Technische Universität München
Department of Informatics

Bachelor's Thesis in Informatics

Evaluating Databases for the Internet of Things

David Gogrichiani

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Evaluating Databases for the Internet of Things

Evaluation von Datenbanken für das Internet der Dinge

Author	David Gogrichiani
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	M. Sc. Stefan Liebald, Dr. Marc-Oliver Pahl
Date	December 15, 2017

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, December 15, 2017

Signature

Abstract

In this thesis our goal is to evaluate suitability different database types for Internet of Thing on example of Distributed Smart Space Orchestration System (DS2OS). For this purpose we present main concepts behind DS2OS and its component Virtual State Layer, that has most relevance for us. Additionally we collect related work on comparison of different databases and their performance. To achieve our goal we select and implement several databases for Virtual State Layer, based on our own assessments and related work. Then we create a benchmark, that can compare the existing solution with our own. After we execute the benchmark, that consists of several evaluation scenarios, and evaluate the results. Based on the results we assess, whether and under which conditions different solution is better than the existing database backend.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Goals of the thesis

In this thesis we evaluate suitability of different types of databases on the Distributed Smart Space Orchestration System (DS2OS). DS2OS is a system developed at the Chair of Network Architectures and Services at the Technical University of Munich. As its name suggests DS2OS is designed to manage and orchestrate Smart Spaces. One its part is Virtual State Layer, that is most relevant for the purpose of this thesis. One of responsibilities of VSL is to store the information about the state of IoT devices, that are present in Smart Space. In the current implementation of VSL a relational database, HyperSQLDB, is used as a data management system. We question whether other database types can be better suited for VSL, considering the specific tree-like structure of the VSL data. To answer that question we choose and implement other data management systems for VSL. Additionally we design and create a benchmark, that helps us evaluate the performance of implemented and existing solutions.

1.1.1 Methodology

To reach the above mentioned goal, we perform following steps:

- We choose several SQL and NoSQL databases based on defined requirements and existing evaluations of their performance.
- We adapt current data model to fit a selected type of database, without sacrificing defined functionality of Virtual State Layer.
- We implement chosen databases, following the adapted data model.
- We design several evaluation scenarios based on defined metrics.
- We implement a DS2OS service that performs the evaluation scenarios.

- After performing above mentioned scenarios we evaluate under which conditions it would be beneficial to use which type of database.

1.2 Outline

The thesis is structured as follows. In Chapter 2 we describe DS2OS and its main parts relevant for this thesis. Then we introduce how the data is organized and stored in the current implementation. After, we describe requirements imposed on the databases, considering the structure of the data and operations that are issued to the database backend. Finally we introduce several types of SQL and NoSQL databases, describe main concepts behind them and discuss their strengths and drawbacks. The discussion takes into account the requirements that we described earlier.

In Chapter 3 we present current research on comparison of different databases, within ones type and cross-types. Along with that we present current research on how to store the data, structurally similar to the data within Virtual State Layer.

In Chapter 4 we describe how the data should be structured to fit efficiently in different types of databases. Different approaches are discussed and evaluated, considering requirements of Virtual State Layer. In the same chapter we define several evaluating scenarios, that we will use assess the suitability of different database backends for Virtual State Layer.

In Chapter 5 we present, how we implemented different databases for Virtual State Layer. What were the challenges that we encountered, and which techniques we used to solve these challenges. Furthermore we will present the implementation of a benchmark, that is used for simulation of above mentioned evaluation scenarios.

In Chapter 6 we present and discuss the benchmark results. Based on the results, we assess, whether our solution is better than existing, in which evaluation scenarios. And what could be the cause of either exceptional or rather bad performance of our solutions.

In Chapter 7 we summarize what was done in this thesis, discuss the limitations and what can be done in the future.

Chapter 2

Analysis

In this chapter Distributed Smart Space Orchestration System (DS2OS), and one of its main components Virtual State Layer (VSL) are introduced. We describe their purpose, how they operate and interact. Then we illustrate what kind of data is generated and processed during the runtime of DS2OS. After we present how this data is structured and currently stored in the database backend.

The main purpose of this thesis is to evaluate possible candidates for a role as database backend of VSL. Therefore we introduce different database architectures that can be used to store the VSL data. We describe their benefits and drawbacks regarding storing and processing data. Additionally we discuss requirements of VSL, that are imposed on a database. Furthermore we present possible metrics and scenarios that can help evaluate the performance of possible candidate.

2.1 Distributed Smart Space Orchestration System

In this section we give a short description of Distributed Smart Space Orchestration System (DS2OS) and its purpose. DS2OS was designed at the Chair of Network Architectures and Services at the Technical University of Munich. As the name suggests its main purpose is management of Smart Spaces.

A Smart Space is essentially a physical place, which contains number of "smart devices". These smart devices are connected within a network and can interact with each other. Through interaction they can perform various scenarios based on current and previous states. The main constrain in such environments is the heterogeneity of the smart devices.

DS2OS is designed to combat this constrain. It enables transparent communication between devices, creation of services, which operate on these devices, and creation of complex scenarios based on interaction of services.

At the moment implementation of DS2OS is under continuous development. It is written using Java 7 and is maintained by several employees at the Chair of Network Architectures and Services at the Technical University of Munich

2.1.1 Virtual State Layer

In this section we describe the structure and functionality of **Virtual State Layer** (VSL), one of most important components of DS2OS. All concepts and descriptions are derived from Ph.D. Thesis by Marc-Oliver Pahl [1].

VSL is a programming abstraction, which acts as a pervasive computing middleware. In contrast to existing solutions it can be described as **µ-middleware**.

> A µ-middleware is a middleware that provides only fundamental, non domain-specific functionality, and that supports transparent extension with functionality at run time via regular services.

Such architecture is achieved through usage of *Virtual Context*. *Virtual Context* uses *Virtual Nodes* to couple static structure of VSL context modes with dynamic services [1]. Through *Virtual Nodes* information can be delivered on-demand. For the scope of this thesis *Virtual Nodes* are not relevant, as they do not stored in a database.

2.1.1.1 Structure and concepts of VSL

In order to understand how Virtual State Layer works it is important to know main concepts behind it.

- **VSL Context Node** is a virtual representation of *real world object*, such as a light bulb №1 in the room №42. It contains the information of current state and properties of this object.
- **VSL Context Model** is a model describing the properties of the *generic real world object*. Context Node is an instance of a specific Context Model. Context Models are stored in Context Model Repository. It is desired that they are extended by developers of DS2OS through crowd-sourcing.
- **VSL Meta Model** describes, how the Context Models are constructed.

VSL Context Nodes can be either **Virtual** or **Regular**.

> *Virtual Nodes* are similar to the Proxy Pattern. They can be registered by services on a Knowledge Agent. When another service tries to retrieve the information from a Virtual Node, following happens. The service, that created a Virtual Node, receives a callback and sends the current information directly to the requesting service.

Regular Nodes are stored directly in the *Context Repository* on a Knowledge Agent. They also can be accessed by services. For services it is transparent, whether they access a *Regular Node* or a *Virtual Node*.

Knowledge Agents are main self-organizing entities of VSL. They run on different machines and are capable of detecting each other in the network. Each KA have two main functions: context management and context repository. Currently each of KAs have a isolated backend database. Distribution of data and metadata is performed on an application level. During runtime each KA advertises information about available Context Nodes on them to the other KAs.

In DS2OS each service binds itself to one of the KA. Through this bond service can advertise its context and query contexts of other services on local or remote Knowledge Agents. This creates a seamless interactive environment, where services are aware of all available devices.

Virtual State Layer API consists of 13 methods, which can be divided into 3 functional groups: *context access*, *access control* and *Virtual Context* [1]. Virtual Context methods manage registration and removal of Virtual Nodes. Access Control methods are used to manage registration, removal and authentication of services on the KA. Contex Access methods are used by services to manage the data on the VSL Nodes.

For the purpose of this thesis Context Access methods have most relevance. Here is a listing with a short description of each method:

- **get** *[nodeAddress]* returns the context at the specified address.
- **set** *[nodeAddress] [value]* changes the context at the specified address.
- **subscribe** *[nodeAddress] [callBack]* subscribes the context subtree that starts at the specified node address. If a context node within registered subtree changes, all subscribers receive a notification, through callback function.
- **unsubscribe** *[nodeAddress]* removes an existing subscription on the context subtree at the specified address.
- **lockSubtree** *[parentAddress] [callBack]* locks a context subtree for exclusive use.
- **unlockSubtree** *[parentAddress]* unlocks a context subtree from exclusive use and commits all changes in the unlocked subtree since the locking.
- **revertSubtree** *[parentAddress]* unlocks a context subtree from exclusive use and reverts all changes in the unlocked subtree since the locking.

During execution of evaluation scenarios we concentrate on the first two methods, namely *get* and *set*. They are expected to be the most used methods in a runtime environment. Therefore evaluation scenarios consisting of these methods can be considerate representative.

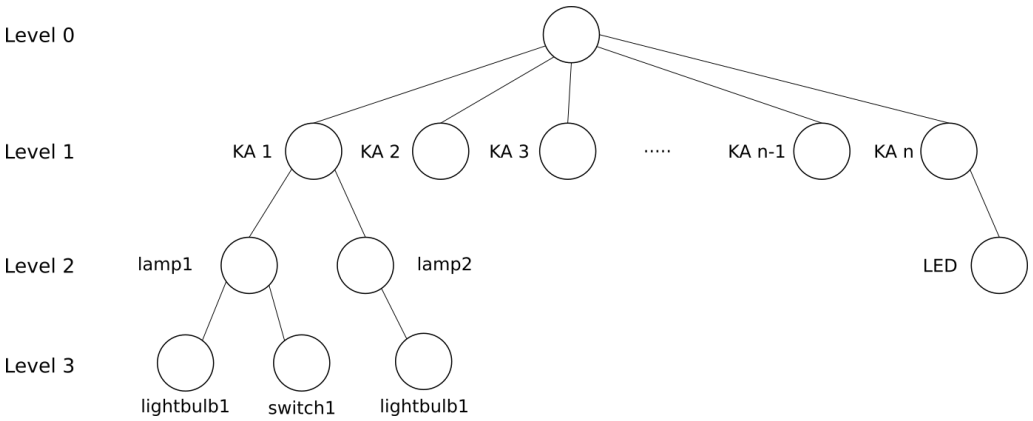

Figure 2.1: VSL structure

2.1.1.2 Model of Context Data

Knowledge Agents and data stored on them can be described as a tree-like structure. We can see a visual example in figure 2.1. KAs are placed in a root space and each VSL Context Node can be seen as sub-tree inside of Knowledge Agent [1]. Descendants of a Context Node represent important parts of the real-world object. This representation style allows to specify a real-world object with desired precision, depending on requirements. It is assumed that this tree can be arbitrary large in terms of depth and width. In the listing 2.1 we can see an example of an instance of such node.

```
<mySmartDevice type="/node/smartDevice" version="2" timeStamp="2017-06-16 12:11:25.126"
    access="rw">
 <button type="/node/button, /basic/composed" version="0" timeStamp="2017-06-16
     12:11:25.145" access="rw">
  <led type="/node/led,/node/isOn, /derived/boolean, /basic/number" version="1"
      timeStamp="2017-06-16 12:11:25.326" access="rw">
    1
   <desired type="/node/isOn, /derived/boolean, /basic/number" version="0" timeStamp="
       2017-06-16 12:11:25.463" access="rw">
    0
   </desired>
  </led>
 </button>
 <lightSensor type="/node/lightSensor, /basic/number" version="0" timeStamp="2017-06-16
     12:11:25.349" access="rw">
   800
 </lightSensor>
 <temperatureSensor type="/node/temperature, /basic/number" version="1" timeStamp="
     2017-06-16 12:11:25.171" access="rw">
   21
 </temperatureSensor>
</mySmartDevice>
```

Listing 2.1: Context Node example

A *smartDevice* consisting of a *button*, with *led* inside, a *lightSensor* and a *temperatureSensor*. Each part has a name, type, version, timestamp and access rights. As you can see, in most cases type consists of multiple entries. Types identify functionality that each part can have. Allowing a model to have multiple type helps describe its functionality more precise. For example we could have add "node/button/switchButton" as a type to describe buttons functionality more accurate.

