A Workbench for Quantitative Comparison of Databases in Multi-Robot Applications

Rubanraj Ravichandran[1], Nico Huebel[2], Sebastian Blumenthal[3], and Erwin Prassler[1]

Abstract— **Robots generate large amounts of data which need to be stored in a meaningful way such that they can be used and interpreted later. Such data can be written into log files, but these files lack the querying features and scaling capabilities of modern databases - especially when dealing with multi-robot systems, where the trade-off between availability and consistency has to be resolved. However, there is a plethora of existing databases, each with its own set of features, but none designed with robotic use cases in mind. This work presents three main contributions: (a) structures for benchmarking scenarios with a focus on networked multi-robot architectures, (b) an extensible workbench for benchmarking databases for different scenarios that makes use of Docker containers and (c) a comparison of existing databases given a set of multi-robot use cases to showcase the usage of the framework. The comparison gives indications for choosing an appropriate database.**

I. INTRODUCTION

With the increase of the availability of cheap sensors, robotic applications produce more and more data. Most robotic applications process the sensor data and then dispose it. However, the trend towards big data shows that storing data for later use can be beneficial. And data from previous executions, so-called episodic memories, stored in databases have also been used in robotics. Niemueller et al. [1] used stored data for fault and performance analysis. Winkler et al. [2] describes how to store and retrieve symbolic plan events and sensor signals together to leverage this information for in future executions while Balint-Benczédi et al. [3] stores perceptual episodic memories and introduces a domain specific language to retrieve perception information and their semantic interpretation, which allows introspection for decisions taken by the robot. There are also cloud-based systems that allow to potentially share data among multiple robots [4], [5].

There are also networked multiple robot systems in which robots have to exchange data to achieve a common goal. This adds complexity and requires design trade-offs between the availability and consistency of the data as described by the CAP theorem [6]. The rise of ubiquitous sensing provided by IoT and Industry 4.0 technologies adds even more data that needs to be exchanged, stored, processed, and archived in a meaningful way.

In robotics these problems are often ignored or solved in an ad-hoc manner. Recently, many databases became available that perform similar tasks while promising better performance as well as additional features. However, they all come with advantages and disadvantages and are not designed with robotic applications in mind. So this work has the following contributions:

- It aims at structuring robotic **benchmarking scenarios** with a focus on networked multi-robot architectures.
- It describes an extensible **workbench** that allows to add new databases and benchmarking scenarios.
- An initial **implementation based on Docker containers**[1] with a set of databases and basic benchmarking scenarios is provided.

Section II describes the classes of databases and some robotic applications that have already applied existing databases. Then Section III describes relevant system architectures for robotics and how they relate to database setups. It also introduces the extensible workbench that can be used to evaluate (robotics) use cases with different databases before Section IV is benchmarking some selected databases on basic robotic use cases. Finally, Section V summarizes and discusses the results.

II. RELATED WORKS

As mentioned before, there are already robotic applications that make use of existing databases.

Dietrich et al. [7] integrated the NoSQL database Cassandra with the Robotic Operating System (ROS) to handle data in smart environments. However, this work compared their results only with two other storage mechanisms.

Fourie et al. [8] applies two databases to Simultaneous Localisation And Mapping. They implemented a two level database on a centralized server, which stored odometry data in a graph database (Neo4j) and larger sensor data like images and point clouds in a key value store (MongoDB). This idea and their result reflect the outcome of our work and can be seen as a more complex benchmarking scenario.

Fiannaca and Huang [9] identify difficulties in handling ROS log messages. They evaluate MongoDB, PostgreSQL, SQLite3 as potential improvements.

Li and Manoharan [10] compared the performance of Couchbase, MongoDB, RavenDB databases with the performance of the SQL database. The comparison is done based on, read, write, delete, and instantiate operations as well as for the performance of iterating over all keys. However, the comparison is limited to a single robot case.

There are also detailed comparison from the database community [11], [12], [13], [14].

[1]Bonn-Rhein-Sieg University of Applied Science, Sankt Augustin, Germany rubanraj.ravichandran@smail.inf.h-brs.de erwin.prassler@h-brs.de

[2]Robotics Research Group, KU Leuven, Belgium, member of Flanders Make nico.huebel@kuleuven.be

[3]Locomotec GmbH, Germany blumenthal@locomotec.com

[1]https://www.docker.com/what-container

III. WORKBENCH FOR BENCHMARKING DATABASES

This section defines a workbench for the comparison of databases with a focus on multi-robot applications. The workbench consists of (a) a distributed system architecture based on existing *container* technology, (b) a set of simulated data sources, and (c) a set of performance criteria to be used for the benchmarks.

The exact choice of data sources and performance criteria depend on the actual application. An example benchmark will be given in the following Section IV. Instructions how to customize the workbench for your problem can be found in the repository mentioned in Section V.

