SQL, noSQL or newSQL – comparison and applicability for Smart Spaces

Christoph Rudolf
Advisors: Stefan Liebald, M.Sc. and Dr. Marc-Oliver Pahl
Seminar Innovative Internet Technologies and Mobile Communication (WS 16/17)
Chair for Network Architectures and Services
Department of Informatics, Technische Universität München
Email: christoph.rudolf@tum.de

ABSTRACT

In this paper, we analyze database architectures in terms of their applicability as a data storage for distributed data in smart spaces. As a concrete example, the paper takes the *Distributed Smart Space Orchestration System (DS2OS)* which acts as a management system for smart spaces and implements a peer-to-peer network. We develop key requirements for a database solution based on the data characteristics of DS2OS. These requirements allow us to analyze representatives of database architectures regarding their applicability. We conclude that the simple data models provided by NoSQL databases are suitable for modeling the desired data structure. Based on their performance and features, we propose *PostgreSQL*, *Redis* and *InfluxDB* as suitable solutions for DS2OS.

Keywords

NoSQL, NewSQL, database types, DS2OS, IoT, smart spaces

1. INTRODUCTION

Smart spaces are real-world spaces with embedded devices that capture data about their environment and control aspects of it [14]. The data that is shared in smart spaces originates from distributed sources connected to each other. This offers challenges regarding the storage of data handled by a common middleware. The *Distributed Smart Space Orchestration System (DS2OS)* is such a middleware and is used as a representative to asses different paradigms for database management regarding their suitability as a data storage in smart spaces. This paper starts by introducing special characteristics of data in DS2OS. Further, it gives an overview over database architectures. We then develop requirements to be met by a suitable database solution and select representatives of each architecture type for evaluation.

2. DATA STRUCTURES IN THE DS2OS

The *Distributed Smart Space Orchestration System (DS2OS)* is a management system for smart devices. It was started in the Ph.D. thesis of Marc-Oliver Pahl [14] and is actively developed at the Chair of Network Architectures and Services at the Technical University of Munich[1]. The system contains a middleware for the uniform abstraction of smart spaces. It targets the heterogenic field of today's smart devices and makes them manageable by a central software.[16]

[1] `www.ds2os.org`

While this paper focuses specifically on the data structure used in DS2OS, we start by introducing key architecture components which are necessary for understanding the data structure. The central element and middleware of DS2OS is the *Virtual State Layer (VSL)*, a peer-to-peer system built of so called *Knowledge Agents (KA)* as its peers [16]. The VSL is self-organizing in a sense that peers automatically locate and connect to each other. In addition to that, the VSL handles synchronization of data between agents. In a real-world scenario, these interconnected agents run on computing devices in different physical locations of a smart space. They provide an interface between a diverse set of services and the VSL. Services can, for example, manage a smart device and encapsulate its complexity, collect sensor data or output information to the user.

One of the major aspects of a system like DS2OS is the design of a suitable data model for all possible services and devices. This is especially important to allow for previously unsupported devices to be introduced into the system. Therefore, domain-specific data has to be added dynamically.[16] To do so, DS2OS provides a directory of *context models* that can be extended by developers. A context model is a data structure that holds properties of a service and acts as an interface. A standard context model for services which are, for example operating a lamp, is desired to be adopted by many services for lamps to decouple service implementation from the specific devices. Context models are currently stored in an XML-format.[15]

The context model of a service is instantiated when a service initially connects to a KA. The context models consist of "hierarchically structured typed key-value pairs (...) with additional management metadata" [15] and can be thought of as a tree data structure that allows to address each attribute, similar to files in a filesystem. Figure 1 visualizes the interconnected agents, their services, the data as well as sample attributes for a lamp as service S_1.

All information depicted in Figure 1 is known to every KA in the VSL as meta data. However, only the KA that is directly connected to a service stores the actual data. Other agents have to actively query the respective KA for this data by referring to its path in the hierarchic structure (e.g. `/KA1/S1/switchedOn/`) which forms a unique identifier. They can also subscribe to an address and be notified of changes. These notifications are sent by the KA in charge of storing the data. Storing the data as well as the structure

doi: 10.2313/NET-2017-05-1_06

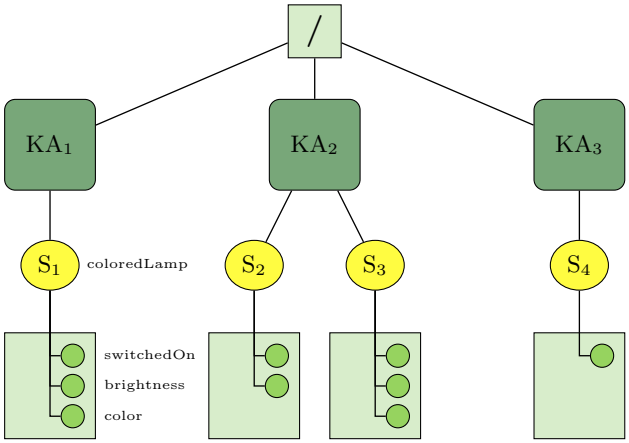

Figure 1: Visualization of the DS2OS data structure known to each KA.

is currently implemented by using the relational database *HSQLDB* with a database schema of three tables:

structure : {[address, type, readers, writers, restriction, cachedParameters]}
version : {[address, timestamp, version]}
data : {[address, timestamp, value]}

The address inside the tree structure references specific entries in all three tables. All addresses are associated with a semantic data type in the context of the VSL that is independent of the type in the database.

