreSQL and MondoDB servers) in the base script, in order to get the script working in the local environment. A problem appeared with the size value of the disk space used by MongoDB. The reason is that newer version of MongoDB has changed format of its command output. The issue was fixed by the author. The problem was that one of the functions in *"mongo_func_lib.sh"* expected the output from MongoDB command execution in one format. However, the output format of the same command in the new version has changed.

Tests of this thesis were made on Lenovo ThinkPad laptop with Intel Core i5-3320M @ 2.60Hz, 8GB RAM, 120GB SSD storage, Ubuntu 14.04 LTS 64-bit, PostgreSQL 9.4.1, and MongoDB 3.0.2. These characteristics allowed setting the maximum number of documents to around one million.

Chart on Figure 2 illustrates the results of the tests made by the author of the thesis. The higher value means that more time was spent by the DBMS. Therefore, the higher value is worse.

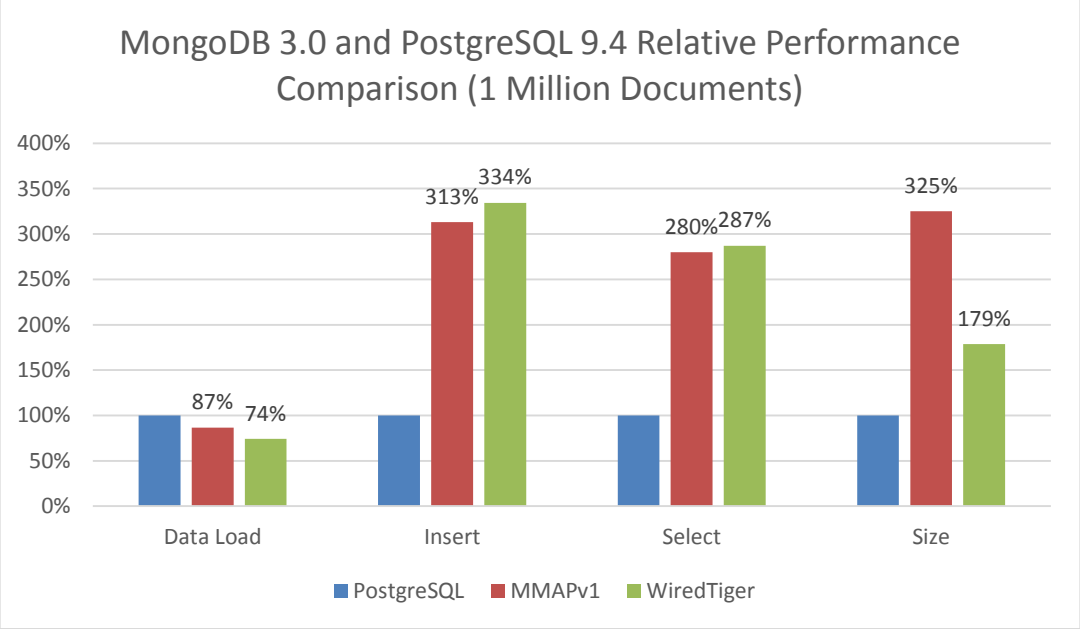


**Figure 2. MongoDB 3.0 and PostgreSQL 9.4 Relative Performance Comparison**

The results of the tests showed that PostgreSQL still outperforms MongoDB in selecting and inserting complex document data.

- Data insertion still takes almost three times longer in MongoDB.
- Data selection takes almost three times longer in MongoDB than in PostgreSQL.

- MongoDB consumes three times more the disk space with MMAPv1 storage engine and 1.8 times more with WiredTiger engine. Data compression in WiredTiger is working well.

However, data loading became faster in MongoDB and outperformed PostgreSQL.

**Performance benchmarking recap**

The MongoDB and PostgreSQL relative performance comparison showed that PostgreSQL is generally *two to three times* more performant than MongoDB. However, benchmarking tool, which the author used for the benchmarking, is created and distributed by EnterpriseDB. This company provides support and tools for PostgreSQL. In addition, the tool is using console applications provided by PostgreSQL and MongoDB. These are *psql* and *mongo*, respectively. The tool calls separate console commands for every action. Thus, there is a probability that performance of these console applications may significantly influence benchmarking results. Moreover, usually console applications are used for databases maintenance, debugging, etc., not for software development.

Real world applications are using databases through the provided API, thus the real performance measurement should be made through the API by using applications.

## 2.2 Field of Application

Every tool should be properly used in right scenarios in order to get the best results. This is true for the DBMSs as well.

When to use a NoSQL (MongoDB in particular) [32] system?

- **High Write Load is Expected** – MongoDB by default prefers high insert rate over transaction safety. If it is necessary to load many facts (propositions) that have low business value, then MongoDB should fit.
- **High Availability in an Unreliable Environment** – setting replicaSet (set of servers that act as Master-Slaves) is easy and fast. Moreover, recovery from a node (or a data center) failure is instant, safe, and automatic.
- **Scalability (and Data Sharding)** – databases scaling is hard (performance of operations based on a single MySQL table will degrade when crossing the 5-10GB threshold per table [32]). If it is necessary to partition and shard the database, MongoDB has a built-in easy solution for that.

- **Data is Location Based** – MongoDB has built in spatial functions. Thus, finding relevant data from specific locations should be fast and accurate.
- **Schema is Not Stable** – as MongoDB database is schema-less. It means that one does not have to explicitly define schema at the database level. Therefore, adding a new field is straightforward. It does not affect old rows (or documents) and will be instant.
- **Data Set is Going to be Big (starting from 1GB)**

When to use a SQL DBMS?

- **Data Consistency** – database normalization in SQL databases allows the DBMS to store a fact only once in a table and set references to it in other tables.
- **Data Linking** – if a table has a relationship with the same or another table, then it is possible to join these tables in order to retrieve all necessary information with one request.
- **Transactions** – if one works with critical data, then there must be guarantees from the DBMS that all the registered data is consistent and conforms to predefined rules. There must also be possibility to put multiple statements together to one transaction.
- **Separation of Concerns** – different people are dealing with programming and database management. Different database levels (external, logical, internal) allow developers to achieve different goals.

### 2.2.1 Alternative Opinions

There are different opinions as to whether one should use a NoSQL or a SQL DBMS.

The first is the idea of Polyglot Persistence introduced by Martin Fowler [13]. It means that each enterprise that has decent size could have a variety of different data storage technologies for different kinds of data.

The second opinion, very thoroughly described by Sarah Mei in her article "Why You Should Never Use MongoDB" [33], is that one should very carefully think as to whether to use documents-based NoSQL systems. The point is that NoSQL data storage is effective only when there are no relationships between documents. That is, all necessary data should be accessible through the one document. The problem arise then due to some reasons (e.g. requirements from a client), it is no longer possible to avoid relationships between different documents.

There are only two bad solutions to that problem: data duplication or documents linking.

Data duplication in this context means that every time when a document D should reference to the data in some other document the data is copied (duplicated) to D in the form of embedded documents. In this way, D will become a hierarchy of embedded documents. This is dangerous, because updating one document now means the need to walk through all the other documents where this updated document appears to change the data in all those different places. This is very error-prone and often leads to inconsistent data and mysterious errors, particularly when dealing with deletions.

Documents linking in NoSQL means that application should issue a separate request to DBMS for every linked document due to the lack of join feature. It will significantly degrade performance of the application and increase difficulty of development.

# 3 Research and Design

## 3.1 Goal

Currently (spring 2015), only the EnterpriseDB has officially conducted a performance comparison of PostgreSQL with JSON data types and MongoDB database. However, as was stated in section 2.1, this comparison has shortcomings. The utilized testing framework uses console applications provided by both DBMSs (*psql* and *mongo*), to accomplish benchmarked actions. The purpose of these programs is to provide direct access to DBMSs for performing administrative, maintenance, debugging, etc. actions. Applications do not use these.

The goal of the present work is to measure performance of the PostgreSQL with JSON data types and compare it to performance of the MongoDB database from the perspective of a real world Java application.

## 3.2 Description

Experiments are based on a real world Java web application named "FlowGrab" and developed at ByteLife Solutions [34]. The author of the present thesis fulfills a role of the lead developer in "FlowGrab's" development team. Application features and overall architecture are not important for the experiments' design, thus only the database layer (DAO and domain objects classes) is described.

Although, the goal is to compare PostgreSQL with JSON data types and MongoDB, there is one design with standard "SQL + JPA + Hibernate" approach. This is the currently working solution of the application.

Application business layer communicates with a database through a package of Data Access Object (DAO) classes and Model classes, which correspond to the application domain objects. There are 17 DAO classes to manage 28 model classes listed in Table 1.

**Table 1. Java application's DAO and Domain Object classes.**

| DAOs | Domain Objects/Models |
|---|---|
| BlacklistedNameDAO.java | BlacklistedName.java |
| CarouselContentDAO.java | CarouselContent.java |
| CategoryDAO.java | Category.java |
| CommitChangeLogDAO.java | CommitChangeLog.java |
| EnduserDAO.java | EnduserContact.java<br>Enduser.java<br>EnduserRole.java<br>EnduserSocialMedia.java |
| EnduserTeamDAO.java | EnduserTeam.java |

| DAOs | Domain Objects/Models |
|---|---|
| LicenceDAO.java | Licence.java |
| MergeResultDAO.java | MergeResult.java<br>PackageElement.java |
| NotificationDAO.java | Notification.java |
| PluginVersionDAO.java | PluginVersion.java |
| ProjectDAO.java | ProjectCategory.java<br>ProjectComment.java<br>Project.java<br>ProjectTag.java |
| ProjectSubscriberDAO.java | ProjectSubscriber.java |
| ProjectVersionDAO.java | File.java<br>ProjectVersion.java<br>ReleasedProjectVersion.java |
| ProjectVersionDownloadDAO.java | ProjectVersionDownload.java |
| RoleTypeDAO.java | RoleType.java |
| TagDAO.java | Tag.java |
| TeamDAO.java | TeamContact.java<br>Team.java<br>TeamSocialMedia.java |

DAO classes accept domain object instances for *create* and *update* operations and return domain object instances for *find* operations. It is noteworthy that some of the domain objects have One-to-Many associations with other objects. The author illustrates associations between domain objects on Figure 3.

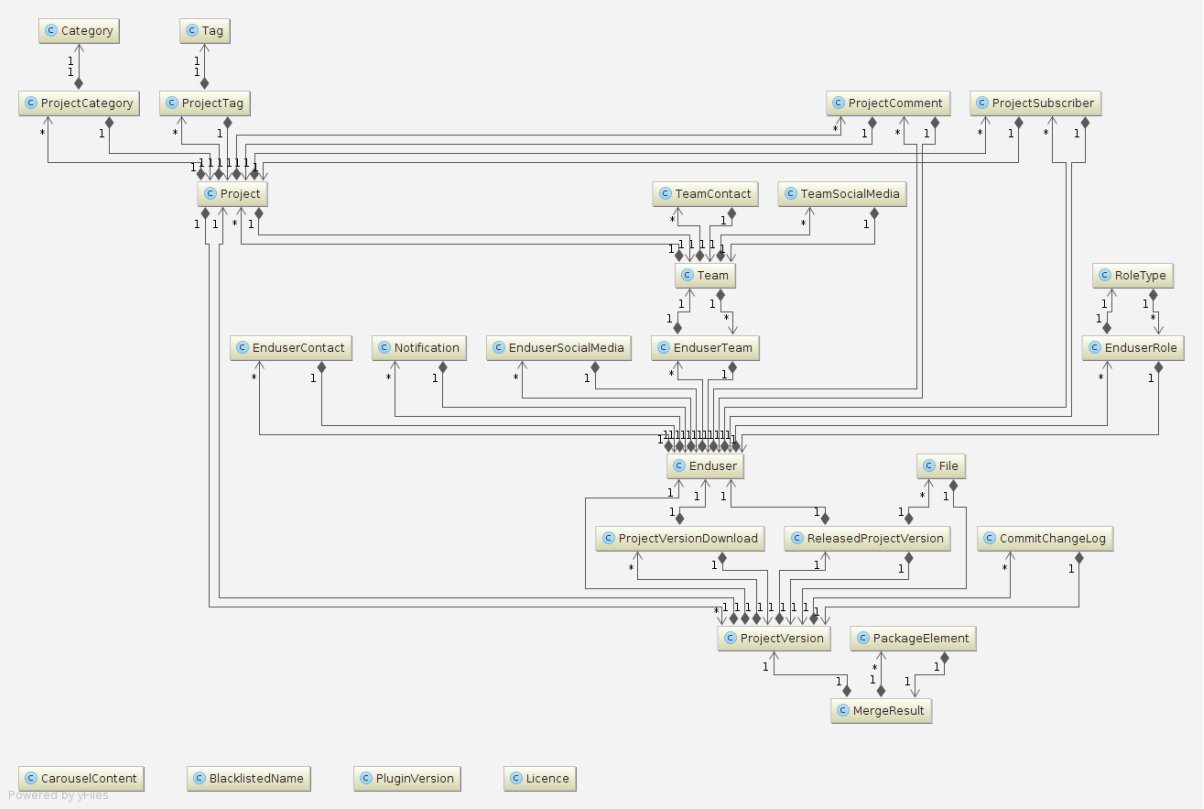


**Figure 3. Class diagram of application domain objects.**

It is easy to notice, that there are “lightweight” objects with no associations, as well as several “heavy” objects with multiple associations.