Information that should be stored in database backend for each node, is summarized in the table 2.1. Each Context Node has an address, value, version, array of types, array of readers and writers IDs and information value restrictions and caching parameters.

address
value
version
types
readerIDs
writerIDs
restriction
cacheParameters

Table 2.1: Generic context node structure

Keeping an archive of all previous version of Context Tree is one of the required feature of Virtual State Layer. Therefore it is important to describe specific versioning scheme of a Context Node. When a node receives a value update, not only the node itself increment its version. Additionally all ancestors of the node up to the service level. This scheme is justified by viewing each Context Tree as a single real world object. Therefore if a part of an object changes its state, then all other parts, that include this specific updated part, are also considered as changed. This specific versioning scheme should be carefully considered during adaption of current data model for a new database backend.

2.1.1.3 Current implementation in HSQLDB

Currently relational database, *HSQLDB*, is used as main database backend on every Knowledge Agent in Virtual State Layer. In this section we present how the tree-like structure, described in section 2.1.1.2, is stored in HSQLDB.

The data is split into 3 tables:

structure : {[address, type, readers, writers, restriction, cachePatameters]}

version : {[address, timestamp, version]}

data : {[address, timestamp, value]}

Structure table contains structural information and is changed only in the moment of creation and deletion of the node. **address** is a general identifier of the node and used as a primary key in the data table. **type** field contains all types of the node. **restriction** field defines, what kind of values can a node have. Readers and writers fields, consists of ID that specify, which services can read or write this Context Node. Restriction field is used to indicate which value this Context Node can have.

Version table contains information about version of Context Node and its children. When a node gets an update in the Data Table, new entries are created in this table for this node and all of its parents up to the service root node. If multiple nodes are set in one operation, there is only one increment for all affected nodes (nodes, that are set and their parents). New entries have same **address**, new **timestamp**, which is identical on all nodes affected, and new **version**, which is created by incrementing the old version of that node by 1.

Data table behaves similarly to *Version table*. On every change of **value**, a new entry is created with new **timestamp**, and incremented **version**. The main difference is that on update of the node's children, the **timestamp** is not updated. The **timestamp** field indicates, when **value** of this particular node was changed.

address	LONGVARCHAR	"service_1/node_42/smartDevice/led"
type	LONGVARCHAR	"/node/led; /node/isOn; /derived/boolean; /basic/number"
readers	LONGVARCHAR	"service_2; service_3"
writers	LONGVARCHAR	"service_1"
restriction	LONGVARCHAR	"lowerBound = 0; upperBound = 1"
cacheParameters	LONGVARCHAR	"TTL = 3"

Table 2.2: Structure table

address	LONGVARCHAR	"service_1/node_42/smartDevice/led"
version	BIGINT	1
timestamp	TIMESTAMP	2017-06-16 12:11:25.326

Table 2.3: Version table

address	LONGVARCHAR	"service_1/node_42/smartDevice/led"
timestamp	TIMESTAMP	2017-06-16 12:11:25.326
value	LONGVARCHAR	1

Table 2.4: Data table

2.2 Metrics

To correctly construct evaluation scenarios for databases we have to define which parameters we can control and change. Also we have to name the metrics that we measure during the evaluation.

During runtime of VSL following operations are performed in a database backend.

- *Insert operations*
  Main interaction of KA with a database will be an insert of data. Using *set* command of VSL API, services add new values to the context nodes.
- *Update operation*
  In the current implementation of VSL concept, context nodes are not updated. Instead, in case of new value, a new context node, with a higher version is created and inserted in a context tree.
- *Read operations*
  The services in DS2OS interact with each other, often requesting information from each other. Through VSL API *get* command, the data queried can range from a one value inside of a tree to a set of values on a different levels of a tree.
- *Delete operations*
  After certain threshold old entries in context tree should be deleted. The frequency of delete operations depend on a threshold itself and the frequency of data updates.

When constructing an evaluating scenario we can alter following variables:

- *Type of performed operation*
  Services are using several fixed methods, when operating with a Context Node. These methods are listed in section 2.1.1.2. For the purpose of this thesis we will use only *get* and *set*, when constructing an evaluation scenario.
- *Frequency of performed operation*
  Depending on functions of the device represented by given Context Node, it can be read/written constantly or in predefined interval.
- *Ratio of performed operations*
  In most cases Context Node will be both read and written by different services, but for example a node representing an actuator will be written more frequently than read.

As mentioned in section 2.1.1.2 VSL Context Node has a tree-like structure. Therefore it is possible to vary following parameters:

- *Size of nodes*
  Size of the internal node value is variable.

- *Depth/breadth of the tree*
  Size of the context node can be adjusted in terms of both depth and breadth of the tree.
- *Inclusion/exclusion of the subtree*
  During the operations on the context node, subtrees can be either included or excluded.

During evaluation we have to measure following:

- *Performance under given workload*
  We measure an amount of time needed to perform a predefined number of operations. Such methodology was also used in this study [2]. Additionally we can measure how many operations can a database perform pro second.
- *Resource consumption*
  DS2OS is expected to run commodity hardware, therefore inadequate resources consumption by a database is undesirable.

Considering the number of possible application areas of DS2OS it is beneficial to construct several evaluating scenarios, which cover the most wide-spread usage patterns. This is discussed in detail in chapter 4.

2.3 Database Types

In this section we will present main databases types and discuss their applicability for storing VSL data. The main challenge is the hierarchical structure of VSL data. Therefore databases that provide native functionality to store and process such data, will have advantage over other types.

2.3.1 Relational databases

SQL or Relational databases have been a standard in the industry since 1970. But in recent decade they began to lose their popularity due to shift of requirements placed on the databases [3].

The core of SQL databases are multiple schemes of n to m relationships. A scheme consists of n rows and m columns with predefined value types. Each row identifiable with a unique *primary key*. Such design make it possible to omit redundancy if constructed correctly.

There are several properties that make relational databases so powerful:

SQL or Search Query Language As you can see from the name, SQL is a query language and is based on relation algebra. The properties of relation algebra let the queries be nested and as complex as needed. In addition it has procedural properties, which make it possible to create, search and manage the data.

ACID Any sequence of database operation is called a transaction if it fulfills following principles:

- **A**tomicity transaction as a single unit, therefore if part of transaction fails, whole transaction should be reversed
- **C**onsistency transaction change the state of the database from one *valid* one to another *valid* one
- **I**solation if executed concurrently provide the same result if would executed sequentially
- **D**urability if transaction is committed, the state of database should remain in case of failure

In previous section we have described how the data model currently implemented in a relational database. Choosing more sophisticated relational database could lead to increase in performance. For example PostgreSQL or MySQL, which are widespread and under continuous development. Additionally it would be useful be adapt the schema of data model, so that hierarchical nature of VSL data can be better represented. In this book [4] author describes several techniques, that can be used when modeling a hierarchical structure:

- *Materialized Path* This technique is currently used to model a hierarchical structure in HSQLDB. "address" field in Structure table 2.2 acts both as identificator and path descriptor. Through string comparison we can retrieve all ancestors.
- *Adjacency List* In this technique direct parents and children of a node are stored in as separate columns. It is possible to use this technique in PostgreSQL as it provides recursive queries [5].
- *Nested Set* There are two columns, that "left" and "right". To assign values we have to traverse the tree starting a the root, and always turning left and incrementing the value, till you reach a leaf. In such fashion column "left" is filled. In the leaf we will increment the value and place it into "right" column. Then we will go one level up and take second most left node. A repeat the same process again. In the end each root of some sub-tree will have the smallest value in this sub-tree in "left" column and biggest in "right" column [6].
- *Closure Table* There is a column indicating whether some node is a descendent of current node, and the second column indicates on which level [7].

Key	Value
"TUM"	{"Sebastian", "Annie", "Florian", "Tobias"}
"LMU"	{"Daniela", "Jean", "Maria", "Dorian", "Franzis"}

Figure 2.2: Example of data stored in key-value store

2.3.2 NoSQL

The term NoSQL or *Not only SQL* was coined in 2009. At the moment relational databases were becoming not a perfect solution for data storage in several domains. Their lack of flexibility, rigid structure and limited ability to scale were major complains in the industry [3]. New types of databases were needed to implement more exotic types of data models. NoSQL became an umbrella term for such databases. In the following sections we will present several most important types of data storage paradigms.

2.3.2.1 Key-value stores

One of the most simplistic data storage paradigm is the Key-value store. It is basically a dictionary with set of unique keys and corresponding values, accessible through the keys.

We can see an example in figure 2.2. In this example we have a name of a university as a key, and student names as values.

Key-value stores are completely oblivious of the value data type and the value itself. Number of fields in each record can be arbitrary, therefore enabling flexible structure of data [8]. While key-value stores are oblivious of values, it is not possible to construct value-based queries. As a consequence key-value stores are not suitable for modeling complex structures, but perfect for large amount of simple data. Some of them are in-memory databases, with increased performance.

The most wide-spread examples are Redis, Memcached and Riak KV [9].

2.3.2.2 Document-oriented database

Document-oriented databases are similar to above mentioned key-value databases. They store document-like objects, that can be identified with an unique key. The most

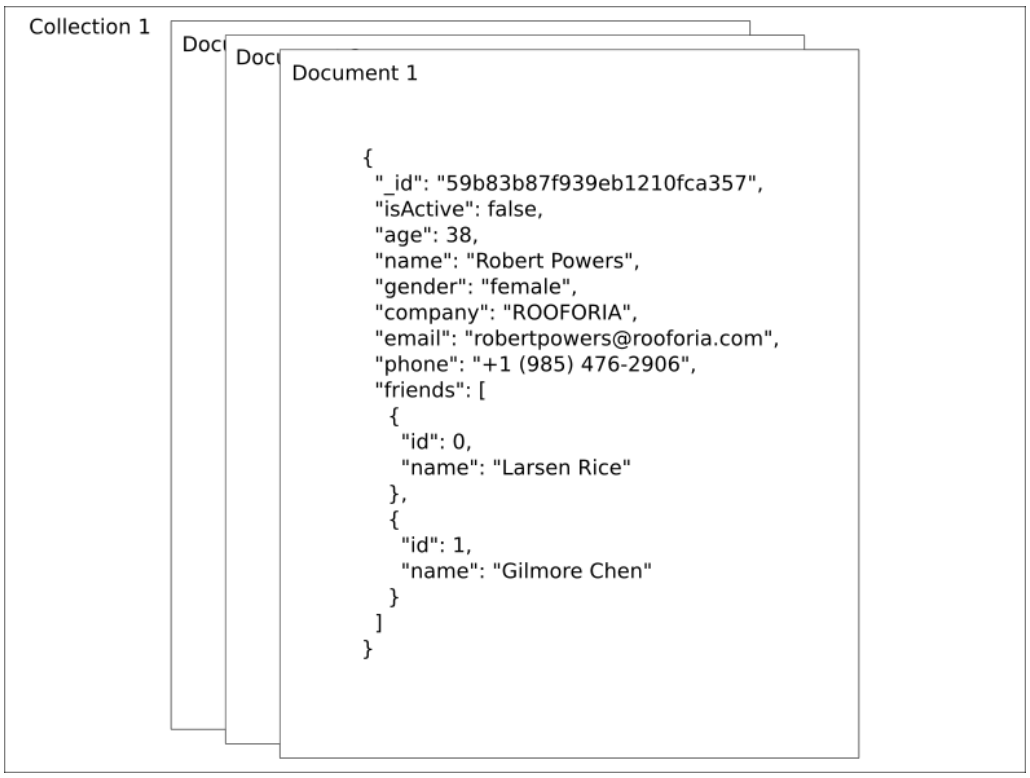

Figure 2.3: Example of data stored in document-oriented database

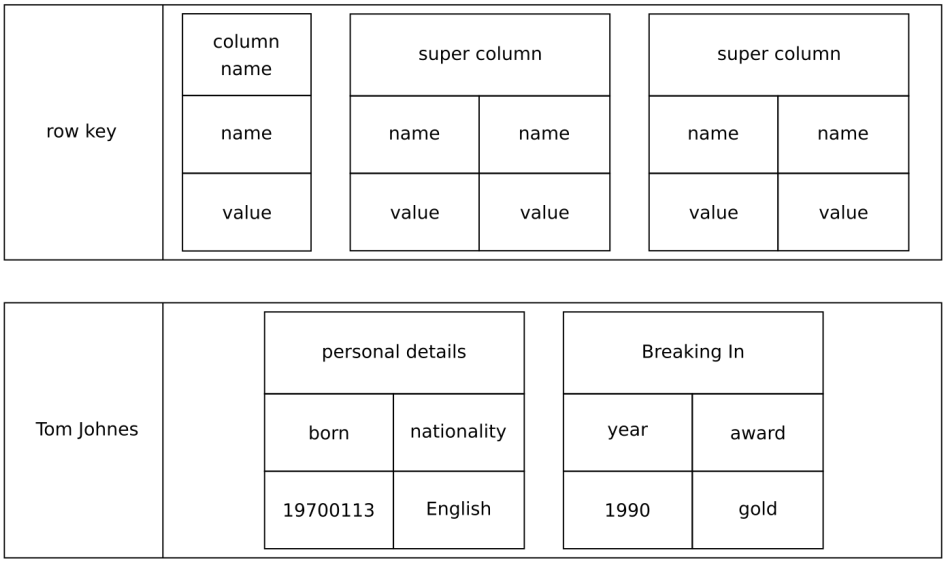

Figure 2.4: Example of data stored in column-oriented database

widespread formats of these objects are XML (eXtensible Markup Language), JSON (JavaScript Object Notation) or BSON (Binary JSON).