The `structure` table is responsible for storing the complete structure depicted in Figure 1 and additional metadata like restrictions and permissions. The `version` table receives new entries for every change made to a value, as the `data` table is not updated in case of a value being changed. Instead, a new entry with a timestamp is added, in addition to the new version in the other table. The version update via insert does also happen for all prefixes of the changed address up to the service level. For this reason, separated tables help to insert versions for all parent nodes without duplicating their values. By inserting new versions, the database for DS2OS has to handle mainly insert operations.

3. DATABASE ARCHITECTURES

Due to new database architectures, the field of *database management systems (DBMS)* has been diversified over recent years. While relational databases are prevalent in the majority of all use cases, unstructured data calls for new approaches [11]. Not all DBMS can be assigned solely to a single architecture type of databases, as many provide features that are characteristics of other groups. This section provides a brief overview over current database architectures.

3.1 Relational database

Relational databases are currently the most widespread type of databases [6], with well-known DBMS like *PostgreSQL* and the briefly mentioned *HSQLDB* used in DS2OS.

Their idea was proposed as early as 1970. With this kind of databases, data is stored in tables which model relations in a mathematical sense. Each table models an n-ary relation, by specifying n columns with each storing a certain type of information for all entries. The column definition provides the scheme of the table. The data itself is stored as row entries in form of n-tuples.[3] This design approach makes relational databases adaptable to many application domains. They allow to define certain columns as unique key values and to create references between relations. If utilized correctly, this reduces redundancy of data.

Relational databases provide powerful querying capabilities based on relational algebra. Relational algebra defines a set of operations working on relations. Every output generated by a query combining relations with their operations is a new relation in itself. This enables complex nested queries. It is within the responsibility of the user or an optimizer integrated into a DBMS to make them as efficient as possible. The features of relational algebra are implemented by SQL which acts as a *de facto* standard query language for modern relational databases.[5]

Relational databases usually do well in terms of concurrency and reaction to failures as they provide ACID transactions. By being ACID compliant, transactions are supposed to fulfill the following principles [10]:

- ***A**tomictiy:* transaction are considered as a single unit
- ***C**onsistency:* results are consistent database states
- ***I**solation:* no side-effects among parallel transactions
- ***D**urability:* their results are persistent

This is necessary if a series of critical changes is made that can only be allowed to complete fully or have no effect at all. While transactions are no inherent attribute of relational databases, it usually distinguishes them from newer approaches. Whether or not transactions are needed is subject to the application domain.

3.2 NoSQL

While relational databases are adaptable to many scenarios, not all data models fit well with a relational schema. Relational databases are inflexible when the schema has to be changed due to redesigns in the underlying application. In addition to that, their sophisticated features, like ensuring data consistency, add overhead which is sometimes unnecessary and critical in contexts of massive amounts of data being handled. This has become an issue for online services over the recent years, which are in need of database systems that can be distributed.[7, 9]

NoSQL databases solve these problems with different approaches to data organization and an increased focus on distributability. The term NoSQL does not describe a specific group of databases, instead, it is a collective term for systems that break the classic relational approach.[8] Without the relational basis, NoSQL databases do not provide a common query language. In order to improve the scalability, many NoSQL databases abandoned ACID transactions. Subsequently, we discuss the most important types of NoSQL databases.

doi: 10.2313/NET-2017-05-1_06

3.2.1 Key-value stores

This kind of NoSQL database models all data as simple key-value pairs. Values are stored as plain byte-arrays. usually without a data type specified on the database side. While this is limiting in terms of data structuring, the storage is extremely adaptable, as changes to the application do not require the database to be adapted. It also does not require `null` values like they appear in relational models. Complex queries are not possible due to the lack of interpreted data, however, because of the flat and simple structure, this is not needed. Instead, simple `get` and `set` commands in conjunction with unique keys are the only way to read and write data to the store. Many implementations, such as *Redis* [6], work as in-memory databases with optional mechanisms for saving to a persistent disk. Being in-memory allows them to have a much faster but, in terms of size, limited storage. *Redis* in particular does also support data types.[8, 9]