In this thesis, nine designs will be used to evaluate the performance of PostgreSQL and MongoDB. The designs differ in how the DAO classes are implemented (what technology is used to communicate with a database). Table 2 describes the DBMS and technologies used in case of each design.

**Table 2. List of Designs.**

| Design Identifier | DBMS | Technologies |
|---|---|---|
| Design 1 | PostgreSQL | JPA + Hibernate |
| Design 2 | PostgreSQL with JSON | JPA + Hibernate |
| Design 3 | PostgreSQL with JSONB | JPA + Hibernate |
| Design 4 | PostgreSQL with JSON | JDBC + JSON processing in Java |
| Design 4a | PostgreSQL with JSON | JDBC |
| Design 5 | PostgreSQL with JSONB | JDBC + JSON processing in Java |
| Design 5a | PostgreSQL with JSONB | JDBC |
| Design 6 | MongoDB | MongoDB Java Driver |
| Design 6a | MongoDB | MongoDB Java Driver |

### 3.2.1 Notes for designs with JPA and Hibernate

#### *3.2.1.1 Associations loading*

JPA specification defines two major strategies of loading associations (*LAZY* and *EAGER*). The *EAGER* strategy is a requirement that all associations' data must be eagerly fetched with the base object (generally it means that JPA provider will generate one SQL query with *joins* in order to get all required data within single request). The *LAZY* strategy means that associations' data should be fetched lazily when it is first time accessed (separate queries are issued on demand).

The *LAZY* loading is commonly used to defer the loading of the attributes or associations of an entity or entities until the point at which they are needed. On the other hand, the eager loading is an opposite concept in which the attributes and associations of an entity or entities are fetched explicitly and without any need for pointing them.

In "FlowGrab" Web application, most of the associations are fetched using *LAZY* initialization. However, for the sake of the experiment, this was changed to be *EAGER* for the most of them. The author did it in order to make all designs to fetch an equal amount of data in benchmarks.

#### *3.2.1.2 Cache*

Another important note that Hibernate extensively uses cache mechanisms. There are two levels of cache: the first-level cache and the second-level cache.

The **second-level cache** is the optional cache and it requires third-party cache providers in order to work. Although, it is used in the application, it was disabled for the experiments.

The **first-level cache** is the Session cache and is the mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database. When an entity is queried for the first time, it is retrieved from the database and stored in the first level cache associated with hibernate session. If the same object is queried again within the same session object, it will be loaded from the cache without issuing a query to the database.

If multiple updates to an object are issued, then Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. When the session is closed, all the objects being cached are lost and either persisted or updated in the database.

There is no documented possibility to disable the first-level cache in Hibernate, although it is allowed to manually delete all entries from the cache. In experiments with Hibernate, each benchmark is operating in context of one Session object. In order to ensure that query to a

database is issued after every operation (except for the new data insertions), Hibernate is ordered to send all pending queries to the database and cache is cleared.

### 3.2.2 Designs

Next, the author presents a description of the designs. Details and measurement results for each experiment based on these designs will be described in chapter 4. Firstly, some general remarks.

Designs 1–5 are for PostgreSQL. Design 6 is for MongoDB. Designs 2, 3, 4, 4a, 5, and 5a use tables with JSON or JSONB columns.

Tables in case of designs 2, 4, 4a have a form:

```
Table(PK, [FK_1, ... , FK_n], column_of_JSON_data_type).
```

Tables in case of designs 3, 5, 5a have a form:

```
Table(PK, [FK_1, ... , FK_n], column_of_JSONB_data_type).
```

where PK means the primary column and FK_n means a foreign key column. Only designs of MongoDB (6 and 6a) use embedded documents.

**Design 1** – Standard implementation meaning that the tables do not have JSON or JSONB columns.

- Each domain object is mapped to a table
- Each domain object's property is mapped to a table's column using JPA annotations.
- JPA's entity manager (EntityManager), using Hibernate, automatically generates SQL queries, manages associations, and processes domain objects' data.

**Design 2 and Design 3**

- Each domain object is mapped to a table.
- All primary and foreign keys are mapped to the corresponding fields of domain objects (to preserve JPA's automatic association management).
- Other properties are automatically serialized into JSON before sending request and deserialized after receiving response to/from the database.
- JPA's entity manager (EntityManager) automatically generates SQL queries, manages associations, and processes domain objects' data.
- Custom Hibernate data types were implemented in order to support PostgreSQL's JSON and JSONB types.

**Design 4 and Design 5**

- Each domain object corresponds to a table.
- All primary and foreign keys are mapped to the corresponding fields of domain objects
- Other properties are manually serialized into JSON before sending request and deserialized after receiving response to/from the database.
- JDBC is used for the communication with database.
- Developer writes SQL queries, manages associations (mostly through the SQL joins), and processes domain objects' data manually.
- JPA annotations are removed from the domain objects.

SELECT-query example:

```
"SELECT id, data FROM category"
```

, where *data* is of JSON or JSONB type.

**Design 4a and Design 5a** – Alternative to Designs 4 and 5. They differ by how the JSON data is handled. JSON data processing is removed from the application and given to the DBMS and thus, perhaps, making these designs more performant.

SELECT-query example:

```
"SELECT id, data->'category_name', data->'deleted', data->'created', data->'modified' FROM category"
```

, where *data* is of JSON or JSONB type.

**Design 6**

- Each domain object correspond to a MongoDB collection.
- Each domain object instance is one document in a collection. MongoDB Java Driver is used for communication with the database.
- Developer writes queries, manages associations, and processes domain objects' data.
- The main problem with this design is how to manage associations. As it was stated in section 2.2.1, there are two solutions: data duplication and/or documents linking. Data duplication is not an option for this application, due to the number of associations and the fact that associated objects may be changed relatively often. The latter will lead to additional operations that increases the workload of the system and may cause data inconsistencies. Therefore, the linking of documents was chosen.

- This design operated with MongoDB MMAPv1 storage engine.

**Design 6a** – differs from Design 6 by that the WiredTiger storage engine was used instead of MMAPv1.

## 3.3 Measurements

We made performance measurements by using the benchmarking tool developed by the Oracle OpenJDK and JRE team. This tool is the Java Microbenchmarking Harness (JMH) framework. JMH is a Java harness for building, running, and analyzing benchmarks written in Java [8]. Its distinctive advantage over other frameworks is that the same developers in Oracle who implement the JIT (Just-In-Time compiler) develop it. The reason is that compiler does assortment of code optimizations during the compilation process. Some of these optimizations should be considered by the author of benchmarks and by the benchmarking tool itself. The JMH is silently trying to bypass some of the optimizations. This allows to receive a more precise performance measurement results.

**How JMH works?**

The necessary minimum is to annotate a method, whose performance is to be measured, with @Benchmark annotation. There are also some guidelines to follow in the documentation. When run, JMH will disable or prevent compiler optimizations and will iteratively measure method's execution time. After benchmarking is done, JMH provides an average execution time with the value of measurement uncertainty (error) for every executed benchmark.

Example of output:

```
Benchmark                              Mode  Cnt   Score   Error  Units
JMHSample_08_DeadCode.baseline         avgt    5   0.361 ± 0.235  ns/op
JMHSample_08_DeadCode.measureRight     avgt    5  23.164 ± 0.346  ns/op
JMHSample_08_DeadCode.measureWrong     avgt    5   0.322 ± 0.070  ns/op
```

JMH has five modes how it will measure performance. These modes are listed in Table 3. Additionally, it allows setting different measurement time units, from nanoseconds to days.

**Table 3. JMH Benchmark Modes**

| Benchmark Mode | Description |
|---|---|
| **Throughput** | "Operations per unit of time. Runs by continuously calling Benchmark methods, counting the total throughput over all worker threads. The mode is time-based, and it will run until the iteration time expires." [35] |
| **AverageTime** | "Average time per operation. Runs by continuously calling Benchmark methods, counting the average time to call over all worker threads. This is the inverse of Throughput, but with different aggregation policy. The mode is time-based, and it will run until the iteration time expires." [35] |
| **SampleTime** | "Samples the time for each operation. Runs by continuously calling Benchmark methods, and randomly samples the time needed for the call. This mode automatically adjusts the sampling frequency, but may omit some pauses, which missed the sampling measurement. The mode is time-based, and it will run until the iteration time expires." [35] |
| **SingleShotTime** | "Measures the time for a single operation. Runs by calling Benchmark once and measuring its time. The mode is work-based, and will run only for a single invocation of Benchmark method." [35] |
| **All** | "Meta-mode: all the benchmark modes." [35] |

The simplest Benchmark listing is on Figure 4.

```
1 @State(Benchmark)
2 @BenchmarkMode(Mode.All)
3 @OutputTimeUnit(SECONDS)
4 public class CategoryDAOBenchmark {
5
6   private final CategoryDAO categoryDAO = new CategoryDAO();
7
8   @Benchmark
9   public List<Category> findAll() {
10    return categoryDAO.findAll();
11  }
12
13 }
```

**Figure 4. The simplest JMH benchmark code listing.**

JMH allows configuring a number of warmup iterations and a number of measurement iterations. The purpose of warmup iterations is to minimize the influence of a "cold start", when system allocates resources, prepares connections, and so on. Warmup iterations are not taken into account when measurement results are processed.

The full raw output of the running *CategoryDAOBenchmark* is listed on Figure 4 with Benchmark mode set to *Throughput*, five warmup iterations and ten measurement operations:

```
# JMH 1.6.1
# VM invoker: /usr/lib/jvm/java-8-oracle/jre/bin/java
# Warmup: 5 iterations, 1 s each
# Measurement: 10 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: com.bytelife.flowhub.dao.CategoryDAOBenchmark.findAll

# Run progress: 0.00% complete, ETA 00:00:15
# Fork: 1 of 1
# Warmup Iteration   1: 1146.256 ops/s
# Warmup Iteration   2: 2475.241 ops/s
# Warmup Iteration   3: 3170.098 ops/s
# Warmup Iteration   4: 3680.288 ops/s
# Warmup Iteration   5: 2449.903 ops/s
Iteration   1: 2411.434 ops/s
Iteration   2: 2353.496 ops/s
Iteration   3: 2413.627 ops/s
Iteration   4: 2517.253 ops/s
Iteration   5: 2458.682 ops/s
Iteration   6: 2487.218 ops/s
Iteration   7: 3151.686 ops/s
Iteration   8: 2449.350 ops/s
Iteration   9: 2496.540 ops/s
Iteration  10: 2471.994 ops/s

Result: 2521.128 ±(99.9%) 342.735 ops/s [Average]
 Statistics: (min, avg, max) = (2353.496, 2521.128, 3151.686), stdev =
226.698
 Confidence interval (99.9%): [2178.393, 2863.863]

# Run complete. Total time: 00:00:17

Benchmark                       Mode  Cnt     Score     Error  Units
CategoryDAOBenchmark.findAll  thrpt   10  2521.128 ± 342.735  ops/s
```

For the actual DAO classes benchmarking 10 warmup and 20 measurement operations on a single JVM were used.

### 3.4 Test Data

Data used for benchmarking is the same as it is for unit testing, which is from ten to twenty rows in every table. This amount of data is the average amount that is fetched from the database with one query in an average web application, according to the experience of author.

# 4 Experiments

Benchmarks, used to measure performance of designs, are written by the author of the thesis utilizing the JMH framework. The unit tests of the application's DAO classes were taken as a basis for the benchmarks.

In total nine designs were proposed (see section 3.2.2). Therefore, there are nine experiments implemented: one experiment per design. Correlation between designs and experiments is listed in Table 4.

**Table 4. Correlation between designs and experiments.**

| Design | Experiment | DBMS | Technologies |
|---|---|---|---|
| Design 1 | Experiment 1 | PostgreSQL | JPA + Hibernate |
| Design 2 | Experiment 2 | PostgreSQL with JSON | JPA + Hibernate |
| Design 3 | Experiment 3 | PostgreSQL with JSONB | JPA + Hibernate |
| Design 4 | Experiment 4 | PostgreSQL with JSON | JDBC + JSON processing in Java |
| Design 4a | Experiment 4a | PostgreSQL with JSON | JDBC |
| Design 5 | Experiment 5 | PostgreSQL with JSONB | JDBC + JSON processing in Java |
| Design 5a | Experiment 5a | PostgreSQL with JSONB | JDBC |
| Design 6 | Experiment 6 | MongoDB | MongoDB Java Driver |
| Design 6a | Experiment 6a | MongoDB | MongoDB Java Driver |

Experiment consists of the design-specific DAO classes' implementation and 92 unique benchmarks. There are 66 benchmarks for SELECT operations, which include

- fetching of the "lightweight" domain objects' data with no associations,
- fetching of the domain objects' data with varying number of associations,
- fetching of the domain objects' data while filtering it (using WHERE clause in SQL statement or its counterpart for MongoDB),
- fetching of the domain objects' data while grouping and counting occurrences (using COUNT(*) and GROUP BY in SQL or theirs counterpart for MongoDB).

There are 16 benchmarks for INSERT operations, which include

- insertion of the "lightweight" domain objects' data with no associations,
- insertion of the domain objects' data with varying number of associations.

There are 10 benchmarks for UPDATE operations.