Documents, which are similar to records in relational databases stored within collections, that can be viewed as analogue to tables in relational databases. But in contrast to relational databases different documents can have variable number and type of fields. It increases flexibility of data modeling [8].

We can see an example in figure 2.3. A collection with number "1" consists of three documents, which can be identified with a unique "_id", and have a JSON structure.

In VSL we deal with hierachical data and official documentation of MongoDB provide several possible ways to store hierarchical data [10]. They are quite similar to the techniques introduced in section 2.3.1 and can be used in other document-oriented databases. Also in case of document-oriented databases it is possible to perform complex queries as the content of the document objects is not opaque to the database.

There is a number of such databases, which are under continuous development. The most popular are MongoDB, Amazon DynamoDB and CouchDB [11].

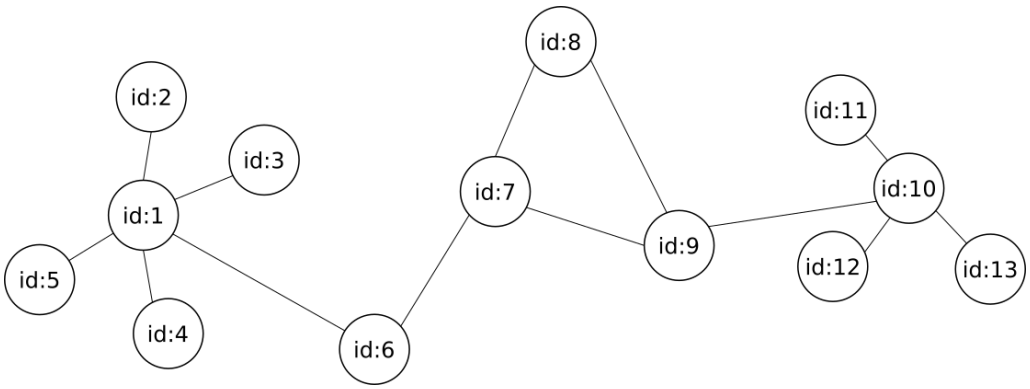

Figure 2.5: Example of data stored in graph database

2.3.2.3 Column-oriented database

First design and implementation of column-oriented databases was embodied in Google Bigtable. After publication of whitepapers several databases implementing similar approach emerged on the market.

The main structure is defined as follows: there are several rows, each can be identified with a *row key*. In every row there is variable number of column-families, which consist of columns and super column, which in their turn also consist of columns [8]. An example in figure 2.4 illustrate this concept.

Column-oriented databases are mostly used in read-intensive environment, where query and aggregation of specific parameter is needed. But in write-specific environment the benefits of column-oriented databases compared to relational databases are not significant. In case of modeling hierarchical data column-oriented databases are not expected to differ in terms of suitability from traditional relational databases.

The most notable examples of column-oriented databases are Google BigTable, Cassandra and HBase.

2.3.2.4 Graph-based database

Graph-based databases are designed for modeling relationships between objects. They consist of nodes and edges, which are the stored objects and their relationships to each.

Utilizing graph traversal algorithms, such as breadth-first traversal or depth-first traversal, it is possible to achieve much better performance than if the graph structure was placed in a standard relational database. Mostly used for storing and computing networks, such as social, transportation and etc. We can see an example in figure 2.5, where nodes represent entities with ids, and edges represent relations between these entities [8].

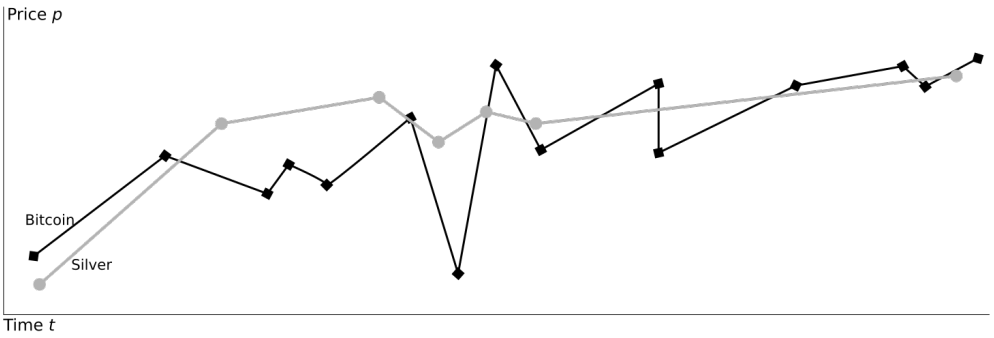

Figure 2.6: Example of graph based on data stored in time series database

As stated in section 2.1.1.2 data of the context node is stored in tree-like structure. And tree-like structure is essentially a constrained graph. Therefore graph-based database could be quite performant when an update of parameters in a whole node is needed.

The most used examples are Neo4j and OrientDB.

2.3.2.5 Time series database

Time series are specifically tailored to process and store an arbitrary amount of time-series data. Time series are build from a key, timestamps and values, at the given time points.

Such databases are best suited for statistical data, for example sensor values or price changes. We can see an visual example of such data in a figure 2.6. In this figure we can see how the price of silver and bitcoin changes across a time span. Some of time series databases provide build in functions for statistical analysis of stored data.

But modeling a complex hierarchical structure in such databases is expected to be difficult. The most wide-spread time series databases are InfluxDB, Graphite and Riak TS.

2.3.2.6 Multi-model database

Multi-model databases usually combine graph-based approach with relational, document-oriented and key/value models. This approach significantly simplifies modeling of complex structures, as they become easier to fit in a database.

An example of graph, containing documents within nodes, can be seen in figure 2.7. Most advanced examples of such databases provide a common query language to all types of data stored [12]. As multi-model databases usually include both graph and

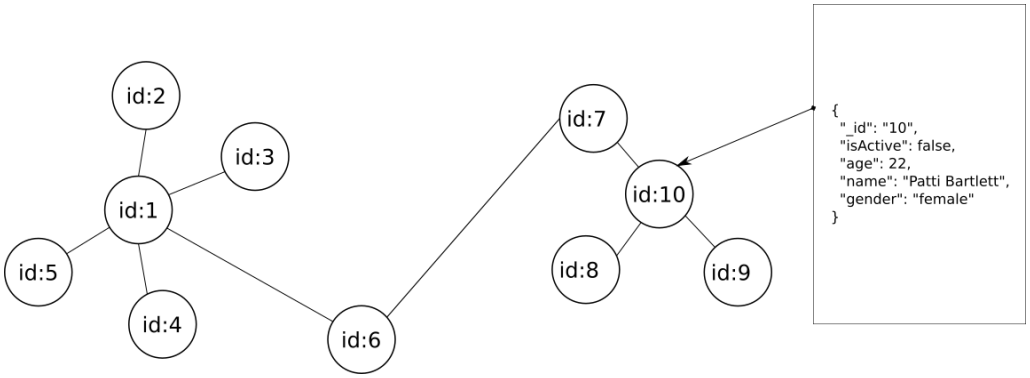

Figure 2.7: Example of data stored in multi-model database

documents as data models, they should be considered as one of the most suitable for storing tree-like structures.

The most known examples are OrientDB and ArangoDB.

2.4 Requirements

There are a number of requirements imposed on a database:

(i) **Ability to store context node data**
Chosen database should be able to store context node data, that has a tree-like structure (see 2.1.1.2). Depending on type of a database data model will be adapted.

(ii) **Java API**
As currently DS2OS is implemented in Java, it will be beneficial to have native compatibility or JDBS Driver (see 2.1).

(iii) **Versioning**
At the moment versioning is done application level of DS2OS. It handled within a database wrapper class, therefore having a native support of versioning inside of database will simplify data model. Additionally it will not require changes to the structure of VSL outside of database wrapper (see 2.1.1.2).

(iv) **Archive**
Previous states of context node should be stored for predefined amount of time or until they reach a certain threshold. While in an archive they should be available for query, either as single entity or as a set (see 2.1.1.2).

(v) **Consistency**
In case of separation of nodes data and structural information of the tree, consistency issue should be considered.

Chapter 3

Related Work

In this chapter we present previous research on comparison of database performance. We have chosen research material, where different database architectures were compared to each other in terms of performance. Several presented papers use specifically IoT data [13], [14], [15]. Other have performed the evaluations using arbitrary data.

In the second half we discuss discovered methods of handling versioning and tree-structures in document-oriented database.

3.1 Current research on comparison of SQL and NoSQL databases

3.1.1 Compared databases: MySQL, MongoDB, CouchDB, Redis

In following study several experiments were conducted, to evaluate the performance of different database types [13]. Researchers have chosen several databases: MySQL as relational database, MongoDB and CouchDB as document-oriented databases, and Redis as key/value storage. As testing material there were two types of data: sensor data and large multimedia data. For us the results of experiments with sensor data are more relevant, while DS2OS mostly works with similar data.

Authors of this paper have found that in write-intensive environment MongoDB has outperformed other databases. It was followed by relational database, namely MySQL. In experiment consisting of querying the data the was almost no significant differences. Also it was worth mentioning that CouchDB was lagging in terms of performance and had unjustified large database size.

3.1.2 Compared databases: PostegreSQL, Cassandra, MongoDB

Comparison of PostegreSQL, Cassandra and MongoDB using sensor data was made in this paper [15]. Researchers have used four types of operations: a single write, a single read and multiple read and writes. In single operations a measurement consisting of a value, timestamp and unique is either read or written to the database. In case of multiple operation 1000 of measurements are written or read in a single statement. During evaluations, researchers have tracked how many operations can each database perform in one second. The results were inconclusive, as databases performed different depending on the task. In case of single writes MongoDB outperformed all other competitors. This finding also correspond with above mentioned suitability of MongoDB in write-intensive environment. In case of multiple reads PostgreSQL has performed better than any other. Cassandra was better in case of multi-write. These results shows us that minor tweaks and differences in the tasks and environment can impact performance of the database.

3.1.3 Compared databases: MongoDB, RavenDB, CouchDB, Cassandra, Hypertable, Couchbase, MS SQL Express

Researchers in this study, compared following databases: MongoDB, RavenDB, CouchDB, Cassandra, Hypertable, Couchbase and MS SQL Express [2]. As a data set a number of artificially generated key/value pairs was used. Following operations were executed: writing, reading, deleting a key/value pair, and fetching all keys. In each test case only one type of the operation was used. Researchers incrementally increased number of performed operations, and recorded needed time for performing them. They have performed 10, 50, 100, 1000, 10000 and 100000 operations. In results they have found that RavenDB and CouchDB were lacking performance in most tasks. Couchbase and MongoDB were two fastest databases in read, write and delete operations. This study gives us a quite simplified overview of database performances, because performed operations and chosen data are not complex.

3.1.4 Compared databases: PostgreSQL, MongoDB

In following study researchers compared PostgreSQL and MongoDB on all basic operations, such as insert, select, update, and delete [16]. Data was designed separately. For PostgreSQL three simple schemas, related to each other, were designed. For MongoDB all the data was designed as one single document. Each operation was executed following number of times: 30000, 90000, 150000, 210000 and 300000. Overall MongoDB has performed better than PostgreSQL, but we have to take data differences into considerations. The differences were more pronounced during insert and delete operations. It is possible that results of this study are biased toward MongoDB.

3.1.5 Compared databases: MySQL, MongoDB, VoltDB

In this study researchers have compared MySQL, MongoDB and VoltDB using also sensor data [14]. During evaluation they varied following parameters: single or multi client, and single or multi operations. As operation were used basic read, write and delete. They have come to a conclusion that VoltDB, which is a newSQL database, has outperformed its competitors in most tasks. MongoDB was the second best, with small difference in performance.