A. *Distributed system architecture*

Collaborative multi-robot applications require the sharing of data among multiple robots. In the face of limited bandwidth and connection losses this can be a difficult task. Thus, the system architecture need to be robust against the fragility of the network.

In the following we discuss three typical database architectures and propose the most promising one to serve as distributed system architecture for robotic use cases: **master-slave**, **centralized master-master** and **decentralized master-master**.

1) Master-slave architecture: Figure 1 shows the master-slave setup. The robots are represented by multiple clients, which can generate (sensor) data with varying frequencies. The generated data is sent to a master database, which is deployed centrally (e.g., a central server).

There can be replicas on further *slave* databases that copy the data from the master. While the robots have to send the data to the master database, they can read data also from the slave databases.

The two main problems with this setup are that (a) in case of a network failure, the robots can neither update nor access the data and that (b) the single master database can get congested if too much data is generated by the robots.

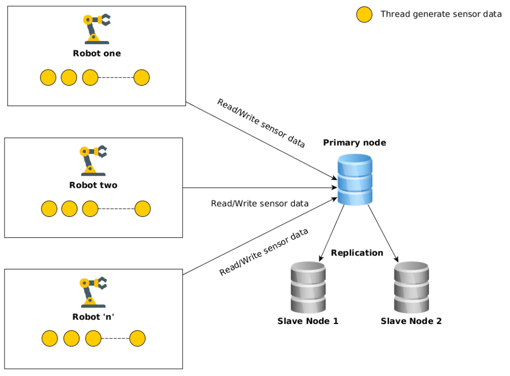

Fig. 1: Master-slave setup of databases

2) Centralized master-master architecture: Figure 2 shows the centralized master-master setup. All master nodes are (centrally) deployed on servers accessible to the robots. The difference to the previous setup is the absence of slave nodes. Instead, all data nodes act as master node. Thus, robots can write/read data to any database node.

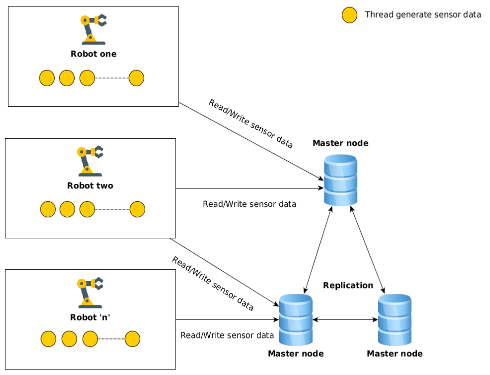

Fig. 2: Centralized master-master architecture. All database nodes are deployed centrally, e.g. a web server.

The master-master replication setup, addresses one of the problems from master-slave architectures: Here, write/read can be distributed to all master nodes. Hence, we can reduce the chance of data loss, caused by congestion. However, this setup is not robust against network failures and resolving conflicting data needs to be addressed.

3) Decentralized master-master architecture: Figure 3 shows the decentralized master-master setup with a database deployed on each robot. In this setup, each robot stores the data in the master node available locally. The data will be replicated automatically among the robots. Here, the shortcomings of both previous architectures are addressed. Even if there is a problem with the network, each robot has the relevant data available locally and can share its updates with other robots once the network connectivity is recovered.

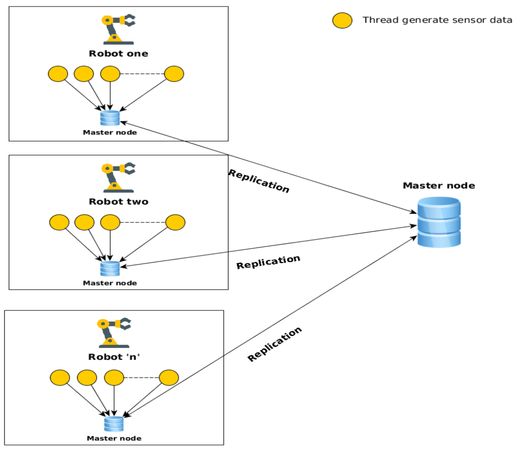

Fig. 3: Decentralized master-master setup. The databases are deployed on the robots in order to foster availability of data in case of network failure.

Each architecture can address the needs of certain scenarios in robotics. Since the focus of this work is on multi-robot systems, where the availability of data is crucial, the decentralized master-master architecture was selected for the quantitative analysis in the following section. Note that not all database support this setup. This can be considered as a *qualitative criteria* for the selection of a database. For the workbench each node is encapsulated into dedicated container. As container technology *Docker* was chosen.