Totally, there are 92 benchmarks for every JMH's benchmark mode and there are four modes. All the operations in case of these benchmarks are non-conflicting meaning that the system does not have to block execution of certain statements or roll them back to ensure consistency.

The code of benchmarks are mostly the same for all experiments. The major difference is that a cache cleaning operation was added to the majority of benchmarks that use Hibernate. The essential work was done for the implementation of DAO classes according to the different designs. Appendix 1 contains a table that shows the number of physical lines of code in case of different DAO classes.

The benchmarks are reusable for any versions of PostgreSQL and MongoDB, because they depend only on DAO classes interface. It is only necessary to change DAO implementations, if other versions of PostgreSQL or MongoDB have different API.

Time unit for the measurements is milliseconds (ms).

JMH was configured to perform ten warmup and twenty measurement operations on a single JVM.

PostgreSQL and MongoDB DBMSs were used "as is". No additional configuration were made after their installation. Moreover, there were no additional indexes created neither for PostgreSQL nor for MongoDB. Both DBMSs automatically create indexes for the row/document primary key/ID. The creation of additional indexes is considered as "premature optimization" by the author. In Donald Knuth's paper "Structured Programming with go to Statements" [36], he wrote: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." That is, in opinion of the author, additional indexes should be created only if there are performance issues, and the average application should be able to work well without additional indexes.

This chapter presents aggregated results of benchmarking divided into four categories per measurement mode for each experiment.

## 4.1 Experiment 1

DAO classes measured in this experiment are real application classes with the difference that the most associations' initialization is changed from *LAZY* to *EAGER*, and cache was disabled.

JPA's EntityManager is used to cooperate with the database. JPA provider is Hibernate ORM.

Listing of the simple DAO method is on Figure 5:

```
1  public Category findById(UUID id) {
2    return em.find(Category.class, id);
3  }
```

**Figure 5. Example of CategoryDAO.findById() method in Experiment 1 (Experiment 2, and 3).**

Table 5 and the chart on Figure 6 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the first experiment. When benchmarking is done, JMH provides an average execution time with the value of measurement uncertainty (error) for every executed benchmark ("±" in this and the following tables).

**Table 5. Experiment 1 measurements.**

| | **SELECTS** | **INSERTS** | **UPDATES** |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.613 ± 0.057 | 0.056 ± 0.005 | 0.113 ± 0.006 |
| **AverageTime** (ms/op) | 9.805 ± 0.231 | 20.188 ± 1.877 | 9.668 ± 0.583 |
| **SampleTime** (ms/op) | 9.742 ± 0.134 | 19.911 ± 0.454 | 9.642 ± 0.229 |
| **SingleShotTime** (ms/op) | 14.320 ± 2.396 | 16.344 ± 3.116 | 15.195 ± 3.038 |

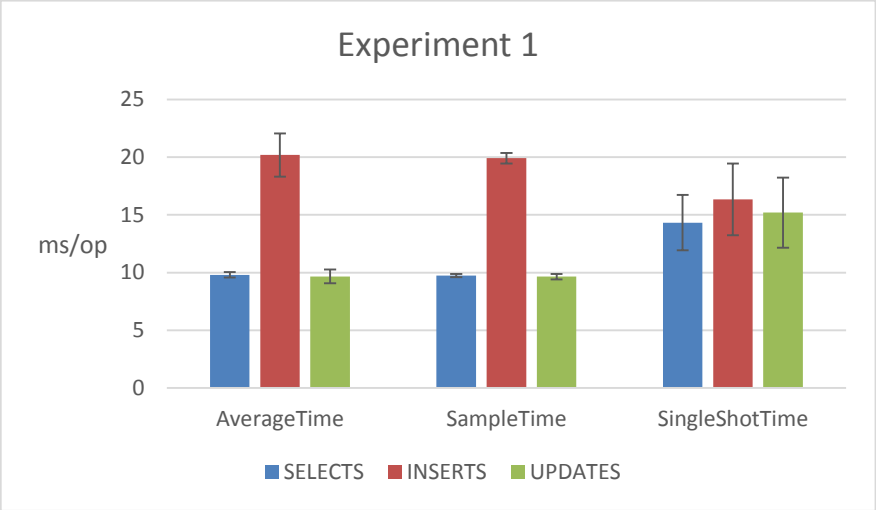


**Figure 6. Experiment 1 results chart.**

## 4.2 Experiment 2

Experiment 2 uses the JPA and the Hibernate to map a String value, which contains serialized JSON object with the values of entity's properties, to a PostgreSQL table's column of JSON data type. Entity preserves all getters and setters for its properties, but values are actually stored in JSON format into the inner property named 'data'.

The Hibernate does not have built-in support for PostgreSQL data type JSON. However, it offers a simple and flexible way for necessary extensions. It was necessary to define a new data type for Hibernate in order to use the new PostgreSQL's JSON data type. This is done by implementing the Hibernate's interface *UserType* as demonstrated on Figure 7…

```
1 public class StringJsonUserType implements UserType {
2
3   private static final int TYPE;
4
5   static {
6     String fileContents = getContentsOfPersistenceXml();
7     if (fileContents.contains("com.bytelife.flowhub.util.hibernate.JsonPostgreSQLDialect")) {
8       TYPE = Types.JAVA_OBJECT;
9     } else {
10      throw new UnsupportedOperationException("Unsupported database dialect!");
11    }
12  }
13
14  @Override
15  public int[] sqlTypes() {
16    return new int[]{TYPE};
17  }
18  // other methods are omitted ...
19 }
```

**Figure 7. Listing of Hibernate's custom UserType for PostgreSQL's JSON data type support.**

…and registering this implementation with the Hibernate's runtime through a custom Dialect, which extends the default PostgreSQL dialect (see Figure 8).

```
1 public class JsonPostgreSQLDialect extends PostgreSQL9Dialect {
2
3   public JsonPostgreSQLDialect() {
4     super();
5     this.registerColumnType(Types.JAVA_OBJECT, "json");
6   }
7
8 }
```

**Figure 8. Extended Hibernate's PosgtreSQL dialect to support PostgreSQL's JSON data type.**

The last thing is to annotate entity's property 'data', that it should be mapped to the *StringJsonUserType* as demonstrated on Figure 9.

```
1 @Type(type = "com.bytelife.flowhub.util.hibernate.StringJsonUserType")
2 protected String data;
```

**Figure 9. Listing of Json User Type mapping to an entity's property.**

With these steps accomplished, Hibernate will automatically map values from the PostgreSQL's columns of type JSON to the annotated with `@Type(type = "*.StringJsonUserType")` entity's properties of type String.

DAO classes are unchanged.

Table 6 and chart on Figure 10 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the second experiment.

**Table 6. Experiment 2 measurements.**

| | **SELECTS** | **INSERTS** | **UPDATES** |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.566 ± 0.056 | 0.059 ± 0.005 | 0.095 ± 0.006 |
| **AverageTime** (ms/op) | 7.459 ± 0.236 | 18.859 ± 1.533 | 11.145 ± 0.626 |
| **SampleTime** (ms/op) | 7.340 ± 0.119 | 18.618 ± 0.416 | 11.042 ± 0.250 |
| **SingleShotTime** (ms/op) | 11.135 ± 1.877 | 18.313 ± 3.462 | 16.337 ± 3.391 |

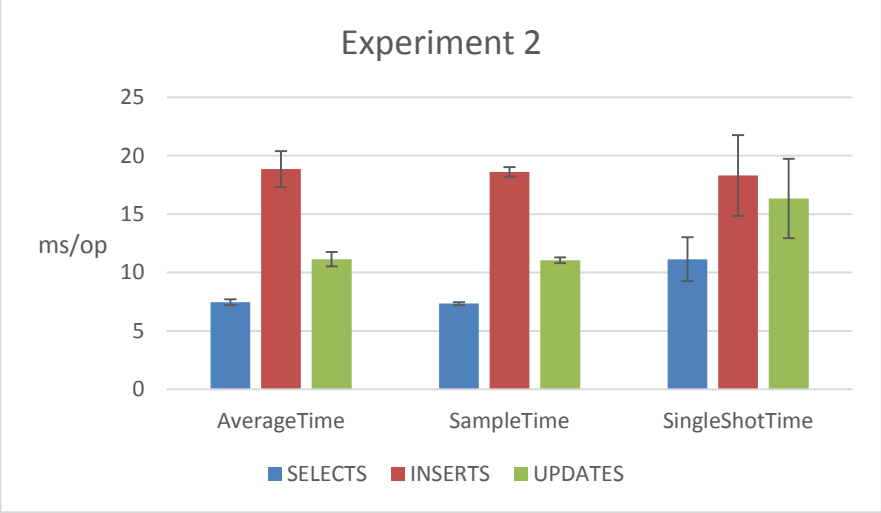


**Figure 10. Experiment 2 results chart.**

## 4.3 Experiment 3

Experiment 3, similarly to the previous experiment, uses the Hibernate to map a String value with a serialized JSON object to a PostgreSQL table's column of JSONB data type. Entity preserves all getters and setters for its properties, but values are stored in JSON format into the inner property called 'data'.

The Hibernate does not have built-in support for PostgreSQL data type JSONB. In order to use the new JSONB data type it was necessary to define a new data type for the Hibernate. This is done exactly the same way like in the second experiment, with a difference in the custom PostgreSQL dialect as demonstrated on Figure 11.

```
1 public class JsonPostgreSQLDialect extends PostgreSQL9Dialect {
2
3   public JsonPostgreSQLDialect() {
4     super();
5     this.registerColumnType(Types.JAVA_OBJECT, "jsonb");
6   }
7
8 }
```

**Figure 11. Extended Hibernate's PosgtreSQL dialect to support PostgreSQL's JSONB data type.**

The Hibernate will automatically map values from the PostgreSQL's columns of type JSONB to the annotated entity's properties of type String.

DAO classes are unchanged.

Table 7 and chart on Figure 12 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the third experiment.

**Table 7. Experiment 3 measurements.**

| | **SELECTS** | **INSERTS** | **UPDATES** |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.588 ± 0.053 | 0.064 ± 0.005 | 0.101 ± 0.007 |
| **AverageTime** (ms/op) | 7.915 ± 0.267 | 17.605 ± 1.398 | 10.613 ± 0.688 |
| **SampleTime** (ms/op) | 7.744 ± 0.122 | 17.482 ± 0.395 | 10.542 ± 0.246 |
| **SingleShotTime** (ms/op) | 11.721 ± 2.086 | 16.107 ± 3.210 | 14.979 ± 3.485 |

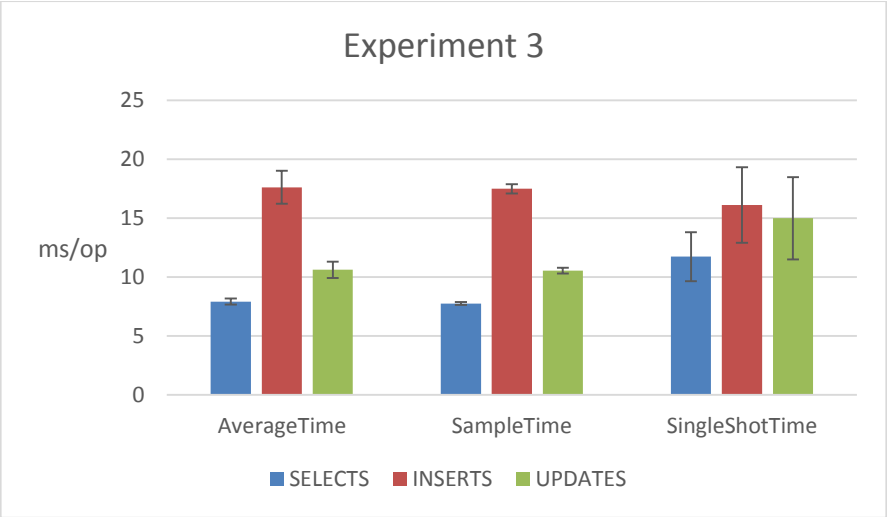


**Figure 12. Experiment 3 results chart.**

## 4.4 Experiment 4

Experiment 4 completely changes the implementation of DAO classes. Support of the JPA and the Hibernate is removed from the project, along with the JPA and the Hibernate specific annotations from the domain object classes, EntityManager, and persistence configuration files.

Each domain object corresponds to a table, all primary and foreign keys correspond to columns, and other properties are manually serialized to JSON before data writing and deserialized after data reading to/from the database. JDBC is used for the communication with database. Developer writes SQL queries, manages (mostly through the SQL joins) associations, and processes domain objects' data.

SQL select query example:

```
SELECT id, data FROM category WHERE id = cast(? as uuid);
```

Figure 13 demonstrates an example of the simplest DAO class method.

```java
public Category findById(UUID id) {
  try (PreparedStatement statement =
         conn.prepareStatement("SELECT id, data FROM category WHERE id = cast(? as uuid)")) {
    statement.setString(1, id.toString());
    try (ResultSet rs = statement.executeQuery()) {
      while (rs.next()) {
        String _id = rs.getString("id");
        String data = rs.getString("data");

        Category category = new Category();
        category.setId(UUID.fromString(_id));
        category.setData(data);
        category.convertToEntityAttribute();

        return category;
      }
    }
  }
  return null;
}
```

**Figure 13. Example of CategoryDAO.findById() method in Experiment 4 (and Experiment 5).**

Table 8 and chart on Figure 14 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the fourth experiment.