3.1.6 Summary

Differences in database performance are depended on the structure of the data and the type of operations that are performed on them. NoSQL databases do not perform significantly better than advanced relational databases, such as PostgreSQL. But they provide other benefits that can be useful for us, such as a more flexible data model. We can not draw concrete conclusions based on the found studies. First we have to implement several databases, according to requirements (see 2.4) to further investigate the the suitability of databases for IoT, and more specifically for DS2OS.

3.2 Versioning

As stated in section 2.1.1.3, when context node is changed, a new node, with incremented version number and changed value is inserted in a database. Additionally all ancestors of this node also receive an increment of their version. Currently a separate table in relational database is used to keep track of versions of the node.

3.2.1 Versioning in document-oriented database

In this article author presents and evaluates possible approaches for storing different versions of a document in MongoDB [17]. These approaches can be extrapolated on other document-oriented databases, as the techniques are not MongoDB specific. According to the author the techniques are following:

- To create new document for every version with incrementing version field. It is quite simple approach, but it has several drawbacks. In case of need to query a current version it will perform rather slowly. A possible solution would be to have a field "current" set in most recent document. But it would be problematic to use in multi-threaded environment, as in case of update we need to set the field to "false" in current document and the add the latest version with the same field set to "true". Failure in operation flow could result in several "current" versions.

Schema	Fetch 1	Fetch Many	Update	Recover if fail
New doc for each	Easy,Fast	Not easy,Slow	Medium	N/A
New doc with “current”	Easy,Fast	Easy,Fast	Medium	Hard
Embedded in single doc	Easy,Fastest	Easy,Fastest	Medium	N/A
Sep Collection for prev.	Easy,Fastest	Easy,Fastest	Medium	Medium Hard
Deltas only in new doc	Hard,Slow	Hard,Slow	Medium	N/A

Table 3.1: Versioning in MongoDB

- To store all the versions embedded in one document. The advantage of this technique is that we always know what is the current version, and non-current versions can also be retrieved easily. But nesting all the versions in the same document can enlarge it to unwanted sizes. For example MongoDB have a threshold of 16 Mb pro document.
- To store current version in main collection, and all previous versions in separate collection. To prevent unwanted states it is important to copy the current version to "previous" collection before the update. That way in case of failure it is not possible to lose the version. [18] Although it is possible to have an duplicate of the latest document version in "previous" collection, it can be handled on the application level.
- To store deltas of each version in separate documents. It is a resource saving technique, but disadvantages are more pronounced then in other approaches. It is quite difficult to fetch multiple versions of the same document, because each of them have to be constructed on the application level.

The author have considered these approaches under two main scenarios: ether query of a current version or query a number of previous versions. The findings are summarized in table 3.1.

3.3 Tree-structures

3.3.1 Storage model of tree-structure in MongoDB

Authors of following study implemented and evaluated four different ways of storing tree-structure in MongoDB [19]. The approaches were following: Adjacency List Model, Children Reference Model, Nested Document Model and their own design Hybrid Storage Model.

In Hybrid Storage Model they kept each node of a tree in a separate document containing all attributes of a node, and hash value of them as an unique ID. The tree-structure itself was modeled as a separate document of nested hash values of the nodes.

Authors have evaluated performance of the examined approaches using two different types of test cases. In the first they used queries based on tree's attributes. In such case reconstruction of the whole tree is needed. Therefore as expected Nested Document Model, followed by Hybrid Model have performed significantly better than Children Reference Model and Adjacency List Model.

In second case queries on node's attribute have been used. They have constructed in such way, that at first node's with corresponding were found, and then the whole tree containing this node was reconstructed. This test partially replicate the first one, leading to the similar results. Besides Nested Document was not correctly included in evaluation.

Even through this study suggests usage of separate documents for values of the nodes and tree-structure, such approach can lead to inconsistency in case of write-operations.

3.3.2 Comparison of NoSQL databases in handling tree-like structure

An important and highly related comparisons were conducted in following study [20]. Researchers have implemented a highly heterogeneous tree-structure in different types of databases.

Representatives were following:

key-value database : Membase

document oriented database : MongoDB

graph database : Neo4j

wide-column database : Apache Cassandra

multi-model database : OrientDB

They have conducted 4 different tests: inserting a tree with a 100 000 nodes, retrieval of a sub-tree, querying a node based on attribute containment, querying a value of an attribute. Even through Membase outperformed its competitors on the tree creation, authors state that it is not suited for tree-structures. Same is said for Apache Cassandra, it was least performant in 3 of 4 executed tests.

On the other hand Neo4j, as a graph database is naturally suited for a tree-structures. But lack of performance on creation of a tree was quite significant. Document oriented MongoDB was behind other databases on sub-tree retrieval. On average multi-model database OrientDB performed well in all of 4 task, proving to be a well-rounded solution.

Chapter 4

Design

In first part of this chapter we present and discuss our approach to construction of evaluation scenarios for Virtual State Layer database backend. In the second part we discuss different database types and their applicability for Virtual State Layer. Following the discussion we present databases, that we choose for further implementation and evaluation. We describe our approach to handling tree-like structure of VSL data, and handling the archive of Context Nodes (see 2.4).

4.1 Design of evaluation scenarios

In section 2.2 we have presented possible parameters which can be altered to create different evaluation scenarios. Our main goal is to cover most possible usage patterns of DS2OS. In the following subsections we construct scenarios, varying previously mentioned parameters, and describe them in more detail.

One of the parameters that we can alter is the structure of the tree. A "wide" tree can be constructed with small number of levels and large amount of subtrees below the root node. On figure 4.1 we can see an example of such tree, with three levels and most of

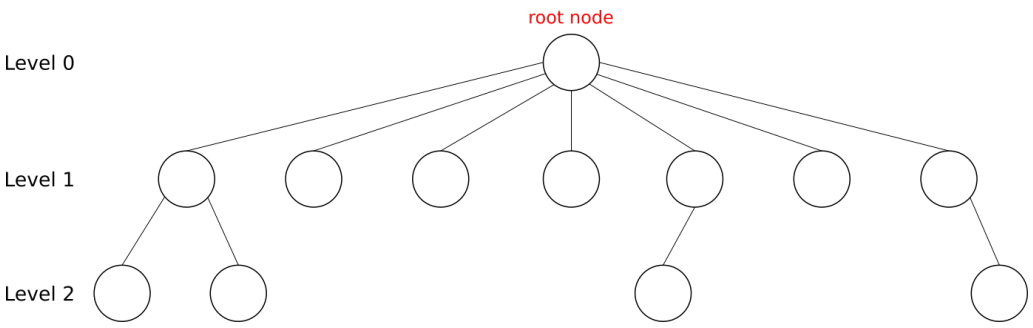

Figure 4.1: Wide tree

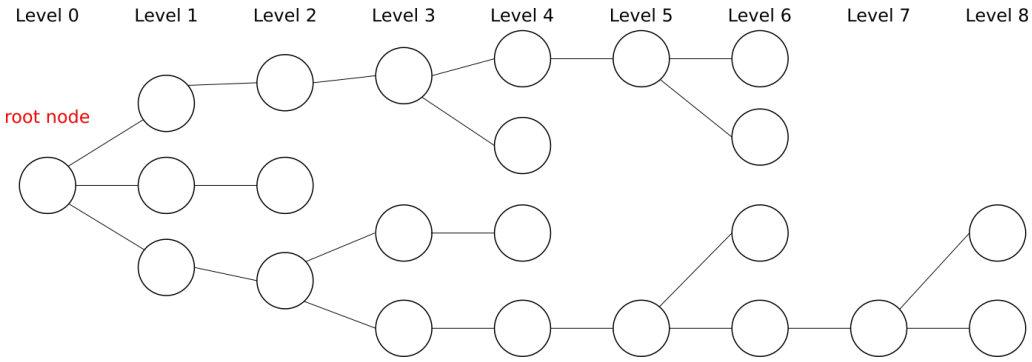

Figure 4.2: Deep tree

the nodes concentrated on the second level. Such tree can represent an IoT device that has multiple number of simple and similar elements. For example a large LCD panel that is used in an art exhibition hall.

On the other hand we can construct a so called "deep" tree. As can be seen on figure 4.2, it consists of high number of levels with a small number of nodes on each level. Such tree can represent a complex device with multiple parts within.

We assume that differences in tree-structure can affect performance of the database. Because for example in case of "deep" tree on update of leaf node, we have to update the state of all ancestor nodes from leaf up to the node, that represent "service" level (see 2.1.1.2). For the purpose of this thesis we construct "wide tree" with 3 levels, consisting of 1 root node, and 10 nodes on each of the subsequent levels. An example of a 'deep tree' we construct with 9 levels: 1 root node, 3 nodes on the second level, 1 node on next 6 levels, and then 3 leaf nodes (see A.3). The choice of numbers is arbitrary.

To evaluate performance of selected databases we inspect how they behave under different workload. We expect that in the real-world environment operations, performed on context node, will be varied in their type. Depending on the services, which use this context node, the majority of operations can be either 'write' or 'read'. To address possible differences in database performance, we constructed tests in the following way.

Each test consists two types of operations, mentioned in section 2.2: *get* and *set*. Whole number of operations executed in a test is *100*, *1000*, *10 000*. Furthermore we vary the proportions of the operations in a test. We define three possible variations:

- 100% *get* / 0% *set*
  This proportion can be seen as edge case, that is not common in real-world environment. But it is useful for evaluation of performance on pure "write" operations.
- 80% *get* / 20% *set*
  This variation simulates a read-intensive environment, where information is

mostly read than written.

- 50% *get* / 50% *set*
  This can be seen as balanced environment, where the proportions of executed operations are equal.
- 20% *get* / 80% *set*
  This will simulate a write-intensive environment, where services frequently set new values in the context node.
- 0% *get* / 100% *set*
  This also is an edge case. Through this proportion we want to evaluate pure "read" performance.

Each test is performed at least five times. We record time needed to perform defined number of operation, and create an average of five executions for each database. We do this to reduce possibility, that results of single evaluation do not represent actual possible performance.

To execute constructed scenarios we create a DS2OS service that communicate with a single Knowledge Agent. This Knowledge Agent has a database connected locally, therefore we don't take network constrains into consideration.

4.2 Suitability of different database types for VSL

We have presented several types of databases in section 2.3 and described main concepts on which they are based. In this section we discuss their suitability for Virtual State Layer, considering its requirements.

As mentioned in section 2.1.1.3 HSQLDB is currently used as database backend for Virtual State Layer. The choice of database was arbitrary during development of the first functioning prototype of Virtual State Layer.

It is possible that a more advanced relational database can deliver better performance. In studies that we presented in chapter 3 PostgreSQL in most cases perform on the same level or even faster that NoSQL databases. Due to time limitation we don't implement more advanced relational database for Virtual State Layer in this thesis. But it can be considered a possible candidate for further research.

Key-value stores are considered as not desired option. Their inability to query objects, based on internal attributes, was a main disadvantage for us. Services can query specific Context Nodes from Knowledge Agents, based on characteristics such as type, timestamp or version. If we would store context nodes data in key-value database, it would be only possible to query it by the "address" of the node, which would be the main key. To leverage this constrain we would be forced to split the data of context node and

store it behind several keys. This is not a desired solution, because it would introduce unnecessary possibility of inconsistency during write operations.

Document-oriented databases are considered a much better fit for our needs. They also have flexible data model, which can be used to represent structures with varying degrees of complexity. For example there are multiple ways to model a tree-structure, some of them are presented in section 3.3.1. Also ability to filter documents based on values of the fields is an important feature for us. As mentioned above services may query Context Nodes, based on several parameters combined.

Graph databases are also a valid option, considering our needs. As they are able to model tree-structures natively. Therefore we have to take care of versioning and keeping the archive. Also some of them provide useful graphical frontend, where we can observe current state of the context node graph. But as described in the following study [20], Neo4j, one of the most popular graph databases, performs significantly slower than other databases on node creation. Authors speculate, that the reason behind it, is creation of metadata overhead regarding graph structure. It is possible that this overhead enables a more fast traversal of the graph. But for us this is an undesirable trait, because node creation is one the core operations in Virtual State Layer. And in most cases we don't traverse the context tree, except for operations with node version update and retrieval of subtree.

In the same study [20] OrientDB, multi-model database, was evaluated alongside with other databases. During evaluation it showed adequate performance in all tasks, without significant unbalance between data creation or data retrieval operations. Same as Neo4j, OrientDB has an ability to natively handle graph-structures. Therefore we decided to implement OrientDB for further evaluation.