3.2.2 Document-oriented database

Document-oriented databases are related to key-value stores, but add a means of structuring to them by allowing to group key-value pairs into documents. The keys then only have to remain unique inside the scope of the document. Each document receives a unique identifier as well. Queries to fetch data are improved beyond simple `get`-operations by also allowing to query documents based on their properties inside. The values associated with the keys are not opaque byte arrays like for key-value stores. Documents are allowed to be nested further and support basic datatypes like lists, strings and numeric values. The features are similar to the ones being expressed by JSON syntax which leads to many document-oriented DBMS using JSON or at least a JSON-like data format. There is no restriction on the schema of documents which allows adding and removing entries to and from, possibly heterogenous, documents. Utilizing the XML document standard is also possible and allows queries with *XQuery*, a query language that aims to fulfill the same role as SQL in the relational world.[9]

3.2.3 Column-oriented database

This type of databases is similar to key-value stores in a sense that key-value pairs are stored without any interpretation of the values inside the database. Like for key-value stores, additional logic regarding relationships and types of the values is shifted to the application. The main difference of this database type to key-value stores is that it deviates from being row oriented in its physical organization. Instead, all values of each column are grouped together. This is beneficial in cases where not all values of a single row are of interest. Due to grouped columns, their values are also stored very close in memory. This optimizes the performance in cases where all rows and only few columns of each one are needed, as there will be less page faults and therefore less slow hard disk access.[9, 8] The concept was initially introduced by Google's Bigtable and is not limited to NoSQL, as relational data can also be organized column-oriented.

3.2.4 Graph-based

Graph-based databases are optimized for data which emphasizes the relationship between data objects. Their primary elements for data modeling are nodes and edges which may be associated with key-value pairs to store additional properties. Querying relies on different graph traversal algorithms like breadth-first search to find single matching nodes or depth-first search for shortest paths. While graph structures can be represented in a relational database, graph traversal cannot be expressed in SQL which leads to decreasing performance for larger data sets [8]. Examples are social networks (e.g. Twitter's FlockDB) or storing location data.

3.2.5 Time series database

This type of database focuses on handling time series data. This covers all data with one or multiple values being stored at continuous time intervals while one is still being interested in past values for processing them. Typical scenarios in which time series appear are statistical data, logging or sensor data. A time series is identified by a name, includes a timestamp and values. This could basically be modeled by a relational database if it is feasible to create a new table for each time series. By putting more than one series into a table, one would have to define the superset of all values as its scheme and deal with many `null` values or make the values opaque to the database which hurts querying. Time series databases are specifically built for handling this data efficiently and are especially useful if the amount of data is very large.[12]

3.2.6 Object-oriented databases

According to the *db-engines.com* ranking [6], object-oriented databases are currently among the least used databases. Their distinct advantage over relational databases is that they are capable of modeling concepts like inheritance and polymorphism. This is especially useful if the application using the database utilizes object orientation as well. Therefore, object-relational mapping (ORM), which is often used to translate relation data into objects for usage in the application, is not needed, thus, removing an additional layer of complexity when accessing the database.

However, there are disadvantages which account for the limited acceptance of object-oriented DBMS. Due to their diverse structure depending completely on the domain model they are used for, they cannot rely on relational algebra and cannot create new objects by joining existing ones in queries. In addition to that, they lack a standard query language, like SQL for relational databases.[2] This kept object-oriented databases from getting increasing market shares. Instead, using relational databases and ORM is used by most object-oriented applications with databases.

3.3 NewSQL

The term NewSQL describes new database architectures, deviating from the way relational DBMS are implemented. In contrast to NoSQL databases, they aim to maintain characteristics of relational databases, like SQL as a query language, support for the relational model and ACID transactions. At the same time, they provide additional performance, scalability and distributability. This is done by leveraging modern hardware and deploying improved algorithms which haven't been available yet, when older existing DBMS had been designed. NewSQL databases rewrite the core of the database system from scratch to remove legacy code that hinders distributability and performance due to its assumptions being outdated.[8] By rewriting the very basis of a DBMS, they utilize modern multi-core architecture

doi: 10.2313/NET-2017-05-1_06

and support clustering in a native way compared to older RDBMS which are missing these features or had them added in a much later stage of their product lifecycle.

The main concepts of NewSQL have been introduced by Stonebraker et. al. [19] in a paper that also introduced the implementation *H-Store* that provides ACID transactions and excels at *online transaction processing (OLTP)*. The paper also points out that a single architecture is not sufficient in all use cases and that multiple but very optimized DBMS for each field of application are preferable.

Operating in-memory is often mentioned as a key property of NewSQL, making relational databases like *SAP Hana* which supports this feature also a NewSQL database to some extent. Other examples are *Google Spanner* and *VoltDB*

4. CRITERIA FOR COMPARISON

The differences among the wide variety of databases introduced in Section 3 show that the most suitable choice for a database depends on the needs of the application at hand. In order to determine a fitting type or even a concrete DBMS software for usage within DS2OS, we have to define the criteria that are important for our use case. In the process, we identify criteria that are quantifiable and can be determined by measurements, like for example performance values. In addition to the measurable criteria, there are requirements which *must* be fulfilled and other properties that are considered to be beneficial but *may* not be fulfilled. We distinguish these as *hard* and *soft* requirements.