**Table 8. Experiment 4 measurements.**

| | **SELECTS** | **INSERTS** | **UPDATES** |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.193 ± 0.032 | 0.102 ± 0.004 | 0.114 ± 0.006 |
| **AverageTime** (ms/op) | 23.116 ± 1.108 | 10.702 ± 0.357 | 9.169 ± 0.380 |
| **SampleTime** (ms/op) | 22.867 ± 0.502 | 10.531 ± 0.233 | 9.079 ± 0.211 |
| **SingleShotTime** (ms/op) | 30.856 ± 6.215 | 13.033 ± 3.211 | 11.545 ± 2.883 |

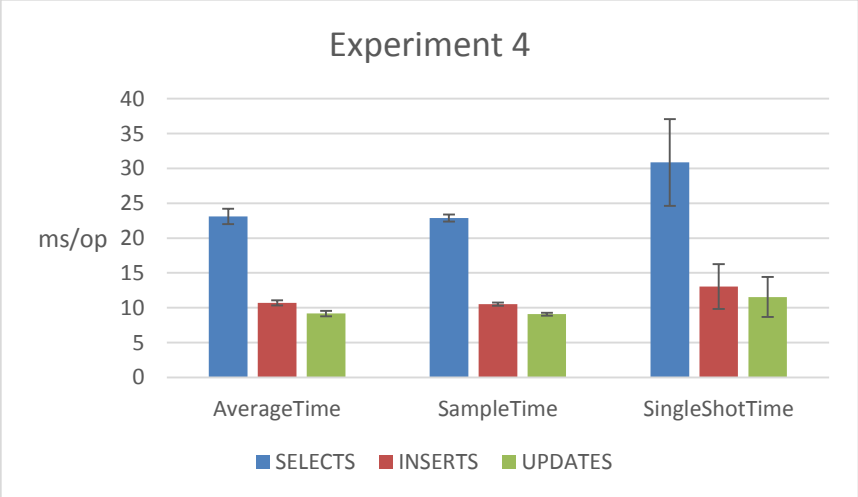


**Figure 14. Experiment 4 results chart.**

There is one major performance issue with this experiment. For the JSON string that is received from the database, the Java API for JSON Processing is used. It seems that the API has poor performance. It drastically degrades the results of performance measurements.

Solution to this problem is introduced and implemented in the alternative experiment, Experiment 4a.

## 4.5 Experiment 4a

Experiment 4a is an alternative implementation of the fourth experiment. Implementation of DAO objects in this experiment uses PostgreSQL system-defined operator '->' in SQL queries to get JSON object field by key. As a result, non-performant JSON processing is removed from the Java code, improving overall performance of the benchmarks.

SQL select query example:

```
SELECT
  id,
  data->'category_name' as category_name,
  data->'deleted' as deleted,
  data->'created' as created,
  data->'modified' as modified
FROM category WHERE id = cast(? as uuid);
```

Figure 15 demonstrates an example of the simplest DAO class method.

```
1  public Category findById(UUID id) {
2      try (PreparedStatement statement = conn.prepareStatement(SELECT_FROM_CATEGORY_BY_ID)) {
3        statement.setString(1, id.toString());
4        try (ResultSet rs = statement.executeQuery()) {
5          while (rs.next()) {
6            Category category = new Category();
7            category.setId(UUID.fromString(rs.getString("id")));
8            category.setCategoryName(rs.getString("category_name"));
9            category.setDeleted(rs.getBoolean("deleted"));
10           category.setCreated(rs.getDate("created"));
11           category.setModified(rs.getDate("modified"));
12
13           return category;
14         }
15       }
16     }
17     return null;
18   }
```

**Figure 15. Example of CategoryDAO.findById() method in Experiment 4a (and Experiment 5a).**

Table 9 and chart on Figure 16 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the Experiment 4a.

**Table 9. Experiment 4a measurements.**

| | SELECTS | INSERTS | UPDATES |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.445 ± 0.038 | 0.132 ± 0.006 | 0.151 ± 0.009 |
| **AverageTime** (ms/op) | 3.584 ± 0.123 | 7.869 ± 0.347 | 6.681 ± 0.373 |
| **SampleTime** (ms/op) | 5.291 ± 0.078 | 7.727 ± 0.195 | 6.643 ± 0.179 |
| **SingleShotTime** (ms/op) | 7.532 ± 1.391 | 8.728 ± 2.867 | 7.427 ± 2.333 |

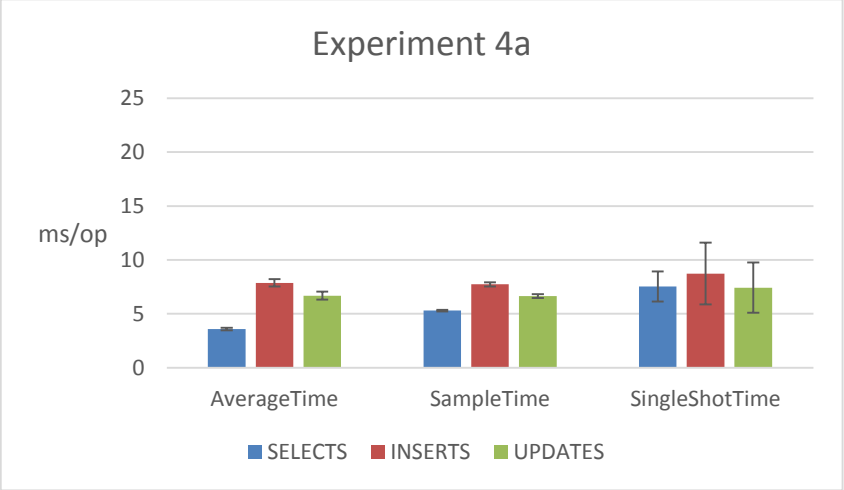


**Figure 16. Experiment 4a results chart.**

## 4.6 Experiment 5

Experiment 5 is almost the same as Experiment 4. It completely changes the implementation of DAO classes. Support of the JPA and the Hibernate is removed from the project, along with the JPA and the Hibernate specific annotations from the domain object classes, EntityManager, and persistence configuration files.

The difference is that it uses PostgreSQL's JSONB data type instead of JSON.

Table 10 and chart on Figure 17 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the fifth experiment.

**Table 10. Experiment 5 measurements.**

| | SELECTS | INSERTS | UPDATES |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.214 ± 0.036 | 0.102 ± 0.004 | 0.115 ± 0.005 |
| **AverageTime** (ms/op) | 24.000 ± 1.644 | 10.884 ± 0.402 | 9.181 ± 0.406 |
| **SampleTime** (ms/op) | 13.198 ± 0.220 | 10.499 ± 0.239 | 9.111 ± 0.211 |
| **SingleShotTime** (ms/op) | 31.252 ± 5.895 | 12.853 ± 2.936 | 10.841 ± 1.842 |

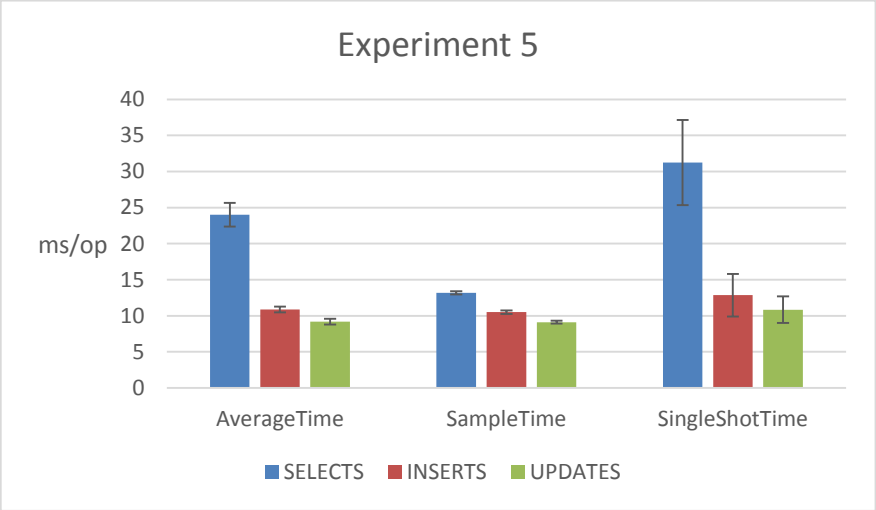


**Figure 17. Experiment 5 results chart.**

There is, same as in Experiment 4, one major performance issue with this experiment. For the parsing of JSON string, which is received from the database, the Java API for JSON Processing is used. It seems that API has a poor performance, drastically degrading the results of performance measurements.

Solution to this problem is introduced and implemented in the alternative experiment, Experiment 5a.

## 4.7 Experiment 5a

Experiment 5a is an alternative implementation of the fifth experiment. Implementation of DAO classes in this experiment uses PostgreSQL system-defined operator '->' in SQL queries to get JSONB object field by key. As a result, non-performant JSON processing is removed from the Java code, improving overall performance of the benchmarks.

Table 11 and chart on Figure 18 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the Experiment 5a.

**Table 11. Experiment 5a measurements.**

| | SELECTS | INSERTS | UPDATES |
|---|---|---|---|
| **Throughput** (ops/ms) | 1.502 ± 0.041 | 0.132 ± 0.006 | 0.154 ± 0.010 |
| **AverageTime** (ms/op) | 4.323 ± 0.148 | 7.843 ± 0.314 | 6.709 ± 0.369 |
| **SampleTime** (ms/op) | 4.228 ± 0.068 | 7.918 ± 0.197 | 6.699 ± 0.179 |
| **SingleShotTime** (ms/op) | 4.735 ± 1.080 | 9.034 ± 2.687 | 6.741 ± 2.559 |

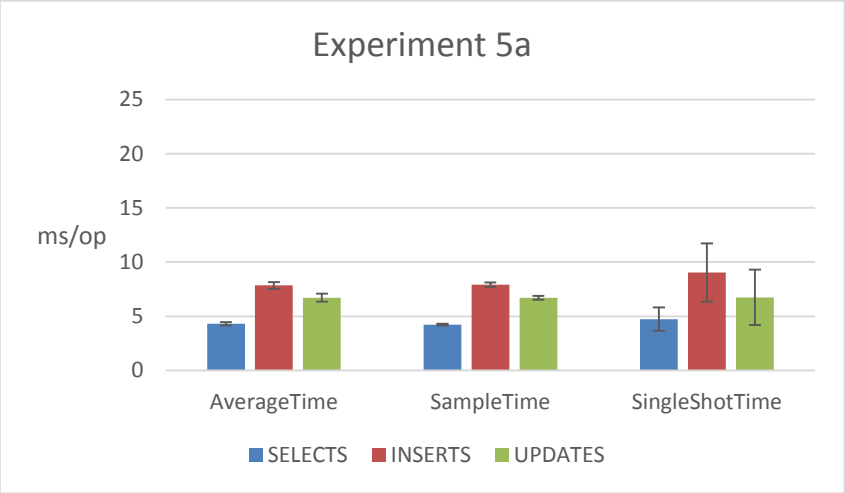


**Figure 18. Experiment 5a results chart.**

## 4.8 Experiment 6

Experiment 6 uses a MongoDB system. Each domain object correspond to a MongoDB collection. Each model instance is one document in the collection.

This experiment operated with MongoDB MMAPv1 storage engine.

Figure 19 demonstrates the simplest DAO class method.

```java
public Category findById(UUID id) {
  Document jsonEntity = collection.find(new Document("_id", id.toString())).first();

  Category category = new Category();
  category.setId(UUID.fromString(jsonEntity.getString("_id")));
  category.setCategoryName(jsonEntity.getString("category_name"));
  category.setDeleted(jsonEntity.getBoolean("deleted", false));
  category.setCreated(jsonEntity.getDate("created"));
  category.setModified(jsonEntity.getDate("modified"));

  return category;
}
```

**Figure 19. Example of CategoryDAO.findById() method in Experiment 6 (and Experiment 6a).**

Table 12 and chart on Figure 20 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the sixth experiment.

**Table 12. Experiment 6 measurements.**

| | **SELECTS** | **INSERTS** | **UPDATES** |
|---|---|---|---|
| **Throughput** (ops/ms) | 4.252 ± 0.165 | 2.869 ± 0.102 | 3.090 ± 0.141 |
| **AverageTime** (ms/op) | 3.540 ± 0.283 | 0.432 ± 0.020 | 0.446 ± 0.012 |
| **SampleTime** (ms/op) | 3.390 ± 0.084 | 0.423 ± 0.010 | 0.436 ± 0.003 |
| **SingleShotTime** (ms/op) | 6.769 ± 1.596 | 2.807 ± 0.836 | 2.748 ± 0.486 |

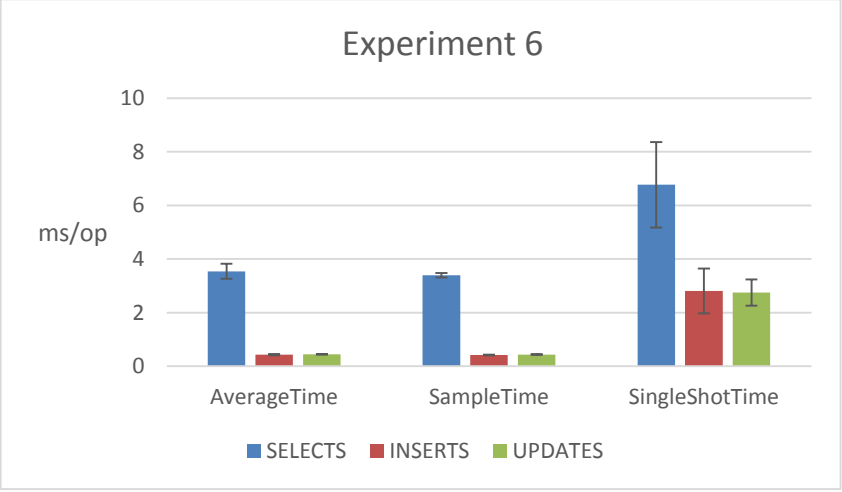


**Figure 20. Experiment 6 results chart.**

## 4.9 Experiment 6a

The Experiment 6a is the same as the sixth experiment with the difference that MongoDB system is using the WiredTiger storage engine instead of the MMAPv1.