4.2.1 Document-oriented database

Based on studies presented in chapter 3 we decide to use MongoDB as an example of document-oriented database. MongoDB has adequate performance, is one the most popular document-oriented databases, is under continuing development, and has extensive documentation [21].

To represent nodes of the Context Tree we use separate document for each node. The structure of the document is presented in the table 4.1. In this table you can see fields of the document and corresponding types, which are MongoDB internal. "id" field is a 12 byte hexadecimal number, that is used for internal identification of the node within the collection. Most of other fields, excluding "parent" and "ancestors", correspond to the general model specified in table 2.1. We will explain the choice of such document structure in the following paragraphs.

According to the section on tree-structure modeling in the official documentation, authors suggest using "Array of Ancestors" technique for working with subtrees [10]. In this technique, addresses of all node's ancestors are stored as an array within a node itself. It is beneficial for us, because, as stated in section 3.2 after adding a new version of node, all ancestors of this node should increment their version as well. Also, to differentiate between direct and indirect ancestors, we added a field "parent", that contains an address of node's direct parent. It was beneficial for us, because depending on the use case, it is necessary retrieve or alter only direct children of a node. This approach similar to other techniques, such as "Child References", "Parent References" [10].

But using them without in their pure form would require multiple fetches of related nodes, to update all of the node ancestors. It is undesirable for us, because it increases amount of interaction between database backend and Knowledge Agent.

The technique, named "Materialized Path", could be applied in our implementation, considering format of VSL Node address. But it would still require a construction of an array of ancestors on application level. Additionally we don't use the technique presented in section 3.3.1, where structure of the node-tree is separated from the values within the nodes. The reason for that, is to reduce the probability of inconsistency within the system. MongoDB does not provide consistency on write operations that affect multiple documents [21]. Therefore if we store the structure of the Context Tree in one document, and values of the nodes in the other document, then during creation or removal on nodes in the Context Tree it can lead to inconsistent state, where either there is a node with missing value, or a value that does not correspond to any of the nodes.

To store an archive of previous states of the node, we use one the techniques presented in section 3.2.1. We store documents, representing the current state of each node in one collection and documents, that represent all previous states in another. On update, document, representing current state of the node, is fetched from "current" collection and copied in the "archive" collection first. Then it is modified and written back into the "current" collection. This also reduces probability of inconsistency. Because even if some failure occur during update, then document, representing previous state of a node, will be both in "archive" and "current" collection. This kind of malfunction can be reversed by using upsert in "archive" collection. In current use cases of DS2OS queries on current state of a node occur more often, then on archived versions of nodes. Therefore storing current state of a node within a separate collection is expected to have positive impact on the performance. Within "current" collection documents are uniquely indexed on "address" field, therefore not allowing duplicates of the same node to exist. And in "archive" collection documents are indexed on both "address" and "version" field, therefore enabling storage of multiple states of the node. As mentioned before "id" field is used in MongoDB to uniquely identify the documents within the collection. There is an option to prevent it, which we enable for "archive" collection.

We did not use other techniques based on following considerations. Storing nodes as separate documents with incrementing version field within single collection will reduce performance on query operations. We would be forced to use the same approach, that we use currently with "archive" collection, namely filtering and aggregation pipeline. But as mentioned before we tolerate the decrease in performance tied with this approach, only because querying of archived states of the node happens infrequently in current use cases of DS2OS. Storing only deltas of changes was considered the most unsuitable approach, because it would have require a reconstruction of the node, on every read operation. Storing a map with "version" as a key and "value" of the node as a value could have been a valid approach. But MongoDB has a limit on size of the document, 16 megabytes [21]. Therefore without limit of stored states on the application level, this approach can not be used.

id	objectId
address	string
parent	string
ancestors	array
value	string
version	long
types	array
readerIDs	array
writerIDs	array
timestamp	timestamp
restriction	string
cacheParameters	string

Table 4.1: MongoDB Object structure

4.2.2 Multi-model database

Based on studies in chapter 3 and discussion in section 4.2 we choose to implement OrientDB as an example of multi-model/graph database.

OrientDB is written in Java, therefore provides an native Java API for most operations. It is beneficial to us, because as mentioned in section 2.4, DS2OS is currently implemented using Java. It has SQL-like query language and can be used in both transaction-full and transaction-less mode. OrientDB supports several database models, such as Graph, Document, Key/Value and Object [12]. It has several concepts of data modeling, that need to described. In OrientDB data is generally divided into "clusters", "classes" and "records".

Record is the smallest unit of data in OrientDB. Depending on the chosen database model it can be "document", "vertex" and "edge". "Documents" are used in such models

as Document, Key/Value and Object model. If we use graph model than all records are automatically seen either as "Vertexes" or "Edges". Records have a specific "class", that holds information about the structure of records.

Class is a concept, that is derived form Object-Oriented Programming. OrientDB can store records in a schema-less or schema-full way. Classes are used to specify types of fields in records and enforce constrains on these fields if needed [12].

Clusters, can be seen the same as "tables" in relational database or "collections" in document-oriented database. It is a way to group records based on their characteristics. By default in OrientDB for each "class" a new cluster is created. Then a user can add additional clusters and move the records across them. When using OrientDB as a distributed database, clusters make it possible shard data across multiple machines [12].

For the purpose of this thesis we use Graph model in OrientDB database. Also for all operations, that change the Context Tree, we use transaction-full mode of operation with database. We are able to store a Context Tree without any alterations in its structure. Each node of Context Tree is represented as "vertex" in database. It has incoming and outgoing "edges", that represent hierarchical relations. We declare a custom a class for our nodes, called "vslNode", which is subclass of general vertex class "V". Within class we define the types of the fields, that a node can have. We use classes to ensure, that each created node correctly structured and filled with information, according to specified schema.

An example of structure of such node can be seen in table 4.2. @rid, @class are OrientDB internal fields, the first indicates the unique identificator of the record within the database. The second one identifies a class of the record in OrientDB. All other fields correspond to the general model of Context Node specified in table 2.1.

Our first approach for archiving the nodes was to use the same technique, as with MongoDB. We would have create two distinct clusters: "current" and "archive". And store the most recent of state the Context Tree in the "current" cluster, and all previous states in "archive" cluster. But during implementation this strategy have proved itself to be inefficient. Starting with version 2.2 OrientDB creates several clusters for every class of records. Number of created clusters corresponds to number of CPU cores on the server, that runs the database. Authors claim that this approach improves usage of parallelism [12]. Therefore artificially limiting the number of clusters for "vslNode" records may have impact on performance. Additionally during update operation of Context Node we would have to either recreate a whole affected subtree in "archive" cluster or copy each affected vertex separately without saving edges, therefore the relations between them. In first case "archive" cluster would be populated with multiple small subtrees, which do not have relations with nodes, that we unaffected on n-th update. In second case we loose the relations completely, therefore losing the benefits of "Graph-database". We would have had a cluster, with multiple "document-like" records,

similar to MongoDB. Taking previously mentioned information into consideration, we decided to try another approach.

Out second approach is similar to one of the techniques mentioned in section 3.2.1. We store all versions of a node within one record. For that we add a field that contains a map object, where "version" is the key and "value" of the node is the value in map. But we don't remove the fields "version", "value" from the "vslNode" structure. These fields are used to store the most recent state and version of the node. Therefore we can access most recent state easily, without the need to iterate over entries in the map. The records in OrientDB can reach up to 2GB in size [12]. We assume that after a certain point, storing all previous states within the node itself will hinder the performance. To elevate this constrain a limit of number of states should be introduced on application level. So that after reaching the certain threshold a part of previous states will be deleted from the node. At the moment this feature is not implemented.

As mentioned before developer can interact with OrientDB using either native Java API or issuing SQL-like queries. We decided to use SQL-like queries, to make it easier to maintain the implementation for other developers, as most of them are at least familiar with SQL queries. Also documentation for SQL command is bigger and more verbose for OrientDB [12]. Therefore it is easier to predict the behavior of database, caused by performed operation.

@rid	Custom
@class	String
address	String
value	String
version	Long
versions	Embedded map
types	Embedded list
readerIDs	Embedded list
writerIDs	Embedded list
restriction	String
cacheParameters	String
timestamp	Date

Table 4.2: OrientDB Object structure

Chapter 5

Implementation

In this chapter we present selected parts of our implementation of databases mentioned in chapter 4. Also we describe the implementation of service, that we use for performance evaluation.

At the moment DS2OS, and its parts, such as Virtual State Layer and a number of services are implemented using Java 7. Therefore we also use language constructs that are compatible with Java 7 in our implementation.

5.1 MongoDB

This section gives an overview of how we implemented MongoDB for Virtual State Layer. We describe the most complex implemented methods of **VslDatabaseInterface**.

For **MongoDB** we use version **3.4.10**. To communicate with MongoDB we use native MongoDB Java Driver, version **3.5.0**. At the moment these are the most recent versions of this software.

In this listing 5.1 you can see a source code of implementation of *SetValueTree()* method. This method is used to set new values to multiple nodes within a Context Tree. As parameter we give him a map containing addresses of nodes and corresponding values. During execution we first gather all nodes, that will need a version update. As mentioned in section 2.1.1.3 if we set a value of node, all its ancestors up to the service level should also increments their version. Moreover if several nodes, that are about to receive a new value, share the same ancestor, then version of an ancestor node should be only incremented once. As a second step we copy the nodes, that change their value, into "archive" collection, and update the nodes in current collection. In the second *for* loop we do the same for nodes, that don't receive a value update, but change their version indirectly.

```
public void setValueTree(Map<String, String> values) throws NodeNotExistingException {
   if (values.size() == 0) {
     return;
   }
   final List<String> affectedNodes = new LinkedList<String>();
   // gather all affected nodes. Every node will be only counted once, even if he
   // is the parent of more then one changed node.
   for (final String address : values.keySet()) {
     if (!affectedNodes.contains(address)) {
       affectedNodes.add(address);
     }
     for (final String parent : AddressParser.getAllParentsOfAddress(address, 2)) {
       if (!affectedNodes.contains(parent)) {
         affectedNodes.add(parent);
       }
     }
   }
   affectedNodes.removeAll(values.keySet());

   // Iterating over valueTree, archiving the old nodes,
   // setting new values and incrementing versions of affected nodes.
   for (final Entry<String, String> entry : values.entrySet()) {
     Document currentDocument = currentCollection.find(eq("address", entry.getKey())).
          first();
     archiveCollection.insertOne(currentDocument);

     currentCollection.updateOne(eq("address", entry.getKey()),
        combine(set("value", entry.getValue()),
           set("timestamp", new Date()),
           inc("version", 1)));
   }
   // incrementing version of ancestor nodes
   for (final String entry : affectedNodes) {
     Document currentDocument = currentCollection.find(eq("address", entry)).first();
     archiveCollection.insertOne(currentDocument);
     currentCollection.updateOne(eq("address", entry), inc("version", 1));
   }
 }
```

Listing 5.1: Implementation of SetValueTree() method

In this listing 5.2 we present a source code on the second most important method of **VslDatabaseInterface**. *getNodeRecord()* takes two input values:

- Node address as a *String*, that represents the root of the subtree that will be processed in the method.
- Parameters object of the internal type *VslAddressParameters*, which can be extended during development of Virtual State Layer. At the moment we can specify: depth of requested subtree, version of the requested nodes in the subtree and

whether or not, we want complete information about the node, only values or only metadata, such as types, restrictions, access parameters and etc.

To filter out the documents based on the parameter, that are given, we use aggregation pipeline. Aggregation pipeline is a concept, where documents go through several data processing stages, which transform them into desired results [22]. During execution of the method we construct several filtering stages, with help of these methods (listing A.1 and listing A.2, which can be examined in chapter A. Through usage of aggragation pipeline and helper methods we want to ensure, that in case of further change and extension of query parameters, we can easily adapt *getNodeRecord()* method, without the need to rewriting from scratch.

```
1  public TreeMap<String, InternalNode> getNodeRecord(String address, VslAddressParameters
       params)
2      throws NodeNotExistingException {
3    TreeMap<String, InternalNode> results = new TreeMap<String, InternalNode>();
4    MongoCollection<Document> chosenCollection = currentCollection;
5    List<Bson> appliedFilters = new ArrayList<Bson>();
6    if (address == null) {
7      throw new NodeNotExistingException("address was null");
8    }
9    int requestedVersion = params.getRequestedVersion();
10   int requestedDepth = params.getDepth();
11   if (requestedVersion != -1) {
12     chosenCollection = archiveCollection;
13     filterRequestedVersion(appliedFilters, requestedVersion);
14   }
15   Document rootNode = chosenCollection.find(eq("address", address)).first();
16   if (rootNode == null) {
17     throw new NodeNotExistingException("node does not exist");
18   }
19   filterRequestedDepth(appliedFilters, requestedDepth, address);
20   MongoCursor<Document> cursor = chosenCollection.aggregate(appliedFilters).iterator();
21   while (cursor.hasNext()) {
22     Document currentDocument = cursor.next();
23     results.put(currentDocument.getString("address"), constructInternalNode(
           currentDocument, params));
24   }
25   if (results.isEmpty()) {
26     throw new NodeNotExistingException("Node not found: " + address);
27   }
28   return results;
29 }
```

Listing 5.2: Implementation of getNodeRecord() method

5.2 OrientDB

In section we present selected parts of implementation of OrientDB for Virtual State Layer. During implementation we have used community edition, version **2.2.29**, the latest version of OrientDB available at the moment.