General criteria for evaluation without a specific use case are presented in related work [9, 20] and cover attributes like scalability, query support, broad data model support and distributability. These are only partly applicable in our case, as they are very general or not of importance in our special case. Nevertheless, ideas for evaluation criteria can be drawn from related work.

4.1 Hard requirements

H1: The database must be able to store the data presented in Section 2. It is not necessary to model the same relational schema. However, the database must support basic key-value pairs associated with a timestamp. The additional versioning of data is currently modeled in a separate database table, however, it would be beneficial to be handled natively (see Section 4.2). The same goes for access control to certain service data.

H2: As DS2OS is currently written in Java, the DBMS of choice has to provide an API accessible from this programming language. This does not limit the field to databases that directly provide Java bindings, as an HTTP API or a connection via ODBC is also perfectly usable from Java.

H3: The current implementation of DS2OS in combination with *HSQLDB* as a database does use transactions. This is important when data is initially inserted or deleted, as this affects not only the data entry itself, but also the version stored in a separate table. Unless the proposed DBMS of choice eliminates the separation between data, version and structure completely, further support for ACID transactions is required. However, a completely different data handling could in fact lift this requirement.

4.2 Soft requirements

Many of the subsequent properties are either currently missing or are handled inside the application, because the current database does not offer support on the particular issue. It is preferable to have native support for these features instead of an implementation in Java. However, due to necessary code changes to the VSL, each of the subsequent requirements has to be put into perspective regarding how *invasive* it is in regard changes necessary to the DS2OS existing code base. With both, benefit and code changes considered, a score (✓or ✓✓) indicating its importance is given.

S1: Versioning is currently handled by the application layer of DS2OS. The relational database schema is designed to allow storing versioning information for each value. A solution like this is feasible for a future choice of database, but a native solution directly integrated within the database is preferred. We attribute such DBMS additional points during the assessment. As the current implementation encapsulates the handling of versions completely within the database wrapper, there are no further changes to the DS2OS coming with this feature. **Bonus: ✓✓**

S2: As the relational *HSQLDB* does not support being distributed directly, the distribution of data is handled on the application layer by the KA in charge of the data. Therefore, the current implementation of DS2OS provides application layer code for notifying peers of changes and ensuring that only structural data is shared. Many NoSQL databases provide native support for distribution over multiple nodes which removes complexity from the application layer. However, as this is currently directly implemented in the VSL, the refactoring of the existing code would be rather complex. Another downside to take into consideration, is that distribution on a database level imposes the usage of the same database for all agents in the network. Mixing databases among agents is currently possible due to the handling on a higher level. Hecht and Jablonski separate distributability into the support for partitioning data among multiple nodes and the replication of data [9]. In our scenario, replication is needed only for structural data. Partitioning has to be controlled to ensure that values are confined to a specific agent. **Bonus: ✓**

S3: Access control is currently available and implemented by storing which agents have read and write access to an address. The application layer processes data queries from remote Knowledge Agents and uses the stored information to grant or deny access. A native support would eliminate complexity from the agents and simplify the data model. The desired mechanism is that a KA can query data under the permissions of a peer. This is a more complex problem than the one addressed by standard role-based permissions of most DBMS. Native support on the database side should also allow querying whether or not permission is given without fetching the actual data. The amount of changes to the given code basis are considered as being in between those of the last two requirements, with smaller changes outside the immediate database wrapper. **Bonus: ✓**

doi: 10.2313/NET-2017-05-1_06

S4: Besides access control to sensor data through its managing KA, there is currently little additional security, like encryption of the data, provided. Confidentiality of the data *at rest* or even *in memory* is desirable as the possible damage coming from malicious changes to sensor data in smart spaces can result in actual physical damage. Encryption for data *in transit* is already provided by DS2OS. **Bonus: ✓✓**

S5: DS2OS allows agents to subscribe to data managed by another peer to be notified of changes. A support for such a messaging system directly inside the database is beneficial as it would reduce computational cost for the KA when processing changes. However, changes to the code outside of the database wrapper are necessary. **Bonus: ✓**

4.3 Measurable criteria

The subsequent criteria are measurable metrics regarding the performance of databases.

C1: The insert performance of the DBMS is important, as DS2OS uses mostly insert requests. Because of versioning, new entries are written for changing values. In nested data structures the number of inserts for a changing value is large as all parents up to the service node get a new version.

C2: One of the aspects a different database can improve, is the read performance. This is important due to possibly many services requesting data in a productive smart environment. Because of the tree-like structure being stored in relational database tables, querying a node and all its child elements results in many different rows to be fetched. The current database has to rely purely on string comparison of addresses to filter for the correct entries to return.

C3: Deleting is important to prevent the database from growing to large. Old versions of data are deleted if the total number of entries exceeds a certain threshold. This means that once this threshold is surpassed for an entry, every insert does also include the removal of old data. Therefore both insert and delete commands are highly important for the new database, while the performance for updates is negligible.