Table 13 and chart on Figure 21 contain the average results for SELECTS, INSERTS, and UPDATES divided into four groups according to the measurement mode for the Experiment 6a.

**Table 13. Experiment 6a measurements.**

| | **SELECTS** | **INSERTS** | **UPDATES** |
|---|---|---|---|
| **Throughput** (ops/ms) | 5.058 ± 0.218 | 2.961 ± 0.156 | 2.981 ± 0.127 |
| **AverageTime** (ms/op) | 3.311 ± 0.308 | 0.426 ± 0.020 | 0.454 ± 0.016 |
| **SampleTime** (ms/op) | 3.197 ± 0.076 | 0.219 ± 0.010 | 0.224 ± 0.007 |
| **SingleShotTime** (ms/op) | 5.977 ± 1.656 | 2.932 ± 1.037 | 3.159 ± 1.318 |

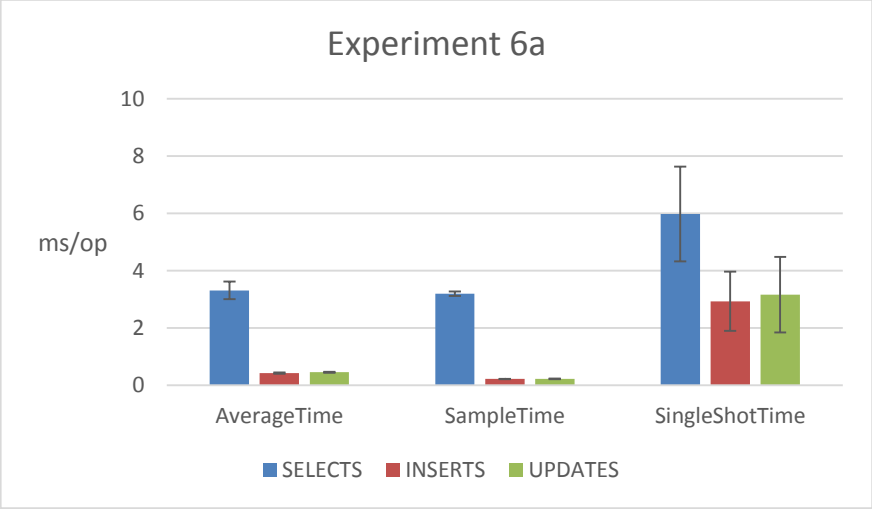


**Figure 21. Experiment 6a results chart.**

# 5 Analysis of the Results

This chapter presents summary tables of the experiments results and the analysis of these results for every type of operations. Each table contains measurement scores and their measurement uncertainty (errors) received using different JMH's measurement modes for each experiment. There are two charts based on each table: the first one illustrates the results of measurements with Throughput mode, and the second one with three other modes (AverageTime, SampleTime, and SingleShotTime).

The experiment with the best result is denoted with the green color in every summary table.

Chapter is divided into 3 sections for every operation type: SELECTs, INSERTs, and UPDATEs.

**Important note 1.** Although the AverageTime is a reverse of Throughput for a one separately taken benchmark, it is not true for the aggregated average values of all benchmarks. Explanation as seen by author follows.

Let us assume that there are two benchmarks for one experiment with the following results:

| | Throughput (ops/ms) | AverageTime (ms/op) |
|---|---|---|
| Benchmark 1 | $a$ | $\frac{1}{a}$ |
| Benchmark 2 | $b$ | $\frac{1}{b}$ |

The average Throughput for the experiment is $\frac{a+b}{2}$ (ops/ms).

The average AverageTime for the experiment is $\frac{\frac{1}{a}+\frac{1}{b}}{2} = \frac{a+b}{2ab}$ (ms/op).

It is obvious that the value of $\frac{a+b}{2ab}$ is not the reverse value of $\frac{a+b}{2}$.

**Important note 2.** There are benchmarks that measure SELECT operations for the entities with no associations. These are the fastest benchmarks. There are also benchmarks that measure SELECT operations for the entities with varying number of associations. These may be significantly slower. For instance, benchmark *ProjectDAOBenchmark.findById* is fetching one *Project* domain object and all of its associations (there are five One-to-Many relationships) with Average Time of 52.587 ± 0.918 ms. In contrast, benchmark

*CategoryDAOBenchmark.findById* is fetching one *Category* domain object without any associations with Average Time of 0.366 ± 0.009 ms.

SQL DBMSs allow using joins to fetch data of the base object with all its associations with one request. With NoSQL system, it is necessary to issue separate query for every association. It is important to note that the usage of joins does not mean that operation will be faster. Nevertheless, which approach is faster is going to be observed.

**Important note 3.** Hibernate has a built-in cache mechanism (first-level cache), that cannot be disabled. In order to put other solutions in equal conditions, this cache was cleared after every operation, using EntityManager's *clear()* method.

## 5.1 Summary of SELECT Operations Measurement Results

SELECT operations is the data reading from the database. Due to the nature of Web applications, it has the most important influence on application's overall performance. The reason is that in common Web application data reads happen more frequently than data writes. Multiple data read queries are issued to the application's data layer with every request. Although usually application heavily facilitates different caching mechanisms, it is still very important that reads from the underlying DBMS were as fast as possible.

**Note.** Results from experiments 4 and 5 may be omitted due to unnaturally slow operations execution time. The reason is that the Java API for JSON Processing is used for the JSON strings processing. It seems that the API has poor performance, drastically degrading the experiment's performance measurements.

Hereafter, two charts illustrating the experiments results and the summary table are presented.

The first chart on Figure 22 demonstrates the Throughput values. The greater the value the better.

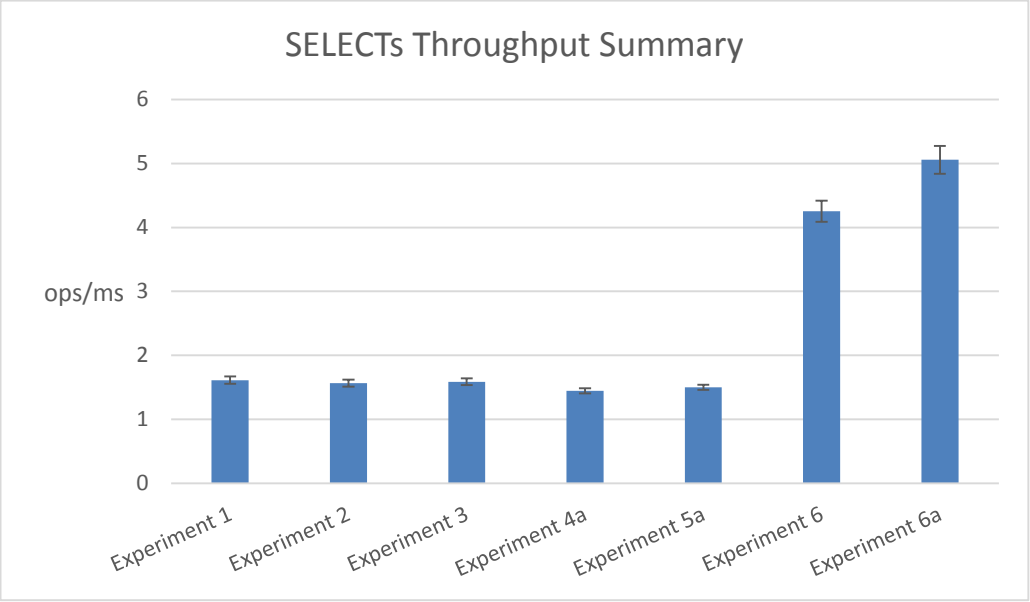


**Figure 22. SELECT operations throughput summary chart.**

The second chart on Figure 23 demonstrates values of AverageTime, SampleTime, and SingleShotTime. The smaller the value the better.


**Figure 23. SELECT operations performance summary chart.**

Table 14 presents results for all experiments. The best and the worst values within every measurement mode are marked with stronger and dashed outline respectively. The most and the least performant experiment is marked with green and red fill.

**Table 14. Experiments summary for SELECT operations.**

| | **Throughput** (ops/ms) | **AverageTime** (ms/op) | **SampleTime** (ms/op) | **SingleShotTime** (ms/op) |
|---|---|---|---|---|
| **Experiment 1** | 1.613 ± 0.057 | 9.805 ± 0.231 | 9.742 ± 0.134 | 14.320 ± 2.396 |
| **Experiment 2** | 1.566 ± 0.056 | 7.459 ± 0.236 | 7.340 ± 0.119 | 11.135 ± 1.877 |
| **Experiment 3** | 1.588 ± 0.053 | 7.915 ± 0.267 | 7.744 ± 0.122 | 11.721 ± 2.086 |
| ***Experiment 4*** | *1.193 ± 0.032* | *23.116 ± 1.108* | *22.867 ± 0.502* | *30.856 ± 6.215* |
| **Experiment 4a** | 1.445 ± 0.038 | 3.584 ± 0.123 | 5.291 ± 0.078 | 7.532 ± 1.391 |
| ***Experiment 5*** | *1.214 ± 0.036* | *24.000 ± 1.644* | *13.198 ± 0.220* | *31.252 ± 5.895* |
| **Experiment 5a** | 1.502 ± 0.041 | 4.323 ± 0.148 | 4.228 ± 0.068 | 4.735 ± 1.080 |
| **Experiment 6** | 4.252 ± 0.165 | 3.540 ± 0.283 | 3.390 ± 0.084 | 6.769 ± 1.596 |
| **Experiment 6a** | 5.058 ± 0.218 | 3.311 ± 0.308 | 3.197 ± 0.076 | 5.977 ± 1.656 |

Table 15 demonstrates all experiments' relative performance interrelation for SELECT operations.

**How to read this table**

Let us assume that one wants to lookup how the second experiment (Exp2) correlates to the first experiment (Exp1). One is looking for Exp2 tagged row on X-axis and for Exp1 tagged column on Y-axis. Afterwards, one is searching for the intersection of found row and column. The intersection shows a value of 76% with green color. It means that the second experiment (Exp2) is $\frac{100\%}{76\%} = \mathbf{1.3}$ **times more** performant than the first experiment (Exp1).

Now the inverse lookup: how the first experiment (Exp1) correlates to the second experiment (Exp2). One is looking for Exp1 tagged row on X-axis and for Exp2 tagged column on Y-axis. Afterwards, one is searching for the intersection of found row and column. The intersection shows a value of 131% with red color. It means that the first experiment (Exp1) is $\frac{131\%}{100\%} = \mathbf{1.3}$ **times less** performant than the second experiment (Exp2).

**Table 15. Experiments relative comparison for SELECT operations.**

| x \ y | Exp1 | Exp2 | Exp3 | Exp4 | Exp4a | Exp5 | Exp5a | Exp6 | Exp6a |
|---|---|---|---|---|---|---|---|---|---|
| Exp1 | 100% | 131% | 124% | 42% | 274% | 41% | 227% | 277% | 296% |
| Exp2 | 76% | 100% | 94% | 32% | 208% | 31% | 173% | 211% | 225% |
| Exp3 | 81% | 106% | 100% | 34% | 221% | 33% | 183% | 224% | 239% |
| Exp4 | 236% | 310% | 292% | 100% | 645% | 96% | 535% | 653% | 698% |
| Exp4a | 37% | 48% | 45% | 16% | 100% | 15% | 83% | 101% | 108% |
| Exp5 | 245% | 322% | 303% | 104% | 670% | 100% | 555% | 678% | 725% |
| Exp5a | 44% | 58% | 55% | 19% | 121% | 18% | 100% | 122% | 131% |
| Exp6 | 36% | 47% | 45% | 15% | 99% | 15% | 82% | 100% | 107% |
| Exp6a | 34% | 44% | 42% | 14% | 92% | 14% | 77% | 94% | 100% |

The most performant experiment with PostgreSQL is Experiment 4a (JDBC + PostgreSQL JSON data type). It is 1.2 times more performant than the analogues experiment with JSONB data type (Experiment 5a). However, MongoDB with MMAPv1 storage engine (Experiment 6) is just a little faster, 1.01 times.

MongoDB with WiredTiger storage engine showed slightly better results, than with MMAPv1 engine. The former is 1.07 times faster than the latter.

Summarizing, the most performant experiment with **MongoDB** showed **1.08-1.09 times better** performance **for SELECT operations than** the most performant experiment with

**PostgreSQL**. Nevertheless, it is possible to conclude that PostgreSQL with JSON data type is able to compete with MongoDB in data reading.