In the listing 5.3 you can see a method, that queries the record of a Context Node from OrientDB. As mentioned in section 4.2.2 for operations that don't change the data within database we use transaction-less connection. In the method we first create a connection, then using **TRAVERSE** query, start to collect all requested nodes. The **MAXDEPTH** parameter specifies how far we travel from the original node. It depends on values within of VslAddressParameters object. We either collect only the node requested on specified address, it and its direct children or we travel all the way down in a Context Tree. We construct final results in a separate method, where depending on values of VslAddressParameters object, specific version of a Context Node can be requested, or wether we need whole node, or only relevant data, such as value.

```
1  public TreeMap<String, InternalNode> getNodeRecord(String address, VslAddressParameters
       params) throws NodeNotExistingException {
2    TreeMap<String, InternalNode> results = new TreeMap<String, InternalNode>();
3    int requestedDepth = params.getDepth();
4    if (address == null) {
5      throw new NodeNotExistingException("address was null");
6    }
7    OrientGraphNoTx graph = graphFactory.getNoTx();
8    try {
9      Iterable<Vertex> vertices = null;
10     switch (requestedDepth) {
11     case -1:
12       vertices = graph.command(new OCommandSQL(
13          "SELECT FROM (TRAVERSE in() FROM (SELECT FROM vslNode WHERE address = ?)
               STRATEGY DEPTH_FIRST)"))
14          .execute(address);
15       break;
16     case 0:
17       vertices = graph.command(new OCommandSQL(
18          "SELECT FROM (TRAVERSE in() FROM (SELECT FROM vslNode WHERE address = ?)
               MAXDEPTH 0 STRATEGY DEPTH_FIRST)"))
19          .execute(address);
20       break;
21     default:
22       vertices = graph.command(new OCommandSQL(
23          "SELECT FROM (TRAVERSE in() FROM (SELECT FROM vslNode WHERE address = ?)
               MAXDEPTH 1 STRATEGY DEPTH_FIRST)"))
24          .execute(address);
25     }
26     if (vertices != null) {
27       Iterator<Vertex> vslNodes = vertices.iterator();
28       while (vslNodes.hasNext()) {
```

```
29          Vertex currentNode = vslNodes.next();
30          results.put((String) currentNode.getProperty("address"), constructInternalNode(
                currentNode, params));
31        }
32      }
33    } catch (Exception e) {
34      System.out.println(e.getMessage());
35    }
36    if (results.isEmpty()) {
37      throw new NodeNotExistingException("Node not found: " + address);
38    }
39    return results;
40  }
```

Listing 5.3: Implementation of getNodeRecord() method in OrientDB

In the listing 5.4 we present our implementation of *setValueTreeMethod()* for OrientDB. We use the same approach to collect all affected nodes, that will require a version update. Then we initialize a transaction, where we do following. For all affected nodes, we query them from the database, check whether they require a value update. If they do, then we copy current version alongside with current into map object, that is stored in the node under field "archive". Then we update the value and increment the version. If they do not require value update, then we still copy current version and value in the "archive" map object, and then only increment the version, keeping the value as it is. After all operations are performed we commit the new state to the database. If not, then all changes are reverted.

```
1  public void setValueTree(Map<String, String> values) throws NodeNotExistingException {
2    final List<String> affectedNodes = new LinkedList<String>();
3    // gather all affected nodes. Every node will be only counted once, even if he
4    // is the parent of more then one changed node.
5    for (final String address : values.keySet()) {
6      if (!affectedNodes.contains(address)) {
7        affectedNodes.add(address);
8      }
9      for (final String parent : AddressParser.getAllParentsOfAddress(address, 2)) {
10       if (!affectedNodes.contains(parent)) {
11         affectedNodes.add(parent);
12       }
13     }
14   }
15   System.out.println(affectedNodes.toString());
16   OrientGraph graph = graphFactory.getTx();
17   try {
18     for (String address : affectedNodes) {
19       Iterable<Vertex> query = graph.command(new OCommandSQL("SELECT FROM vslNode WHERE
             address = ?")).execute(address);
20       Vertex currentNode = query.iterator().next();
21       Map<Long, String> archive = currentNode.getProperty("archive");
22       if (values.containsKey(address)) {
```

```
23        String valueToSet = values.get(address);
24        long currentVersion = currentNode.getProperty("version");
25        currentVersion++;
26        archive.put(currentVersion, valueToSet);
27        graph.command(new OCommandSQL("UPDATE vslNode SET value = ?, version = ?, archive
              = ? WHERE address = ?"))
28                .execute(valueToSet, currentVersion, archive, address);
29        graph.command(new OCommandSQL("UPDATE vslNode SET timestamp = ? WHERE address = ?"
              )).execute(new Date(), address);
30      } else {
31        long currentVersion = currentNode.getProperty("version");
32        currentVersion++;
33        String currentValue = currentNode.getProperty("value");
34        archive.put(currentVersion, currentValue);
35        graph.command(new OCommandSQL("UPDATE vslNode SET version = ?, archive = ? WHERE
              address = ?"))
36           .execute(currentVersion, archive, address);
37        graph.command(new OCommandSQL("UPDATE vslNode SET timestamp = ? WHERE address = ?"
              )).execute(new Date(), address);
38      }
39    }
40    graph.commit();
41   } catch (Exception e) {
42    System.out.println(e.getMessage());
43    graph.rollback();
44   } finally {
45    graph.shutdown();
46   }
47 }
```

Listing 5.4: Implementation of setValueTree() method in OrientDB

5.3 Database Benchmark

In section section 4.1 we have constructed a possible evaluation scenarios for Virtual State Layer database backend. In following paragraphs we present selected parts of implementation of them.

To evaluate a database backend we created a service within DS2OS, that connects to local Knowledge Agent and then performs several tests. During setup phase, service connects to specified Knowledge Agent and registers its Context Model. Context Model have to be created and placed in Context Model Repository beforehand. It is an *xml* file, that specifies how the Context Tree is structured. Our Context Model consists of root and two subtrees. Each subtree represent different tree structure, either 'wide' or 'deep' (See section 4.1). It is constructed in such way to evaluate how the databases handle different tree structures. On this listing 5.5 you can see an example of 'wide' tree part of our test Context Model. For Extensibility it is divided into three parts:

- "broadRoot", that represent a root node and its direct children of a 'wide' tree
- "broadSecondLevel", that represent the form of direct children of a root node
- "broad", that represent the leaf nodes in the 'wide' tree

Each node within a tree has a simple address consisting of "node" and a number between 1..10. This is made for easier generation of addresses, represented as strings, that we use for get/set operations within service *DatabaseBenchmarkService*.

Other parts of our Context Model, such as 'deep' tree can be seen in Appendix A.

```
//broadRoot

<model type="/basic/text, /basic/composed">
  <node1 type="/test/tree/broadSecondLevel" />
  <node2 type="/test/tree/broadSecondLevel" />
  <node3 type="/test/tree/broadSecondLevel" />
  <node4 type="/test/tree/broadSecondLevel" />
  <node5 type="/test/tree/broadSecondLevel" />
  <node6 type="/test/tree/broadSecondLevel" />
  <node7 type="/test/tree/broadSecondLevel" />
  <node8 type="/test/tree/broadSecondLevel" />
  <node9 type="/test/tree/broadSecondLevel" />
  <node10 type="/test/tree/broadSecondLevel" />
</model>

//broadSecondLevel

<model type="/basic/text, /basic/composed">
  <node1 type="/basic/text, /test/tree/broad" />
  <node2 type="/basic/text, /test/tree/broad" />
  <node3 type="/basic/text, /test/tree/broad" />
  <node4 type="/basic/text, /test/tree/broad" />
  <node5 type="/basic/text, /test/tree/broad" />
  <node6 type="/basic/text, /test/tree/broad" />
  <node7 type="/basic/text, /test/tree/broad" />
  <node8 type="/basic/text, /test/tree/broad" />
  <node9 type="/basic/text, /test/tree/broad" />
  <node10 type="/basic/text, /test/tree/broad" />
</model>

//broad (represents leafs in a tree)

<model type="/basic/composed">
  <node1 type="/basic/text" />
</model>
```

Listing 5.5: Context Model of 'wide' tree, divided into three parts, which results in 110 nodes total

In the listing 5.6 you can an part of DatabaseBenchmarkService Class, that is responsible

for execution of the tests. At first we generate addresses of 'deep' and 'wide' tree, based on previously discussed Context Model of our test tree. After, depending on number of executed get and set operations we generate unique random alphabetic strings. These strings are used as values for Context Nodes. To generate the strings we use Apache Commons *RandomStringUtils* instance. Then for each type of a tree we execute all tests 5 times and calculate an average time time needed to perform a single test. The results are written into a file.

```
public void run() throws IOException {
   List<String> deepTreeAddresses = generateAddressesDeepTree();
   List<String> wideTreeAddresses = generateAddressesWideTree();
   System.out.println("Start tests");
   ArrayList<List<String>> strategies = new ArrayList<List<String>>();
   strategies.add(wideTreeAddresses);
   strategies.add(deepTreeAddresses);
   for(List<String> strategy : strategies) {
    int[] numberOfOperartions = {1, 10, 100};
    for (int n : numberOfOperartions) {
     ArrayList<String> values = generateRandomValues(n);
     List<Double> resultsOfExecution = executeAllTests(5, n, values, strategy);
     System.out.println(resultsOfExecution);
     // write the results to a file
     FileWrite write = new FileWrite("result");
     int totalNumber =n*strategy.size();
     write.append("Number of operations: " + totalNumber);
     write.newLine();
     for (Double averageTime : resultsOfExecution) {
      write.append(averageTime.toString());
      write.newLine();
     }
     write.closeFile();
    }
   }
   System.out.println("End tests");
 }
```

Listing 5.6: Implementation of run() method

As an example, an implementation of a balanced test can be seen in listing 5.7. In the method we can specify how many times should it be executed, number of operations performed on each node of a tree and what data set to use. During the execution will get a number of get/set commands. In case of balanced test, the proportion is 50/50. To record the time need to perform all runs of the test we use *System.nanoTime()* method, which is recommended if two points in time should be accurately compared. In the end we return the average time of all runs in milliseconds.

```
 /**
  * Execution of 50% set and 50% get commands
  */
```

```
4  private Double balancedTest(int numberOfRuns, int numberOfOperations, ArrayList<String>
       values, List<String> tree) {
5    long start = System.nanoTime();
6    for (int j = 1; j <= numberOfRuns; j++) {
7     for (String nodeAddress : tree) {
8      for (int i = 0; i < numberOfOperations; i++) {
9       if (i % 2 == 0) {
10        try {
11          connector.get(rootAddress + nodeAddress);
12        } catch (VslException e) {
13          System.out.println("Error on get operation: " + e.getMessage());
14          continue;
15        }
16       } else {
17        try {
18          connector.set(rootAddress + nodeAddress, nodeFactory.createImmutableLeaf(
                values.get(i)));
19        } catch (VslException e) {
20          System.out.println("Error on set operation: " + e.getMessage());
21          continue;
22        }
23       }
24      }
25     }
26   }
27   long finish = System.nanoTime();
28   Double averageExecutionTimeInMilliseconds = (finish - start) / numberOfRuns /
       1000000.0;
29   return averageExecutionTimeInMilliseconds;
30 }
```

Listing 5.7: Implementation of balanced test

Chapter 6

Evaluation

In this chapter we present the results, obtained during performance evaluation of different database backend for Virtual State Layer. As mentioned in chapter 4 and chapter 5 we evaluate the databases using a service within DS2OS, that is also designed and constructed by us.