5. ASSESSMENT OF DBMS

Due to the large amount of existing DBMS, we pick representatives for each of the categories introduced in Section 3. Afterwards, we conduct a preliminary based on hard requirements **H1**–**H3** to narrow down the representatives before comparing them in regard of the soft requirements **S1**–**S5** and the quantifiable criteria **C1**–**C3**.

5.1 Selection of DBMS

Table 1 lists representative DBMS for each database type. The DBMS *PostgreSQL*, *Redis*, *MongoDB*, *Cassandra* and *Neo4j* are picked based on db-engines.com's DBMS ranking of popular DBMS [6]. This is reasonable based on the assumption that popular databases are under active and continuing development as well as field-tested and optimized to a degree where using them in production can be recommended. In addition to that, all selected databases are open source to fit into DS2OS which aims to act as an "enabler for a software maker culture" [16]. Therefore, it is not desired to introduce a commercial database solution.

Table 1: Representatives of each database type for further evaluation.

Database type	Representative DBMS
Relational	HSQLDB, PostgreSQL, VoltDB
Key-value stores	Redis
Document-oriented	MongoDB, Elasticsearch
Column-oriented	Cassandra
Graph-based	Neo4j
Time series database	InfluxDB
Object-oriented	ZeroDB

With *HSQLDB*, we include the current database of DS2OS for comparison. *VoltDB* is included as a in-memory NewSQL database. Note that NewSQL is not a distinct category as its representatives can implement any architecture. *Elasticsearch* is special as it is technically a search engine but can function as a document-oriented database due to storing schema-free JSON. It emphasizes on distributability by offering features like automatic clustering. The relatively unknown *ZeroDB* is included due to its unique capabilities in terms of encryption. *InfluxDB* is the most prominent time series database [6] which works in clusters of many instances. Querying can be done in a SQL-like query language that returns JSON data, with very sophisticated capabilities regarding time based queries.

5.2 Rating based on hard requirements

With the total of 10 DBMS picked, we can assess their features in regard to the hard requirements **H1**–**H3**. If one or more of these requirements are not met, the respective DBMS is eliminated from further evaluation.

Table 2: Compliance of DBMS with our hard requirements (see Section 4.1).

DBMS	H1	H2	H3
HSQLDB	✓	✓	✓
PostgreSQL	✓	✓	✓
VoltDB	✓	✓	✓
Redis	✓	✓	✓
MongoDB	✓	✓	(✗)
Elasticsearch	✓	✓	(✗)
Cassandra	✓	✓	✗
Neo4j	✓	✓	✓
InfluxDB	✓	✓	(✗)
ZeroDB	✓	✓	✓

H1: Table 2 shows that all DBMS are capable of handling the relatively simple data structure of DS2OS. Even the time-series DBMS *InfluxDB* is applicable in that regard as it allows to store strings in contrast to other time-series databases like *RDRTool*. Time-series databases, alongside graph-based databases, are the only ones where special restrictions of a concrete DBMS could hinder the modeling of DS2OS's data.

H2: No specific type of database is likely to have issues with being coupled to a Java application. The database systems listed provide either a Java API or allow access over the *JDBC (Java Database Connectivity)* interface. *ZeroDB*

doi: 10.2313/NET-2017-05-1_06

is a small exception, as it is based on the object-oriented DBMS *ZODB* which can only be used with Python. *ZeroDB* mitigates this issue by providing an API server that forwards queries to the database on behalf of the client.

H3: The requirement for ACID transactions can be lifted if the DBMS allows for a different approach without the need for transactions. This is promising in the case of *MongoDB*, *Elasticsearch* and *InfluxDB*. The column-oriented *Cassandra* is still in a sense relational and would have to model the DS2OS data like it is done currently (see Section 2). Therefore we eliminate *Cassandra* from further evaluation.

5.3 Rating based on soft requirements

The remaining databases are analyzed regarding the soft requirements **S1**–**S5**. Table 3 summarizes this evaluation. The bonus values for each soft requirement are given partially based on how elaborate the feature provided by the DBMS is on the specific regard. Unless stated otherwise, the information on available features is taken from the product's documentation.

Table 3: Compliance of DBMS with our soft requirements (see Section 4.2).

DBMS	S1	S2	S3	S4	S5	Σ
HSQLDB				✓		**1**
PostgreSQL			✓	✓	✓	**3**
VoltDB					✓	**1**
Redis		✓			✓	**2**
MongoDB		✓		✓	✓	**3**
Elasticsearch	✓	✓	✓			**3**
Neo4j			✓			**1**
InfluxDB	✓✓					**2**
ZeroDB				✓✓		**2**

The narrow field in terms of score and the highest value of 3/7 indicates that there is no ideal solution that performs exceptionally well.

S1: The first requirement revolves around versioning of data. The time series database *InfluxDB* excels naturally as versioning is a key concept of this database type. *Elasticsearch* is the only other DBMS that provides dedicated versioning features by an internal integer that can be atomically incremented [13]. For all other database solutions, versioning like it is currently done with *HSQLDB* via a database table, is the only option.