## 5.2 Summary of INSERT Operations Measurement Results

INSERT operations mean writing new propositions to the database that do not replace existing propositions. Common Web application issues a relatively small number of inserts. Nonetheless, data write performance is important, because it may not benefit from the cache mechanisms (probably, except of some deferred write techniques). Usually data write is issued after some action of the user. The speed of the response depends on data write performance. Consequently, better execution speed of write operation means faster response time for the end user.

Hereafter, two charts illustrating the experiments results and the summary table are presented.

The first chart on Figure 24 demonstrates the Throughput values. The greater the value the better.

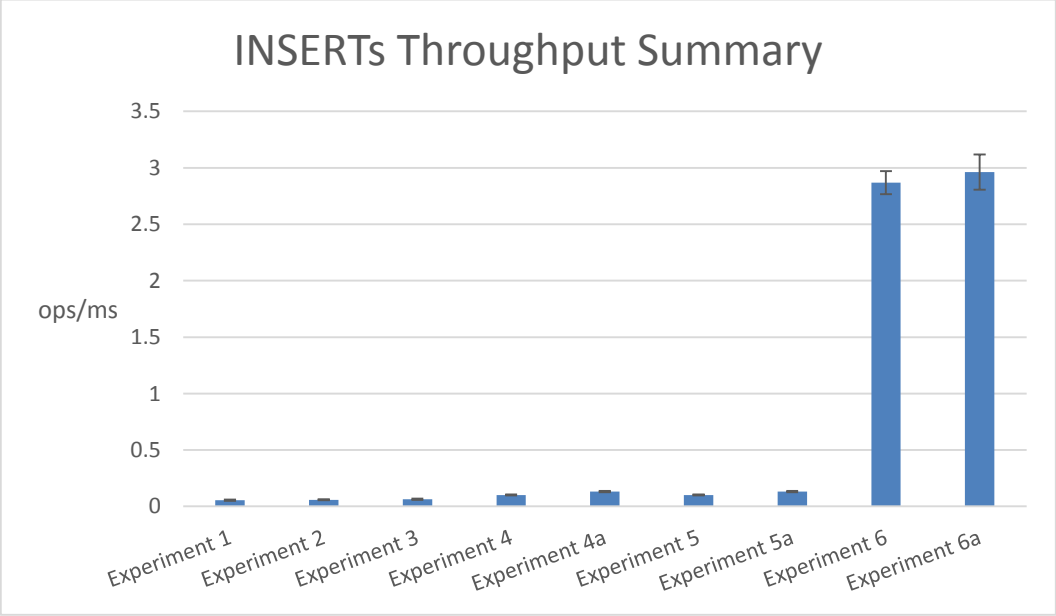


**Figure 24. INSERT operations throughput summary chart.**

The second chart on Figure 25 demonstrates values of AverageTime, SampleTime, and SingleShotTime. The smaller the value the better.

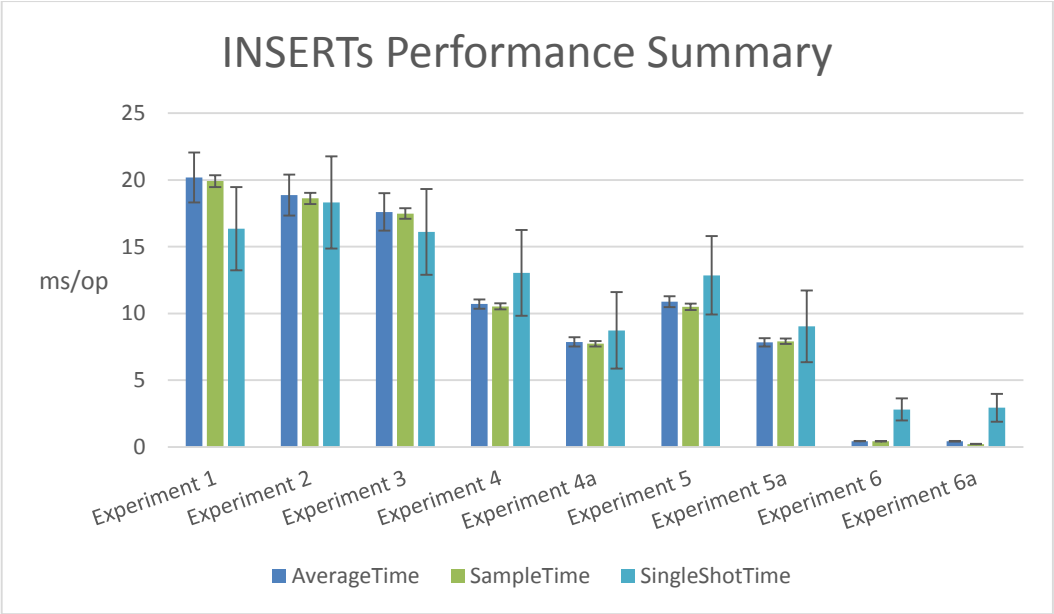


**Figure 25. INSERT operations performance summary chart.**

Table 16 presents results for all experiments. The best and the worst values within every measurement mode are marked with stronger and dashed outline respectively. The most and the least performant experiment is marked with green and red fill.

**Table 16. Experiments summary for INSERTS.**

| | **Throughput** (ops/ms) | **AverageTime** (ms/op) | **SampleTime** (ms/op) | **SingleShotTime** (ms/op) |
|---|---|---|---|---|
| **Experiment 1** | 0.056 ± 0.005 | 20.188 ± 1.877 | 19.911 ± 0.454 | 16.344 ± 3.116 |
| **Experiment 2** | 0.059 ± 0.005 | 18.859 ± 1.533 | 18.618 ± 0.416 | 18.313 ± 3.462 |
| **Experiment 3** | 0.064 ± 0.005 | 17.605 ± 1.398 | 17.482 ± 0.395 | 16.107 ± 3.210 |
| **Experiment 4** | 0.102 ± 0.004 | 10.702 ± 0.357 | 10.531 ± 0.233 | 13.033 ± 3.211 |
| **Experiment 4a** | 0.132 ± 0.006 | 7.869 ± 0.347 | 7.727 ± 0.195 | 8.728 ± 2.867 |
| **Experiment 5** | 0.102 ± 0.004 | 10.884 ± 0.402 | 10.499 ± 0.239 | 12.853 ± 2.936 |
| **Experiment 5a** | 0.132 ± 0.006 | 7.843 ± 0.314 | 7.918 ± 0.197 | 9.034 ± 2.687 |
| **Experiment 6** | 2.869 ± 0.102 | 0.432 ± 0.020 | 0.423 ± 0.010 | 2.807 ± 0.836 |
| **Experiment 6a** | 2.961 ± 0.156 | 0.426 ± 0.020 | 0.219 ± 0.010 | 2.932 ± 1.037 |

Table 17 demonstrates all experiments' relative performance interrelation for INSERT operations.

**How to read this table**

Let us assume that one wants to lookup how the second experiment (Exp2) correlates to the first experiment (Exp1). One is looking for Exp2 tagged row on X-axis and for Exp1 tagged column on Y-axis. Afterwards, one is searching for the intersection of found row and column. The intersection shows a value of 93% with green color. It means that the second experiment (Exp2) is $\frac{100\%}{93\%} = \mathbf{1.075}$ **times more** performant than the first experiment (Exp1).

Now the inverse lookup: how the first experiment (Exp1) correlates to the second experiment (Exp2). One is looking for Exp1 tagged row on X-axis and for Exp2 tagged column on Y-axis. Afterwards, one is searching for the intersection of found row and column. The intersection shows a value of 107% with red color. It means that the first experiment (Exp1) is $\frac{107\%}{100\%} = \mathbf{1.07}$ **times less** performant than the second experiment (Exp2).

**Table 17. Experiments relative comparison for INSERT operations.**

| x \ y | Exp1 | Exp2 | Exp3 | Exp4 | Exp4a | Exp5 | Exp5a | Exp6 | Exp6a |
|---|---|---|---|---|---|---|---|---|---|
| Exp1 | 100% | 107% | 115% | 189% | 257% | 185% | 257% | 4673% | 4739% |
| Exp2 | 93% | 100% | 107% | 176% | 240% | 173% | 240% | 4366% | 4427% |
| Exp3 | 87% | 93% | 100% | 165% | 224% | 162% | 224% | 4075% | 4133% |
| Exp4 | 53% | 57% | 61% | 100% | 136% | 98% | 136% | 2477% | 2512% |
| Exp4a | 39% | 42% | 45% | 74% | 100% | 72% | 100% | 1822% | 1847% |
| Exp5 | 54% | 58% | 62% | 102% | 138% | 100% | 139% | 2519% | 2555% |
| Exp5a | 39% | 42% | 45% | 73% | 100% | 72% | 100% | 1816% | 1841% |
| Exp6 | 2% | 2% | 2% | 4% | 5% | 4% | 6% | 100% | 101% |
| Exp6a | 2% | 2% | 2% | 4% | 5% | 4% | 5% | 99% | 100% |

The most performant experiments with PostgreSQL are Experiment 4a (JDBC + PostgreSQL JSON data type) and Experiment 5a (JDBC + PostgreSQL JSONB data type). They showed almost identical results.

MongoDB with WiredTiger storage engine showed slightly better results, than with MMAPv1 engine. The former is 1.01 times faster than the latter, which is insignificant difference.

Summarizing, experiments with **MongoDB** showed approximately **18 times better** performance **for INSERT operations than** the most performant experiment with **PostgreSQL**.

## 5.3 Summary of UPDATE Operations Measurement Results

Update operations is replacing some proposition in a database with a new proposition. Common Web application issues a relatively small number of updates. Data write performance is important, because it may not benefit from cache mechanisms (probably, except of some deferred write techniques). Usually data write is issued after some action of the user; the speed of the response depends on data write performance. Therefore, better execution speed of write operation means faster response time for the end user.

Hereafter, two charts illustrating the experiments results and the summary table are presented.

The first chart on Figure 26 demonstrates the Throughput values. The greater the value the better.

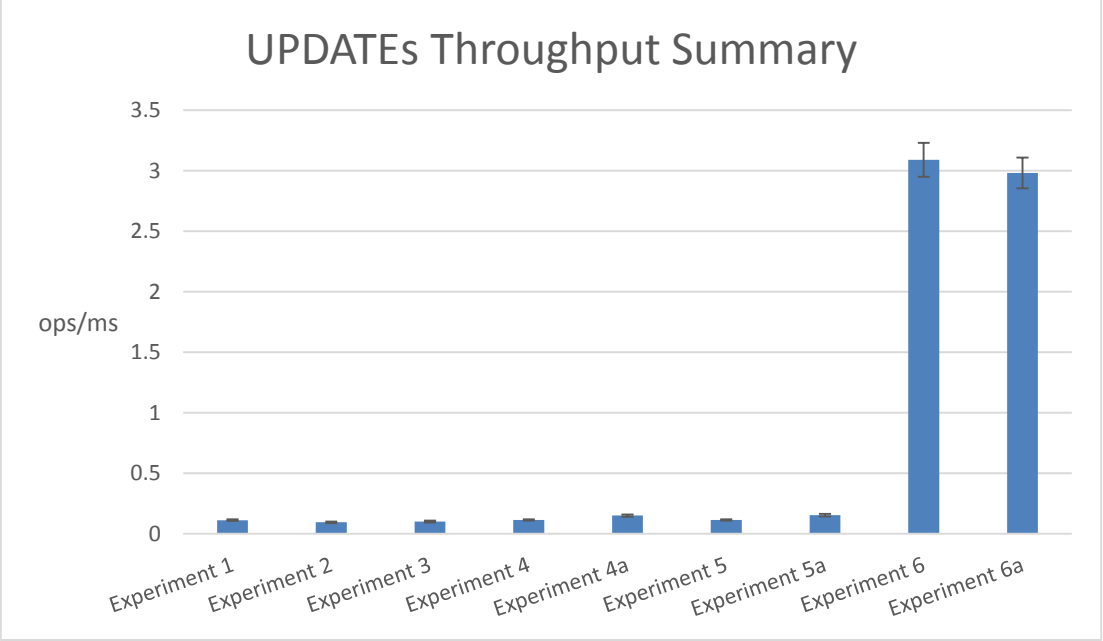


**Figure 26. UPDATE operations throughput summary chart.**

The second chart on Figure 27 demonstrates values of AverageTime, SampleTime, and SingleShotTime. The smaller the value the better.