6.1 Assessment of expected functionality

After implementing MongoDB and OrientDB for Virtual State Layer, their functionality was checked by unit tests, that we derived and extended from previously existed unit tests for HSQLDB. After the functionality of MongoDB and OrientDB was assessed, we have connected them to Virtual State Layer and started a local Knowledge Agent. After several adjustments MongoDB has flawlessly started to function as database backend for Knowledge Agent. Using Robo 3T (version 1.1, previously known as Robomongo), which a lightweight GUI for MongoDB, we could see that Context Nodes are created and updated according to required scheme.

After starting Knowledge Agent with a OrientDB as database backend, we have used a build-in web GUI to assess whether the data is created properly, according to the scheme, specified in section 4.2.2. The web GUI of OrientDB has several different ways to represent the data: as separate records in a table, similar to relational database, and as a graph. In the figure 6.1 you can observe a view for graph representation with whole Context Tree that is created on start-up of a single Knowledge Agent. Using this web GUI we assessed that a Context Tree and nodes within it are created and updated according to our design.

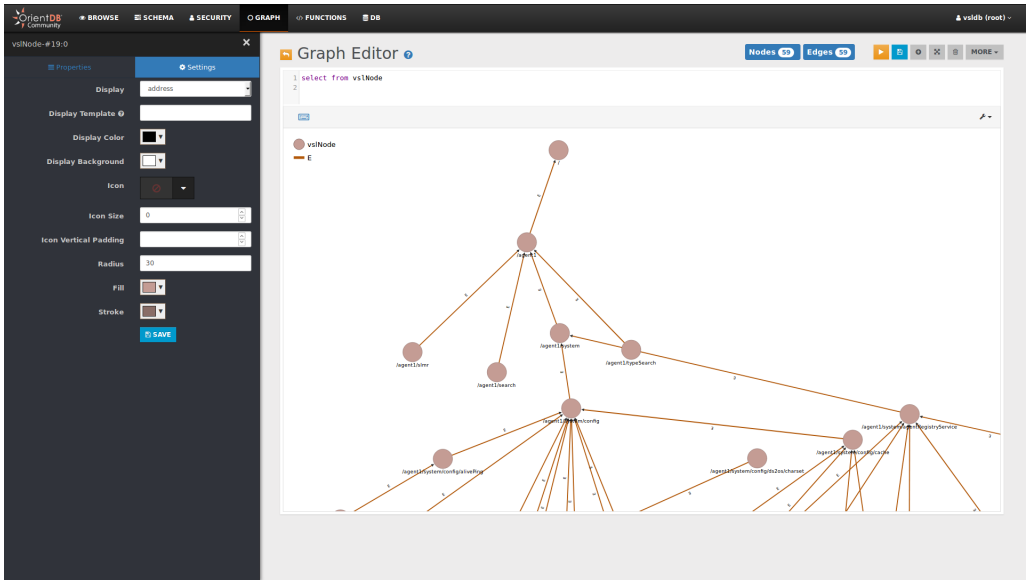

Figure 6.1: Web GUI for graph representation in OrientDB

6.2 Evaluation of performance under different load and environment

After the above mentioned assessments we started the evaluation process. We evaluated databases on the PC, located on the the Chair of Network Architectures and Services at the Technical University of Munich. As mentioned before this choice was met, because we want to get reproducible and comparable results. So that if other databases will be implemented in the future, we would be able to directly compare their evaluation with existing results. The specifications of the PC are listed in table 6.1.

OS	Debian 9.2 stretch
Kernel	x86_64 Linux 4.9.0-4-amd64
CPU	Intel Xeon CPU E3-1265L V2 @ 3.5 GHz
RAM	15750 MiB

Table 6.1: PC specification

Sequence of evaluated databases was following: HyperSQLDB, MongoDB and OrientDB. The results can be seen in the following figures: 6.2, 6.3, 6.4, 6.5, 6.6. The graphics are based on results of 110, 1100, 11000 get and set operations. The number of operations slightly differs from mentioned in section **??**. The difference occurs because number of operations in tied with the structure of test tree, that we use. For a tree structure we used a "wide" tree mentioned in chapter 4. Other evaluation scenarios, that for example assess difference in handling "wide" and "deep" tree have not been performed and are considered a possible material for future work. All graphics have following

structure: x-axis represent time, counted in milliseconds, needed to perform number of operations specified on y-axis. Bars represent evaluated databases in the following order: HyperSQLDB, MongoDB and OrientDB.

In the figure 6.2 you can see results of evaluation scenarios, where we simulate write-intensive environment. For that purpose we issue only "set" operations using DatabaseBenchmarkService. As you can see, HyperSQLDB performs almost as good as MongoDB. The difference between these databases becomes significant only after 11000 performed operations. OrientDB has performed significantly worse. We assume that this behavior is related to the way we keep the archive in OrientDB. After 11000 set operations each node has at least 100 previous states, stored in a map within the node itself. As mentioned in chapter 4 we can lift this constrain by introducing a mechanism, that monitors and limits the number of states in each node, on the application level.

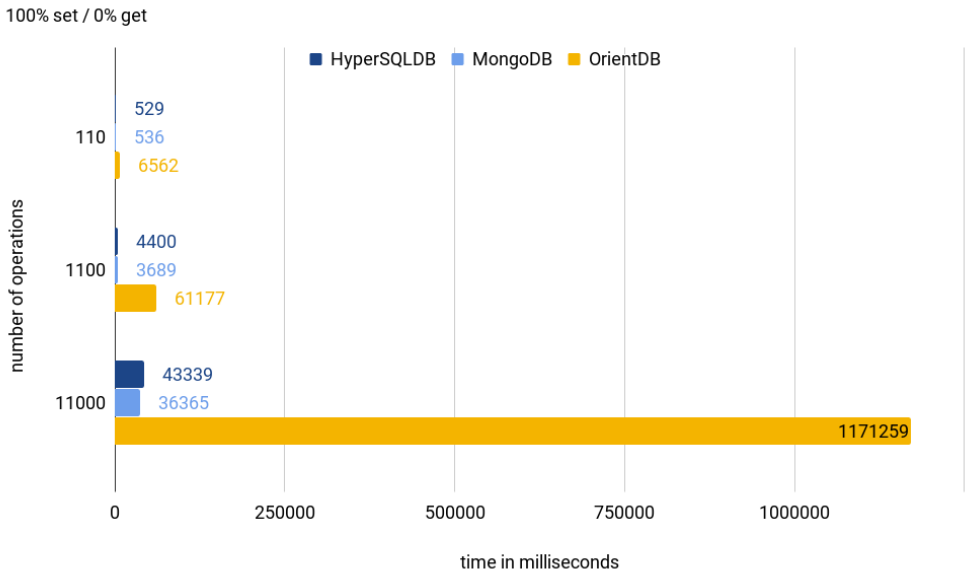

Figure 6.2: A graph representing performance by evaluation scenario with 100% set / 0% get operations

In the figure 6.3 you can see results of evaluation scenarios, where we simulate slightly less write-intensive environment. 20% of operations are in this case are "get" operations. In them we query value of a single node on specified address. The differences in performance are similar to the previous case, although in this case, HSQLDB has performed even slightly better than MongoDB at 1100 and 11000 operations.

In the figure 6.4 you can see results of evaluation scenarios, where we simulate balanced environment, consisting of 50% set and 50% get operations. On the scale of hundred operations, MongoDB performs significantly better then other solutions. On the same scale HSQLDB and OrientDB are similar in therms of performance. After 1100 operations

MongoDB and HSQLDB perform almost on the same level. And OrientDB, as expected, performs several times slower.

In the figure 6.5 and figure 6.6 you can see results of evaluation scenarios, where we simulate a more read-intensive environment. Here difference between MongoDB and other databases becomes more pronounced on each scale of performed operations. MongoDB performs at least 2 times faster then other competitors, when number operations exceed 1100.

6.3 Summary

After evaluation of our solution we can summarize our findings as follows.

OrientDB fits all requirements, that are posed in chapter 2. It can store tree-structures natively, has Java API, can ensure consistency during operations. But our implementation of archival mechanism is proven itself to be slower than expected. To solve this problem, we either have to monitor and restrict number of node's archived states on the application level. Or use similar approach that we used with MongoDB, and store all previous states in another cluster in the database. Based on this information we suggest that OrientDB is not a suitable solution at the moment.

MongoDB also fits above mentioned requirements. We are able to efficiently model a Context Tree using techniques, that are described in section 4.2.1. Based on results of performance evaluation we can suggest that MongoDB is in most cases as fast as existing solution, or even faster when it comes to read-intensive environments. The main constrain of MongoDB is that we can not ensure full consistency when performing update of a Context Node. Because each interaction is atomic, during update of node's ancestors versions, an error in execution can lead to unexpected state of database.

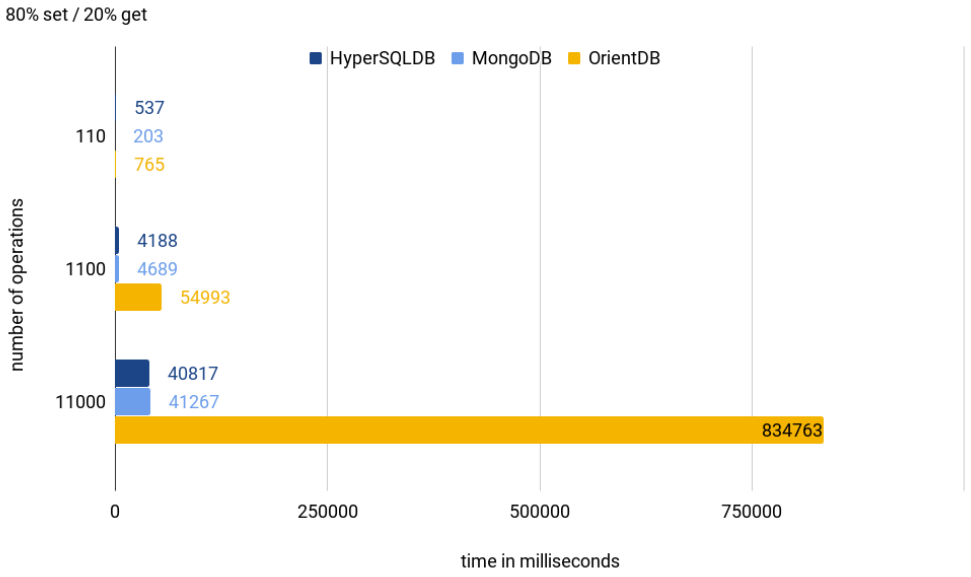

Figure 6.3: A graph representing performance by evaluation scenario with 80% set / 20% get operations

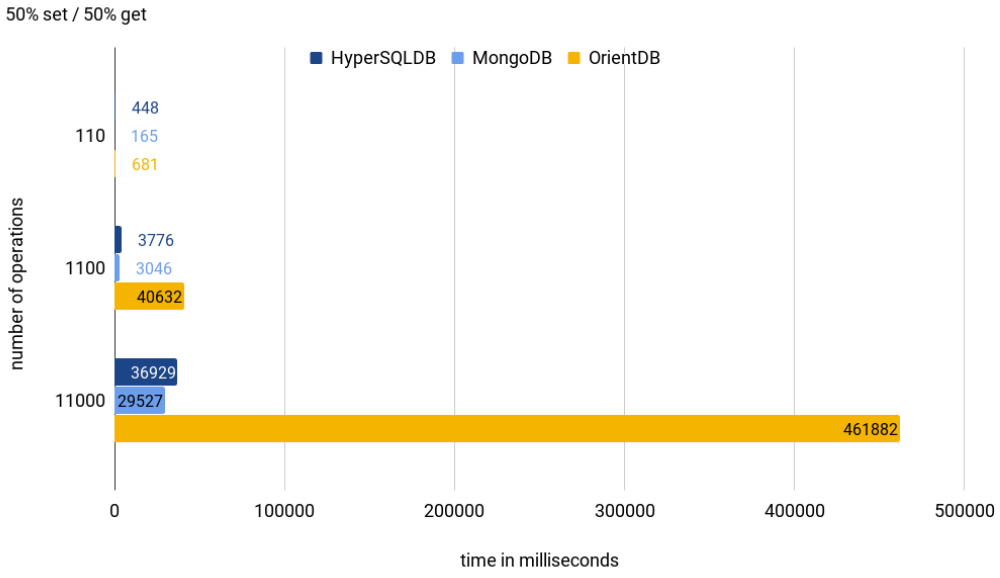

Figure 6.4: A graph representing performance by evaluation scenario with 50% set / 50% get operations