S2: Distributability, in terms of replication and partitioning, is natively supported by three candidates. *Redis* can partition its key space. Each of the nodes in the resulting cluster holds a portion of all keys. Queries for a certain key can be redirected to the node storing the correct portion. *MongoDB* provides a similar mechanism with *sharded clusters*. This is also the way *Elasticsearch* handles distribution of data. *Neo4j* does also allow to create clusters, but only in its enterprise version. Based on requirement **S2**, it is necessary to enforce that certain values are only stored on a particular host. All three candidates can solve this by using tags. For *MongoDB* this is called *Tag Aware Sharding*. *Elasticsearch* supports it as *Shard Allocation Filtering*.

S3: A permission system is beneficial for DS2OS if it allows to take the role of another KA when querying data. Both *PostgreSQL* and *Elasticsearch* provide features to query as a certain database user. The complete permissions of a user can also be queried separately. *Neo4j* models permissions with special edges that can be taken into consideration even if querying as a different user.

S4: As a database that focuses on security, *ZeroDB* provides the most advanced encryption features. It is the only assessed DBMS that does not even hold decrypted data in its memory. Other DBMS like *HSQLDB*, *PostgreSQL* and *MongoDB* provide encryption features for data at rest, but hold plain data in memory.

S5: A native subscription mechanism is provided by four of the DBMS. *PostgreSQL*, *Redis* and *MongoDB* all deploy a mechanism where the database actively published notifications to all client applications on inserts, updates and the removal of data. Access to this feature is possible from Java clients in all cases. *VoltDB* does not have a publish-subscribe mechanism, but its export functionality can be used to send live updates to an external application [21].

5.4 Evaluation of measurable criteria

The measurable performance criteria **C1**–**C3** focus on write, read and delete operations. Due to the consistent versioning, update operations are hardly used in DS2OS. To narrow down the field of databases for performance evaluation, we only consider candidates that earned at least 2 points in the evaluation of Section 5.3 and the currently used *HSQLDB*. The remaining databases are *HSQLDB*, *PostgreSQL*, *Redis*, *MongoDB*, *Elasticsearch*, *InfluxDB* and *ZeroDB*.

As existing work on the comparison of databases is sparse, we use a combination of the existing measurements and our own experiments. This covers the candidates in a way their performance can be put into relation. This way, not all databases are measured together. Instead, we compare the performance transitively to make a reasonable statement about the databases.

An existing evaluation based on the *Yahoo! Cloud Serving Benchmark (YCSB)* covers *MongoDB*, *Redis* and *Elasticsearch* regarding their performance for insert (**C1**) and read (**C2**) commands [1]. Their execution time for different sizes of data is visualized in Figure 2 and Figure 3 respectively.

The comparison shows that *Elasticsearch* lacks in terms of insert performance compared to the other DBMS, but excels when reading large data. We also conclude that *Redis* has the best overall performance for reading and writing. Due to limited hardware used for this benchmark, the absolute values are not comparable to subsequent tests.

To create additional comparison options beside the existing work, we conduct our own measurements including *HSQLDB*, *PostgreSQL* and *Redis*. This does help to put *MongoDB* and *Elasticsearch* in perspective, as they have been compared to *Redis*. All measurements are conducted on a desktop system (Intel® Core™ i5-2520M CPU @ 2.50GHz, 8 GB RAM, Ubuntu 16.04) by using either a Python client with the respective DBMS' API, or, in case of *HSQLDB*, a Java appli-

doi: 10.2313/NET-2017-05-1_06

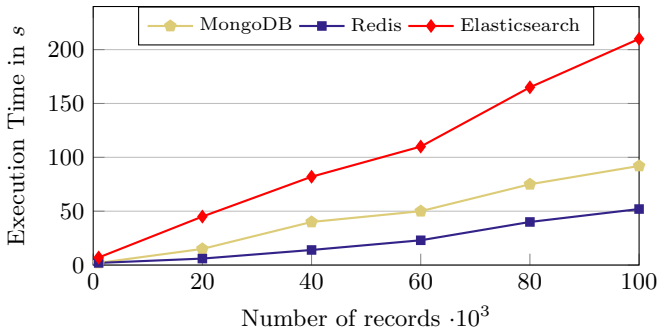

Figure 2: Insert performance of *MongoDB*, *Redis* and *Elasticsearch* (adapted from [1])

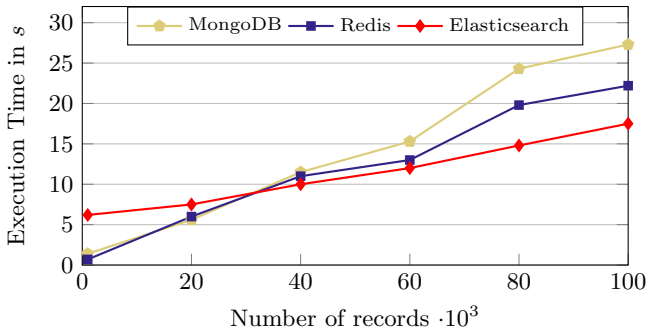

Figure 3: Read performance of *MongoDB*, *Redis* and *Elasticsearch* (adapted from [1])

cation that works like the data access wrapper in DS2OS. The values shown in the subsequent figures are the average out of 5 repetitions for each data set size.