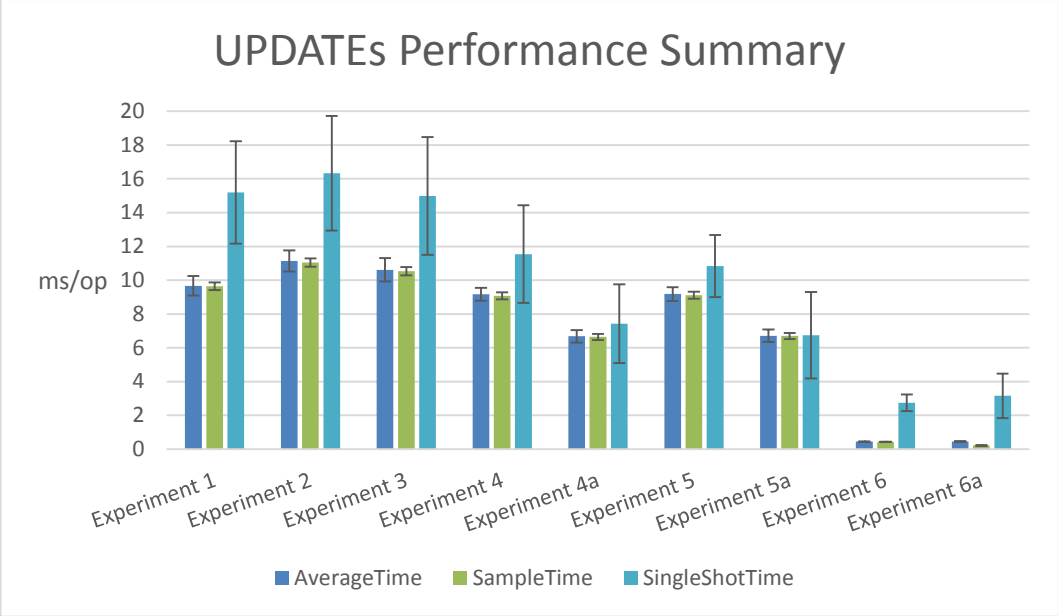


**Figure 27. UPDATE operations performance summary chart.**

Table 18 presents results for all experiments. The best and the worst values within every measurement mode are marked with stronger and dashed outline respectively. The most and the least performant experiment is marked with green and red fill.

**Table 18. Experiments summary for UPDATES.**

| | Throughput (ops/ms) | AverageTime (ms/op) | SampleTime (ms/op) | SingleShotTime (ms/op) |
|---|---|---|---|---|
| **Experiment 1** | 0.113 ± 0.006 | 9.668 ± 0.583 | 9.642 ± 0.229 | 15.195 ± 3.038 |
| **Experiment 2** | 0.095 ± 0.006 | 11.145 ± 0.626 | 11.042 ± 0.250 | 16.337 ± 3.391 |
| **Experiment 3** | 0.101 ± 0.007 | 10.613 ± 0.688 | 10.542 ± 0.246 | 14.979 ± 3.485 |
| **Experiment 4** | 0.114 ± 0.006 | 9.169 ± 0.380 | 9.079 ± 0.211 | 11.545 ± 2.883 |
| **Experiment 4a** | 0.151 ± 0.009 | 6.681 ± 0.373 | 6.643 ± 0.179 | 7.427 ± 2.333 |
| **Experiment 5** | 0.115 ± 0.005 | 9.181 ± 0.406 | 9.111 ± 0.211 | 10.841 ± 1.842 |
| **Experiment 5a** | 0.154 ± 0.010 | 6.709 ± 0.369 | 6.699 ± 0.179 | 6.741 ± 2.559 |
| **Experiment 6** | 3.090 ± 0.141 | 0.446 ± 0.012 | 0.436 ± 0.003 | 2.748 ± 0.486 |
| **Experiment 6a** | 2.981 ± 0.127 | 0.454 ± 0.016 | 0.224 ± 0.007 | 3.159 ± 1.318 |

Table 19 demonstrates all experiments' relative performance interrelation for UPDATE operations.

**How to read this table**

Let us assume that one wants to lookup how the second experiment (Exp2) correlates to the first experiment (Exp1). One is looking for Exp2 tagged row on X-axis and for Exp1 tagged column on Y-axis. Afterwards, one is searching for the intersection of found row and column. The intersection shows a value of 115% with red color. It means that the second experiment (Exp2) is $\frac{115\%}{100\%} = \mathbf{1.15}$ **times less** performant than the first experiment (Exp1).

Now the inverse lookup: how the first experiment (Exp1) correlates to the second experiment (Exp2). One is looking for Exp1 tagged row on X-axis and for Exp2 tagged column on Y-axis. Afterwards, one is searching for the intersection of found row and column. The intersection shows a value of 87% with red color. It means that the first experiment (Exp1) is $\frac{100\%}{87\%} = \mathbf{1.15}$ **times more** performant than the second experiment (Exp2).

**Table 19. Experiments relative comparison for UPDATE operations.**

| x \ y | Exp1 | Exp2 | Exp3 | Exp4 | Exp4a | Exp5 | Exp5a | Exp6 | Exp6a |
|---|---|---|---|---|---|---|---|---|---|
| Exp1 | 100% | 87% | 91% | 105% | 145% | 105% | 144% | 2168% | 2130% |
| Exp2 | 115% | 100% | 105% | 122% | 167% | 121% | 166% | 2499% | 2455% |
| Exp3 | 110% | 95% | 100% | 116% | 159% | 116% | 158% | 2380% | 2338% |
| Exp4 | 95% | 82% | 86% | 100% | 137% | 100% | 137% | 2056% | 2020% |
| Exp4a | 69% | 60% | 63% | 73% | 100% | 73% | 100% | 1498% | 1472% |
| Exp5 | 95% | 82% | 87% | 100% | 137% | 100% | 137% | 2059% | 2022% |
| Exp5a | 69% | 60% | 63% | 73% | 100% | 73% | 100% | 1504% | 1478% |
| Exp6 | 5% | 4% | 4% | 5% | 7% | 5% | 7% | 100% | 98% |
| Exp6a | 5% | 4% | 4% | 5% | 7% | 5% | 7% | 102% | 100% |

The most performant experiments with PostgreSQL are Experiment 4a (JDBC + PostgreSQL JSON data type) and Experiment 5a (JDBC + PostgreSQL JSONB data type). They showed almost identical results.

MongoDB with MMAPv1 storage engine showed slightly better results, than with WiredTiger engine. The former is 1.02 times faster than the latter, which is insignificant difference.

Summarizing, experiments with **MongoDB** showed approximately **15 times better** performance **for UPDATE operations than** the most performant experiment with **PostgreSQL**.

## 5.4 Discussion

According to the results of different measurements mentioned in this thesis, it is possible to conclude that the results of performance comparisons heavily depend on tools, which are used to accomplish the comparisons. Moreover, in case of each such comparison one should defined the target audience that representatives could make decisions based on the result of the performance comparison. For instance, in case of the present measurements and comparisons, the target audience is software developers.

There were two different performance benchmarking approaches (that specify process and tools) described in this thesis.

The first one was initially carried out by the EnterpriseDB. They developed and open-sourced a special testing framework for this purpose. It uses console applications provided by the vendors of DBMSs to execute database operations. Tests showed that INSERT and SELECT operations are, correspondingly, three and four and a half times more performant in PostgreSQL. These tests were made with MongoDB 2.6 and PostgreSQL 9.4 beta. However, at the time of writing this thesis (spring 2015) MongoDB 3.0.2 and PostgreSQL 9.4.1 were available. Therefore, in the context of this thesis, the author made new tests by using the same testing framework. According to this INSERT and SELECT operations are still at least three time more performant in PostgreSQL.

However, the author assumed that there is a problem with this testing approach. The problem is that aforementioned testing framework is using console applications provided by DBMS vendors: *psql* and *mongo*. Probably it might be satisfactory for the target audience of system administrators. However, for the software developers the results of such performance measurement are at least unreliable. The reason is that applications are not using console applications to communicate with a database. They use a corresponding DBMS API and features of programming languages. That is, in order to make performance comparison of DBMSs for software developers, it is necessary to measure performance through an application by using the correct API.

The goal of this thesis was to test what performance PostgreSQL with JSON data types can offer for real applications and to verify as to whether the PostgreSQL can outperform MongoDB in this context. To achieve the goal, the author took a real world Java application and

reconfigured its database layer and its DAO classes to use PostgreSQL and MongoDB with different configurations. In total nine different database/application designs were introduced, implemented, and performance of operations measured. The results are opposite to that the author received by using EnterpriseDB's testing framework. MongoDB NoSQL DBMS outperforms PostgreSQL with JSON data types if used through some popular APIs.

The main reason of faster data write in case of MongoDB may be that MongoDB uses the *acknowledged* write concern by default [37]. The acknowledged write concern means that the MongoDB confirms that it received the write operation and applied the change to the in-memory view of data. It does not confirm that the write operation has persisted to the disk system. The other reason may be that transaction management through the API in the application in case of PostgreSQL may influence the overall performance.

The tests also showed that the "standard" design of PostgreSQL with no use of JSON types lead mostly to slower performance than the designs with JSON types.

Of course, one must understand that there are multiple criteria based on that to decide the usage of a particular technology, architecture, and design. Performance is only one of them and depending the context, it could be more or less important.

The tests were made in the following context that affects the results of the benchmarking made for this thesis.

- No additional configuration of DBMSs was made after their installation.
- There were no additional indexes created neither for PostgreSQL nor for MongoDB. The only indexes used are the ones created automatically by the DBMS.
- The tests were made on one computer not on cluster of computers where NoSQL systems are quite often used.
- Each executed operation was treated as a transactional unit. No transactions that involve multiple operations were tested. PostgreSQL support multi-statement transactions and in MongoDB (3.0) one can emulate multi-statement "like" transactions by implementing two-phase commit algorithm in the application [38].
- Performance benchmarking was made only by using the application created in Java programming language.
- As in case of console applications, it is not exactly clear how much the implementation of the APIs influences the overall performance. However, at least now the DBMSs are tested in a realistic situation from the point of view of developers.

Limitations of this work open up new avenues for future work.

## 5.5 Future Work

Future work should include additional experiments to explore the results in case of some other context than previously mentioned. For instance, some "correct" configuration and/or fine-tuning of a DBMS may significantly improve its performance. One could create indexes to JSON data in both PostgreSQL and MongoDB and it should affect both read and write performance. Some other programming language than Java may show different results.

It would be interesting to dig deep to the internals of PostgreSQL (because it is open source it is possible) to find out what could cause the performance problems.

Moreover, results of the current work may have several implications for further research and analysis.

It is obvious that a NoSQL system is great for writing high amounts of data, whereas SQL system is generally more reliable because of transactions. Let us assume that there is an application that operates over two sets of data. One set of data is very important (transactions between bank accounts) and the second set of data is not important, but just useful for some reason (for instance, it may some kind of "audit log" within the application). It would be interesting to compare real world application's performance when it used only SQL DBMS and when it uses SQL DBMS for the first set of data and some NoSQL system for another set of data. This is the idea of Polyglot Persistence. Further investigation of "application managed transactions" is also interesting. This would allow developers to start to consider using NoSQL systems that do not support multi-statement transactions for the processing of important data.

## Summary

*Performance Comparison of MongoDB and PostgreSQL with JSON types*

The goal of this work was to create a set of benchmarks, to measure the performance of PostgreSQL with JSON data types, and to compare it with the performance of MongoDB NoSQL system from the perspective of a real world Java application. This goal was inspired by the debates that introduction of JSON data types in PostgreSQL not only provides the flexibility of a documents-based data model but offers comparable or even better performance.

Firstly, the author shortly described the theoretical background that stands behind SQL and NoSQL systems, as well as Java specific technologies for database communication. The second chapter described the performance comparison of PostgreSQL with JSON data type and MongoDB, which was firstly done by the EnterpriseDB (the leading worldwide provider of PostgreSQL software). Additionally, the author made a performance comparison of the newest (at the time of writing) PostgreSQL (9.4) and MongoDB (3.0) versions by using the same benchmarking framework. Both these comparisons showed that PostgreSQL actually significantly outperforms NoSQL system MongoDB. However, the author doubted the results, because the benchmarking tool was written and provided by EnterpriseDB. It is a company that provides support and tools for PostgreSQL. This tool uses console applications by calling separate console commands for every action. The performance of these console applications may significantly influence benchmarking results. Moreover, console applications are not used as mediators if an application wants to interact with a DBMS.

The main part of the work started from the third chapter. There the author described the database layer of Java application, which served as a platform for experimenting and benchmarking. Most importantly, the different designs of possible database layer implementations were introduced and briefly described. Lastly, JMH, the tool utilized for performance benchmarks creation, was explained.

The fourth chapter introduced the nine experiments based on the aforementioned designs. The author explained the implementation of database layer and corresponding benchmarks for every experiment. Aggregated results of benchmarking, divided into three categories for SELECT, INSERT, and UPDATE operations, were presented as well.

The fifth chapter presented an analysis of the outcomes of the experiments. It contains one section for every category of operations. Each section contains several summary charts and tables with the results of experiments.

As a result of this work, the following conclusions about PostgreSQL (9.4) with JSON data types and MongoDB (3.0) performance comparison were made.

- SELECT operations or data read performance from Java application in PostgreSQL with JSON data types is almost identical with MongoDB.
- INSERT operations or data write performance from Java application in MongoDB is 18 times more performant than in PostgreSQL with JSON data types.
- UPDATE operations performance from Java application in MongoDB is 15 times more performant than in PostgreSQL with JSON data types.

To summarize, the introduction of JSON data types in PostgreSQL does not mean that it may replace NoSQL systems. Performance of PostgreSQL for data write is drastically lower than the performance of the considered NoSQL system. The biggest merit of JSON data types in PostgreSQL is that it may offer certain flexibility of the document-based data model in a traditional SQL DBMS. Results of this work proved once more that one must find a proper tool for every problem. The DBMSs will come and go but the idea that one should get from this thesis is that it is both necessary and possible to test the performance of DBMSs from the point of view of software developers by using tools that are in their disposal.