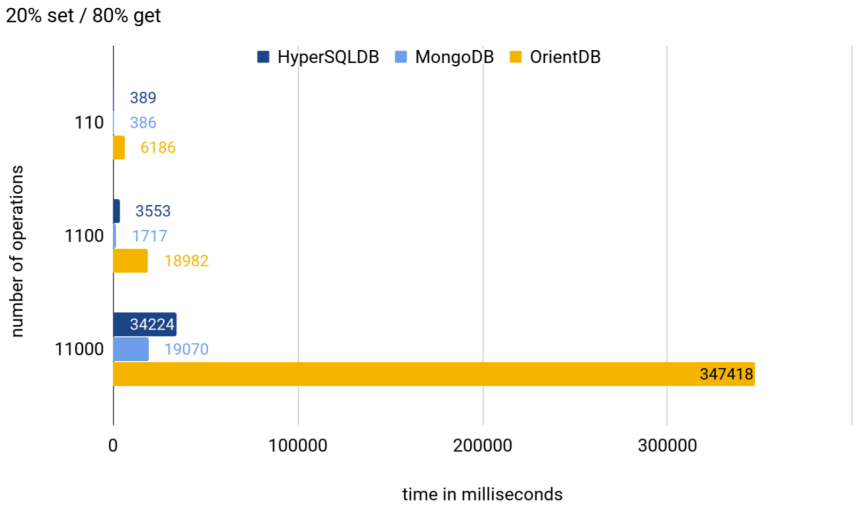

Figure 6.5: A graph representing performance by evaluation scenario with 20% set / 80% get operations

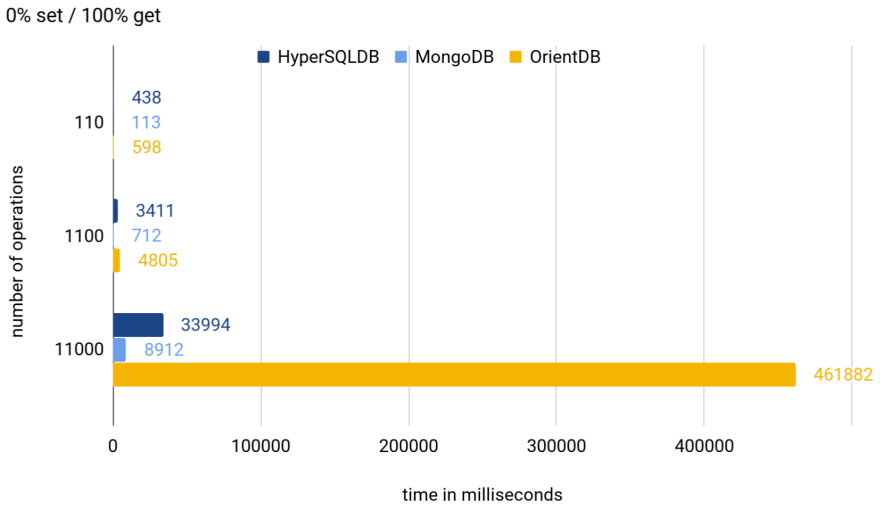

Figure 6.6: A graph representing performance by evaluation scenario with 0% set / 100% get operations

Chapter 7

Conclusion

In this thesis we have evaluated suitability of different types of databases for the Virtual State Layer, that is part of Distributed Smart Space Orchestration System. At first we presented and describe the concepts behind the Virtual State Layer, its purpose and current state of developed prototype. The most important for us was the hierarchical structure of Virtual State Layer data and specific versioning scheme inside Virtual State Layer. Additionally we have explored different types of databases, their strengths and limitations.

We have derived relevant requirements of the Virtual State Layer, considering the data model it uses. Based on these requirement we have evaluated the most suitable database types for it. Additionally we found and discussed several evaluations of databases performance, that were done by other researchers before us. Using these evaluations and our own assumptions we have limited our choice to several most relevant specific databases, MongoDB and OrientDB. The choices about how to represent Virtual State Layer data in different databases, were influenced by relevant work of others, that we have identified. This relevant information gave us an idea how to efficiently handle hierarchically structured Virtual State Layer data.

During implementation phase we have implemented above mentioned databases for Virtual State Layer, enabling all required CRUD operations. For further evaluation of performance we have design and implemented several evaluations scenarios in form of a service within Distributed Smart Space Orchestration System. Such choice was made to simulate the real-world interaction between a service, that is not aware of database behind Knowledge Agent, which is the entity of which Virtual State Layer consists.

The evaluation scenarios were constructed in way, that simulates different environments, in which Distributed Smart Space Orchestration System can be used. We wanted to observe how different databases will behave in environments, that are either read or write intensive.

During evaluation phase we have ran prepared tests first on a private machine, to obtain first results. Then we have ran them on a hardware, that is stationed on the networking chair of TU Munich. We decided to so to create a results, that can be comparable in the future with other, if further databases will be implemented and evaluated for Virtual State Layer.

After evaluation we came to conclusion, that MongoDB can be used as a database backend for Virtual State Layer. It has similar performance as HSQLDB in write-intensive environment and performs better than HSQLDB in read-intensive environment. On other hand, implementation of OrientDB at its current state is not a suitable solution. It performs significantly worse than other evaluated databases.

7.1 Future work

We consider implementation of other database backend for Virtual State Layer, as a main direction for future work. In particular it would be beneficial to consider more advanced relational databases, such as PostgreSQL as possible candidate for implementation in the future. Additionally current implementation of selected databases can be evaluated and further improved.

The evaluation scenarios can be further extended and tuned to represent the conditions, under which DS2OS functions, more precisely. At the moment they are relatively straightforward, enabling much configuration. In the future the service that is used for benchmarking can be extended to support highly configurable scenarios. It would make future evaluation more accurate and easier.

It should be kept in mind, that databases, that we selected and implemented, are in constant development. Therefore in the future new features can be introduced by their developers. Which can potentially improve performance and make the implementation of them for Virtual State Layer easier and more efficient. This in its case benefit the Distributed Smart Space Orchestration System as a whole.

Appendix A

Code examples

```
/**
 * Adds filters to aggregation pipeline, which are needed to retrieve requested
 * version of a node from "archive" collection
 *
 * @param appliedFilters
 * @param requestedVersion
 */

private void filterRequestedVersion(List<Bson> appliedFilters, int requestedVersion) {

  appliedFilters.add(Aggregates.match(lte("version", requestedVersion)));
  appliedFilters.add(Aggregates.sort(Sorts.ascending("address", "version")));
  appliedFilters.add(Aggregates.group("$address", Accumulators.last("types", "$types"),
     Accumulators.last("readerIDs", "$readerIDs"), Accumulators.last("writerIDs", "
         $writerIDs"),
     Accumulators.last("restriction", "$restriction"),
     Accumulators.last("cacheParameters", "$cacheParameters"), Accumulators.last("
         version", "$version"),
     Accumulators.last("value", "$value"), Accumulators.last("timestamp", "$timestamp"),

     Accumulators.last("ancestors", "$ancestors"), Accumulators.last("parent", "$parent
         ")

  ));
  appliedFilters.add(Aggregates.project(fields(computed("address", "$_id"), include("
      types", "readerIDs", "writerIDs",
      "restriction", "cacheParameters", "version", "value", "timestamp", "ancestors", "
          parent"), exclude("_id"))));
}
```

Listing A.1: Helper method for getNodeRecord() method, version choice

```
private void filterRequestedDepth(List<Bson> appliedFilters, int requestedDepth, String
     address) {
  switch (requestedDepth) {
```

```
    case -1:
      appliedFilters.add(Aggregates.match(or(eq("address", address), in("ancestors",
          address))));
      break;
    case 0:
      appliedFilters.add(Aggregates.match(eq("address", address)));
      break;
    case 1:
      appliedFilters.add(Aggregates.match(or(eq("address", address), eq("parent", address)
          )));
      break;
    }
  }
```

Listing A.2: Helper method for getNodeRecord() method, depth choice

```
//combinedTree, root node of whole Context Model
<model type="/basic/text, /basic/composed">
  <wide type="/test/tree/broadRoot" />
  <deep type="/test/tree/deepTree/rootDeepNode" />
</model>

//rootDeepNode, root node of 'deep' tree
<model type="/basic/composed">
  <node1 type="/basic/text, /test/tree/deepTree/intermediateDeepNode" />
  <node2 type="/basic/text, /test/tree/deepTree/intermediateDeepNode" />
  <node3 type="/basic/text, /test/tree/deepTree/intermediateDeepNode" />
</model>

//intermediateDeepNode, levels between root node and leafs of 'deep' tree
<model type="/basic/composed">
  <node1 type="/basic/text, /basic/composed">
    <node1 type="/basic/text, /basic/composed">
      <node1 type="/basic/text, /basic/composed">
        <node1 type="/basic/text, /basic/composed">
          <node1 type="/basic/text, /basic/composed">
            <node1 type="/basic/text, /test/tree/deepTree/finalDeepNode"/>
          </node1>
        </node1>
      </node1>
    </node1>
  </node1>
</model>

//finalDeepNode, leaf nodes of 'deep' tree
<model type="/basic/composed">
  <node1 type="/basic/text" />
  <node2 type="/basic/text" />
  <node3 type="/basic/text" />
</model>
```

Listing A.3: Test tree Context Model

Bibliography

[1] M.-O. Pahl, "Distributed smart space orchestration," Dissertation, Technische Universität München, München, 2014.

[2] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Aug 2013, pp. 15–19.

[3] C. Strauch, "Nosql databases," accessed: 2017-05-11.

[4] J. Celko, *Joe Celko's Trees and Hierarchies in SQL for Smarties*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[5] "Adjacency list vs. nested sets: Postgresql," https://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/, accessed: 2017-06-25.

[6] "Nested set model," https://en.wikipedia.org/wiki/Nested_set_model, accessed: 2017-06-25.

[7] "Closure tables for browsing trees in sql," https://coderwall.com/p/lixing/closure-tables-for-browsing-trees-in-sql, accessed: 2017-06-25.

[8] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," in *2011 International Conference on Cloud and Service Computing*, Dec 2011, pp. 336–341.

[9] "Db-engines ranking of key-value stores," https://db-engines.com/en/ranking/key-value+store, accessed: 2017-07-25.

[10] "Model tree structures," https://docs.mongodb.com/manual/applications/data-models-tree-structures/, accessed: 2017-06-26.

[11] "Db-engines ranking of document stores," https://db-engines.com/en/ranking/document+store, accessed: 2017-07-25.

[12] "Orientdb manual - version 2.2.x," https://orientdb.com/docs/2.2/, accessed: 2017-11-25.

[13] T. A. M. Phan, J. K. Nurminen, and M. D. Francesco, "Cloud databases for internet-of-things data," in *2014 IEEE International Conference on Internet of Things (iThings),*

and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom), Sept 2014, pp. 117–124.

[14] H. Fatima and K. Wasnik, "Comparison of sql, nosql and newsql databases for internet of things," in *2016 IEEE Bombay Section Symposium (IBSS)*, Dec 2016, pp. 1–6.

[15] J. S. van der Veen, B. van der Waaij, and R. J. Meijer, "Sensor data storage performance: Sql or nosql, physical or virtual," in *2012 IEEE Fifth International Conference on Cloud Computing*, June 2012, pp. 431–438.

[16] M. G. Jung, S. A. Youn, J. Bae, and Y. L. Choi, "A study on data input and output performance comparison of mongodb and postgresql in the big data environment," in *2015 8th International Conference on Database Theory and Application (DTA)*, Nov 2015, pp. 14–17.

[17] "How to track versions with mongodb," http://www.askasya.com/post/trackversions/, accessed: 2017-07-20.

[18] "Further thoughts on how to track versions with mongodb," http://www.askasya.com/post/revisitversions/, accessed: 2017-07-20.

[19] J. Yao, "An efficient storage model of tree-like structure in mongodb," in *2016 12th International Conference on Semantics, Knowledge and Grids (SKG)*, Aug 2016, pp. 166–169.

[20] D. Jayathilake, C. Sooriaarachchi, T. Gunawardena, B. Kulasuriya, and T. Dayaratne, "A study into the capabilities of nosql databases in handling a highly heterogeneous tree," in *2012 IEEE 6th International Conference on Information and Automation for Sustainability*, Sept 2012, pp. 106–111.

[21] "Mongodb docs," https://docs.mongodb.com/, accessed: 2017-11-25.

[22] "Aggregation pipeline," https://docs.mongodb.com/manual/core/aggregation-pipeline/, accessed: 2017-11-25.