The insertion speed (see Figure 4) is measured for different sizes of data. These inserts show that *HSQLDB* gets slower with larger data sets. The amount of data already in the database did not affect this for any DBMS in our test. As seen in Figure 2, *Redis* performs very well for large data.

Both, reading (see Figure 5) and deleting data (see Figure 6), was measured for different record sizes out of a constant data set of 100000 elements. For *HSQLDB*, we observed a high fluctuation in terms of the performance. Therefore, the number of repetitions for each selection was increased to 10.

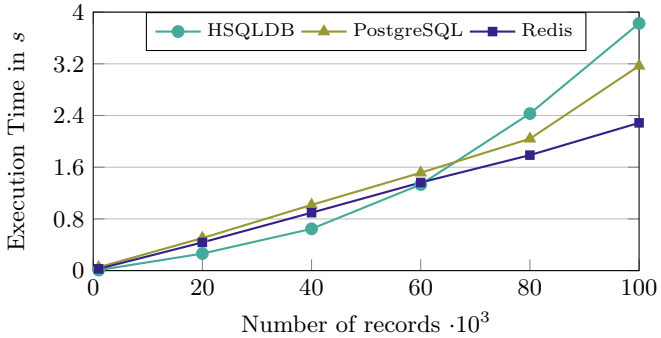

Figure 4: Insert performance of *HSQLDB*, *PostgreSQL* and *Redis*

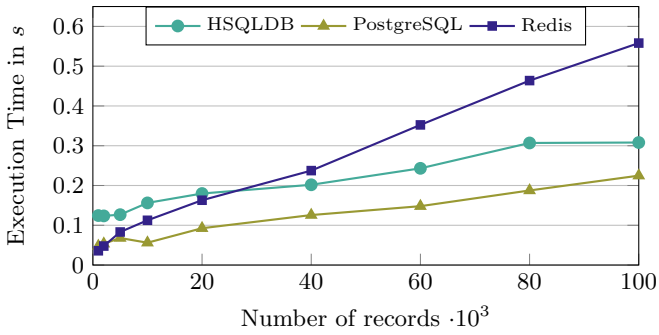

Figure 5: Read performance of *HSQLDB*, *PostgreSQL* and *Redis*

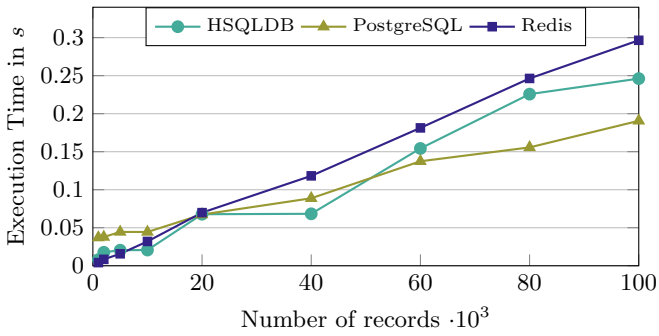

Figure 6: Delete performance of *HSQLDB*, *PostgreSQL* and *Redis*

We observe that *Redis* behaves the most predictable and is linear in regard to the time it takes for increasing data. Besides the high fluctuation in *HSQLDB's* performance, we observe that *Redis* does well for small records up to 5000, as the two RDBMS seem to have an initial overhead. *Redis* is preferable for reading smaller record numbers while *PostgreSQL* is a better overall choice. These differences in behavior based on the size of the data does not allow to put all DBMS into an order for each criteria **C1**–**C3**.

The newer and lesser known *InfluxDB* is compared to *MongoDB* and *Elasticsearch* by two white papers. The comparison with *MongoDB* is based on a data set depicting monitoring data of 100000 different values. Every value is updated every 10 s for 6 h which results in a total of 216 million values. This data is very suitable for time series databases which leads to *InfluxDB* outperforming *MongoDB* by a factor of 27 when inserting. It is also observed that *InfluxDB* needs considerably less disk storage for the data. In terms of reading data, both DBMS performed equally with *MongoDB* getting ahead if concurrency is introduced. [18]

A similar benchmark is conducted for *Elasticsearch*. The data set has 10000 different values updating every 10 s over one day which results in a total of 86.4 million values. For this data, *InfluxDB* is 8 times faster than *Elasticsearch* when inserting it. *InfluxDB* is also faster on querying by a factor of 4 which increases for larger data sets. [17] It is important to note that both papers regarding *InfluxDB* have been released by the developers themselves and are likely to have chosen data sets which are beneficial to *InfluxDB's* architecture.

doi: 10.2313/NET-2017-05-1_06

For *ZeroDB*, there is no quantifiable information on its performance in regard to other databases. However, due to its architecture, it shifts encryption to the client as the database does only handle encrypted data. This way, a single query needs multiple messages, as clients receive an encrypted tree structure, decrypt it and resend a more specific query to get a certain node of the tree. This is repeated until the final value is retrieved. In combination with the Python wrapper for data access from a Java application (see Section 5.2), the DBMS is by design slower than *ZODB* on which it is based on. *ZODB* is able to compete with *PostgreSQL* for very large data stores [4].