**Kokkuvõte**

*MongoDB ja JSON tüüpe kasutava PostgreSQLi jõudluse võrdlemine*

Käesoleva töö eesmärk on mõõta JSON tüüpe pakkuva PostgreSQL andmebaasisüsteemi jõudlust ja võrrelda seda MongoDB NoSQL süsteemi jõudlusega reaalses Java rakenduses. Eesmärk sai inspiratsiooni aruteludest, kas JSON andmetüüpide kasutusvõimalus PostgreSQLis mitte ainult ei paku dokumendipõhise andmemudeli paindlikkust, vaid ka NoSQL süsteemidega võrreldavat või isegi paremat jõudlust.

Esmalt kirjeldab autor lühidalt SQL-i ja NoSQL süsteemide teoreetilist tausta ning andmebaasidega suhtlemiseks mõeldud Java spetsiifilisi tehnoloogiaid. Teises peatükis kirjeldatakse JSON tüüpe pakkuva PostgreSQLi ja MongoDB jõudluse võrdlust, mille viis esmalt läbi EnterpriseDB (juhtiv PostgreSQL tarkvara pakkuja). Lisaks teostas autor sama võrdlusraamistikku kasutades jõudluse võrdluse (töö kirjutamise ajal) kõige uuema PostgreSQL-i (9.4) ja MongoDB (3.0) versiooni vahel. Mõlemad võrdlused näitasid, et PostgreSQL pakub märgatavamalt paremat jõudlust kui NoSQL süsteem MongoDB. Autor kahtles nendes tulemustes, kuna kasutatud võrdlusraamistik on toodetud EnterpriseDB poolt. Sama ettevõte pakub ka tuge ja rakendusi PostgreSQL-le. See töövahend kasutab terminalirakendusi, kutsudes välja nende käsurea käske. Terminalirakenduse jõudlus võib märgatavalt mõjutada mõõtmiste tulemusi. Lisaks ei ole terminalirakendused kasutuse rakenduste ja andmebaasisüsteemide suhtluse vahendajatena.

Töö põhiosa algab kolmandast peatükist, kus kirjeldatakse eksperimenteerimise ja mõõtmise platvormiks olnud Java rakenduse andmebaasikihti. Lühidalt on kirjeldatud erinevaid disaine andmebaasikihi realiseerimiseks ning lõpetuseks antakse ülevaade jõudluse mõõtmise rakendusest JMH.

Neljandas peatükis tutvustatakse üheksat eelpool nimetatud eksperimentidel põhinevat disaini. Teostatud on andmebaasikihi realisatsioon ja loodud vastav mõõtmisraamistik igale eksperimendile. Esitatakse kogutud tulemused, mis on jaotatud kolme kategooriasse – SELECT, INSERT ja UPDATE operatsioonid.

Analüüsi tulemused esitatakse viiendas peatükis, mis sisaldab alajaotust iga operatsioonide kategooria jaoks. Iga alajaotus sisaldab omakorda mitmeid skeeme ja tabeleid eksperimentide tulemuste kohta.

Käesoleva töö tulemusena jõuti andmebaasisüsteemide PostgreSQL (koos JSON andmetüüpidega) ja MongoDB jõudluse võrdlemisega järgmiste tulemusteni.

- SELECT operatsioonid e andmete lugemise jõudlus Java rakendusest on PostgreSQLi (koos JSON tüüpidega) ja MongoDB korral peaaegu ühesugune.
- INSERT operatsioonid e andmete kirjutamine Java rakendusest MongoDB on 18 korda kiirem kui PostgreSQL (koos JSON tüüpidega) korral.
- UPDATE operatsioonid e andmete kirjutamine Java rakendusest MongoDB on 15 korda kiirem kui PostgreSQL (koos JSON tüüpidega) korral.

Testid näitasid ka seda, et PostgreSQLi standardne disain kus JSON tüüpe ei kasutatud viis enamasti madalama jõudluseni kui disainid, kus kasutati JSON tüüpe.

Kokkuvõtvalt, JSON andmetüüpide kasutuselevõtt PostgreSQL-is ei tähenda, et PostgreSQL võiks asendada NoSQL süsteeme. PostgreSQL-i andmete kirjutamise jõudlus on drastiliselt madalam kui jõudlus käsitletud NoSQL süsteemis. JSON andmetüüpide PostgreSQLi poolt toetamise suurim väärtus on, et see võib pakkuda teatavat dokumendipõhisest andmemudelist tulenevat paindlikust. Käesoleva töö tulemused tõestasid, et igale probleemile on vaja leida selle lahendamiseks sobiv vahend. Andmebaasisüsteemid tulevad ja lähevad. Idee mille lugeja võiks sellest tööst kaasa võtta on, et on vajalik ja võimalik hinnata andmebaasisüsteemide töökiirust tarkvaraarendajate vaatepunktist, kasutades nende käsutuses olevaid töövahendeid.

## References


[1] *Information technology - Home Electronic System - Guidelines for product interoperability - Part 1: Introduction: ISO/IEC 18012-1:2004(en).*

[2] *Systems and software engineering - Vocabulary: ISO/IEC/IEEE 24765:2010(en).*

[3] "BSON - Binary JSON," [Online]. Available: http://bsonspec.org/. [Accessed 9 May 2015].

[4] "CMS noun - Oxford Advanced Learner's Dictionary," [Online]. Available: http://www.oxfordlearnersdictionaries.com/definition/english/cms?q=cms. [Accessed 9 May 2015].

[5] *Information technology - Reference Model of Data Management: ISO/IEC TR 10032:2003(en).*

[6] EnterpriseDB, "About EnterpriseDB | EnterpriseDB Web site," EnterpriseDB, [Online]. Available: http://www.enterprisedb.com/company/about-enterprisedb. [Accessed 9 May 2015].

[7] Oracle Corparation, "JDBC Introduction (The Java(TM) Tutorials > JDBC(TM) Database Access)," [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/overview/. [Accessed 9 May 2015].

[8] Oracle OpenJDK, "Code Tools: jmh," [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/. [Accessed 21 April 2015].

[9] Oracle Corparation, "Introduction to the Java Persistence API," [Online]. Available: https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm#BNBPZ. [Accessed 9 May 2015].

[10] T. Lindholm, F. Yellin, G. Bracha and A. Buckley, "The Java® Virtual Machine Specification," Oracle Corporation, 28 February 2013. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/index.html. [Accessed 9 May 2015].

[11] MongoDB Inc., "MongoDB Web site," MongoDB Inc., [Online]. Available: https://www.mongodb.org/. [Accessed 9 May 2015].

[12] M. Keith and M. Schincariol, Pro JPA 2, Apress Media, 2013.

[13] M. Fowler, "PolyglotPersistence," 16 November 2011. [Online]. Available: http://martinfowler.com/bliki/PolyglotPersistence.html. [Accessed 20 April 2015].

[14] *Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework): ISO/IEC 9075-1:200x..*

[15] M. Fowler, "NosqlDefinition," 09 January 2012. [Online]. Available: http://martinfowler.com/bliki/NosqlDefinition.html. [Accessed 16 April 2015].

[16] MongoDB, Inc., "Top 5 Considerations When Evaluating NoSQL Databases," 2015.

[17] P. Sadalage, "NoSQL Databases: An Overview," 3 October 2014. [Online]. Available: http://www.thoughtworks.com/insights/blog/nosql-databases-overview. [Accessed 16 April 2015].

[18] "DB-Engines Ranking," [Online]. Available: http://db-engines.com/en/ranking. [Accessed 16 April 2015].

[19] MongoDB, Inc, "Introduction to MongoDB," 2015. [Online]. Available: http://docs.mongodb.org/manual/core/introduction/. [Accessed 17 April 2015].

[20] A. Pradhan, "Embedded data model in MongoDB," 7 December 2013. [Online]. Available: http://www.dbtalks.com/UploadFile/e6aced/embedded-data-model-in-mongodb/. [Accessed 25 April 2015].

[21] Wikipedia, the free encyclopedia, "ANSI-SPARC Architecture," [Online]. Available: http://en.wikipedia.org/wiki/ANSI-SPARC_Architecture. [Accessed 8 May 2015].

[22] "Database Normalization," 2014. [Online]. Available: http://www.1keydata.com/database-normalization/. [Accessed 18 April 2015].

[23] The PostgreSQL Global Development Group, "E.20. Release 9.2," 10 September 2012. [Online]. Available: http://www.postgresql.org/docs/9.4/static/release-9-2.html. [Accessed 18 April 2015].

[24] The PostgreSQL Global Development Group, "E.2. Release 9.4," 18 December 2014. [Online]. Available: http://www.postgresql.org/docs/9.4/static/release-9-4.html. [Accessed 18 April 2015].

[25] The PostgreSQL Global Development Group, "8.14. JSON Types," 2014. [Online]. Available: http://www.postgresql.org/docs/9.4/static/datatype-json.html. [Accessed 18 April 2015].

[26] "Hibernate Reference Documentation," Red Hat, Inc., 15 April 2015. [Online]. Available: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/. [Accessed 19 April 2015].

[27] R. Biswas and E. Ort, "The Java Persistence API - A Simpler Programming Model for Entity Persistence," May 2006. [Online]. Available: http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html. [Accessed 19 April 2015].

[28] C. Roe, "ACID vs. BASE: The Shifting pH of Database Transaction Processing," 1 March 2012. [Online]. Available: http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/. [Accessed 10 May 2015].

[29] M. Linster, "Postgres Outperforms MongoDB and Ushers in New Developer Reality," 24 September 2014. [Online]. Available: http://www.enterprisedb.com/postgres-plus-edb-blog/marc-linster/postgres-outperforms-mongodb-and-ushers-new-developer-reality. [Accessed 20 April 2015].

[30] E. Horowitz, "Announcing MongoDB 3.0," 3 February 2015. [Online]. Available: http://www.mongodb.com/blog/post/announcing-mongodb-30. [Accessed 20 April 2015].

[31] EnterpriseDB, "EnterpriseDB/pg_nosql_benchmark," [Online]. Available: https://github.com/EnterpriseDB/pg_nosql_benchmark. [Accessed 20 April 2015].

[32] M. Kaplan, "When to Use MongoDB Rather than MySQL (or Other RDBMS): The Billing Example," 3 March 2014. [Online]. Available: http://java.dzone.com/articles/when-use-mongodb-rather-mysql. [Accessed 20 April 2015].

[33] S. Mei, "Why You Should Never Use MongoDB," 11 November 2013. [Online]. Available: http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/. [Accessed 20 April 2015].

[34] ByteLife Solutions, "FlowGrab.com," 2014. [Online]. Available: https://flowgrab.com/. [Accessed 21 April 2015].

[35] Oracle, "JMH JavaDoc," [Online]. Available: http://javadox.com/org.openjdk.jmh/jmh-core/0.9/overview-summary.html. [Accessed 21 April 2015].

[36] D. E. Knuth, "Structured Programming with go to Statements," *Computing Surveys,* vol. 6, no. 4, pp. 261-301, 1974.

[37] MongoDB Inc., "Write Concern," [Online]. Available: http://docs.mongodb.org/manual/core/write-concern/#write-concern-acknowledged. [Accessed 11 May 2015].

[38] MongoDB, Inc, "Perform Two Phase Commits," [Online]. Available: http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/. [Accessed 09 May 2015].

[39] D. E. A. Brewer, "Towards Robust Distributed Systems," 19 July 2000. [Online]. Available: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf. [Accessed 10 May 2015].

# Appendix 1

Table 20 shows the number of physical lines of code in case of different DAO classes. The number shows the actual code lines. Comments and blank lines were not included.

**Table 20. Source lines of code count for DAO implementations.**

| | **1** | **2, 3** | **4, 5** | **4a, 5a** | **6, 6a** |
|---|---|---|---|---|---|
| *AbstractDAO.java* | 10 | 10 | 23 | 23 | 20 |
| *BlacklistedNameDAO.java* | 8 | 8 | 31 | 48 | 32 |
| *CarouselContentDAO.java* | 9 | 9 | 32 | 59 | 38 |
| *CategoryDAO.java* | 12 | 12 | 55 | 63 | 35 |
| *CommitChangeLogDAO.java* | 11 | 11 | 80 | 88 | 35 |
| *EnduserDAO.java* | 56 | 56 | 440 | 427 | 196 |
| *EnduserTeamDAO.java* | 24 | 24 | 132 | 171 | 59 |
| *LicenceDAO.java* | 8 | 8 | 30 | 44 | 32 |
| *MergeResultDAO.java* | 18 | 18 | 103 | 135 | 95 |
| *NotificationDAO.java* | 33 | 33 | 145 | 181 | 86 |
| *PluginVersionDAO.java* | 8 | 8 | 35 | 54 | 38 |
| *ProjectDAO.java* | 65 | 67 | 620 | 596 | 259 |
| *ProjectSubscriberDAO.java* | 29 | 29 | 109 | 121 | 49 |
| *ProjectVersionDAO.java* | 71 | 72 | 251 | 307 | 211 |
| *ProjectVersionDownloadDAO.java* | 7 | 7 | 31 | 31 | 23 |
| *RoleTypeDAO.java* | 38 | 38 | 126 | 136 | 59 |
| *TagDAO.java* | 30 | 30 | 111 | 145 | 189 |
| *TeamDAO.java* | 24 | 24 | 224 | 232 | 99 |
| ***SUM*** | **461** | **464** | **2578** | **2861** | **1555** |