6. CONCLUSION

We conclude that the area of NoSQL covers a wide variety of interesting database architectures. In cases like DS2OS where the data is a large set of equally structured data with very little relations, lightweight storing mechanisms like the ones introduced by key-value stores (see Section 3.2.1) or document-oriented databases (see Section 3.2.2) can be used. In our scenario, the data structure can be handled by all categories of databases. By picking popular representatives for each category we are able to narrow down the field and pick suitable candidates for introduction into DS2OS.

Based on their performance and the features they implement in regard to the soft requirements (see Section 4.2), we suggest three candidates worth of further consideration.

PostgreSQL as a more sophisticated and slightly faster alternative to the currently deployed *HSQLDB*. As a RDBMS, it can be introduced with very little changes to the current system.

Redis as a simple key-value store that is very fast on large insert blocks and scalable due to its simple schemaless data storage that inherently supports distributed systems. It is proposed instead of the slightly faster *HSQLDB* as it is more compliant to the soft requirements

InfluxDB as a highly performant solution that implements the versioning as a central concept and looks promising for future improvements as it has barely reached version 1.0 as of now.

7. REFERENCES

[1] Y. Abubakar, T. S. Adeyi, and I. G. Auta. Performance Evaluation of NoSQL Systems using YCSB in a Resource Austere Environment. *International Journal of Applied Information Systems*, 7(8):23–27, September 2014. Published by Foundation of Computer Science, New York, USA.

[2] H. Alzahrani. Evolution of Object-Oriented Database Systems. *GJCST*, pages 37–40, 2016.

[3] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.

[4] R. Compaan. ZODB Benchmarks revisited. `http://www.upfrontsystems.co.za/Members/roche/where-im-calling-from/zodb-benchmarks-revisited`, Mar. 2008. Accessed: 2016-12-17.

[5] C. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.

[6] DB-Engines Ranking. Popularity Ranking Of Database Management Systems. `http://db-engines.com/en/ranking`, Nov. 2016.

[7] C. Hallenbeck. NoSQL, OldSQL, NewSQL, In-Memory & SAP HANA. `https://blogs.saphana.com/2015/05/19/nosql-oldsql-newsql-memory-sap-hana/`, May 2015. Accessed: 2016-11-18.

[8] G. Harrison. *Next Generation Databases: NoSQL, NewSQL, and Big Data*. Apress, 2015.

[9] R. Hecht and S. Jablonski. NoSQL evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341, Dec 2011.

[10] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung*. Oldenbourg, 7 edition, 2009.

[11] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, Feb 2010.

[12] B. Leighton, S. J. D. Cox, N. J. Car, M. P. Stenson, J. Vleeshouwer, and J. Hodge. *A Best of Both Worlds Approach to Complex, Efficient, Time Series Data Delivery*, pages 371–379. Springer International Publishing, Cham, 2015.

[13] B. Leskes. Elasticsearch Versioning Support. `https://www.elastic.co/blog/elasticsearch-versioning-support`, June 2013. Accessed: 2016-12-10.

[14] M.-O. Pahl. *Distributed Smart Space Orchestration*. Dissertation, Technische Universität München, München, 2014.

[15] M.-O. Pahl and G. Carle. Crowdsourced Context-Modeling as Key to Future Smart Spaces. In *Network Operations and Management Symposium 2014 (NOMS 2014)*, May 2014.

[16] M.-O. Pahl, G. Carle, and G. Klinker. Distributed Smart Space Orchestration. In *Network Operations and Management Symposium 2016 (NOMS 2016) – Dissertation Digest*, May 2016.

[17] T. Persen and R. Winslow. Benchmarking InfluxDB vs Elasticsearch for Time-Series. Technical report, InfluxData, Inc., San Francisco, CA, September 2016.

[18] T. Persen and R. Winslow. Benchmarking InfluxDB vs MongoDB for Time-Series Data, Metrics and Management. Technical report, InfluxData, Inc., San Francisco, CA, September 2016.

[19] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[20] S. Tiwari. *Professional NoSQL*. EBL-Schweitzer. Wiley, 2011.

[21] VoltDB, Inc. VoltDB Export – Connecting VoltDB to Other Systems. `https://www.voltdb.com/blog/voltdb-export-connecting-voltdb-to-other-systems`, Aug. 2014. Accessed: 2016-12-11.