4) Realization with Docker containers: The workbench for a decentralized master-master architecture is realized with **Docker** containers. Docker is an open source platform for developing, shipping and running applications. A container is an isolated component run by a so-called docker engine. Multiple containers can be connected and communicate with each other within an isolated, virtual network. They can be deployed on a single machine. This allows to simulate a multi-robot scenario on a single PC.

We have chosen Docker for two main reasons:

1) Scientific experiments should be **reproducible** [15]. Once, an experiment is containerized, it can be run on any hardware that runs the container engine.
2) The workbench should be **extensible** with other databases and new benchmarking scenarios. The presented scenarios are running examples that can downloaded, tested, and modified. In addition, many databases offer already a Docker container that can be integrated into the workbench.

Each Docker container represents a single robot with a database and a simulated data source. The complete setup is is composed from these containers using a *Docker-Compose* configuration file. The examples presented in the following section are available for download[2].

B. Data sources

Each Docker container represents a single robot with randomly generated sensor events. Each data source is realized by a dedicated thread. The threads are using a blocking communication pattern: once they send data to a database, they will wait for a response from the database and then generate next sensor event (according to the configured frequency). Of course, this can compromise writing data at high frequencies, but that can be monitored.

In the examples a set of data types were generated that are common in robotics

- Location event - GPS location of a robot as longitude and latitude
- Ultrasonic sensor event - Distance value
- Motor voltage and current - Voltage, current supplied to motor
- Pose event - Current pose of robot as position and quaternion vector
- RGB Event - RGB image (as binary BLOB or as link to the file system)

Some data is similar with respect to the amount of data in the sense that only a few fields need to be stored, while the RGB images are using large (>=10MB) "blob" files. Such blob files are commonly seen in robotics but pose a challenge to existing databases as seen in the Section IV.

[2]https://github.com/rubanraj54/research-and-developement

C. Performance criteria

For the quantitative benchmarks three performance categories are considered:

1) Write performance of the simulated sensor data. Furthermore, writing while performing read queries at the same time are considered since this gets close to a real-time interaction on a robot. As benchmarked variable the average write time t_{write} is defined.
2) Read performance of the example queries below. Further, the behaviours is tested while new data is written. As benchmarked variable the average read query execution time t_{read} is defined.
   - Get RGB events (without blob) for last 10 seconds
   - Get RGB events (with blob) for last 10 seconds
   - Get first 10 Pose events generated today
   - Get all location events within a bounding box as defined by latitude and longitude
3) Network failures between the Docker containers are injected by shutting down the virtual network interfaces used by the Docker engine. After turning them on again, the average replication time $t_{\sigma\_replication}$ is measured.

IV. BENCHMARKING DATABASES

This section applies the proposed workbench (cf. section III) to a set of databases. The goal is to gain insights into which databases are good candidates for multi-robot scenarios.

Our **hypothesis** h_0 is: *there exists a single database suitable for multi-robot scenarios*. It means it is superior in the categories read, write and fail-over tolerance performance.

In order to make the quantitative analysis feasible, only some databases could be evaluated. In the following qualitative selection criteria are motivated.

A. Requirements for selecting database

Databases for quantitative analysis have been selected based on the following criteria and detailed justifications for each chosen databases will be explained afterwards.

- Types of interfaces to database and number of supported languages
- Querying capabilities
- Suitability for multi-robot scenarios
- High availability
- Dynamic consistency configuration
- Storage type and caching mechanism
- Handling of blob data

For handling large **blob** data like point clouds or images (approx. 10 MB and more), there are two approaches:

1) Store point clouds/images as binary blobs in the database itself.
2) Store blob data in file system and store the reference in the database. However, the disadvantage is that the data cannot be replicated automatically by the database.

B. Selected databases

For a fair comparison at least one database from each category of underlying storage data model was picked: graph database, document store, wide column, multi-model and time series.

1) Graph database: **Neo4J** was chosen because it focuses more on high availability compared to other graph databases. It provides multiple interface (Cypher query, REST, HTTP, TinkerPop 3, etc.) to access data. Neo4j also supports more than ten languages including Java, Python, and Scala. Other graph databases have support for only few languages. Neo4J has been already used for performance evaluation with mobile robots, but in a single robot scenario. In terms of handling blob files, Neo4j supports byte-array type to store large BLOBS. However, developers of Neo4J consider storing blob data in database as an anti pattern[3].

2) Document store: For the document store category Apache CouchDB and MongoDB were chosen.

The reasons for choosing **CouchDB** over other document store databases are the following:

- It supports a simple HTTP protocol to access the data.
- CouchDB supports more languages including C, Pyhton, and Java.
- It allows to treat data as *immutable*. This helps to add new data while staying consistent and deletes the old data based on revision numbers during the so-called compaction process.

CouchDB has also some unique features:

- Latency: Sync is designed to increase the partition tolerance. If there are multiple nodes located in different places around the world, Sync chooses the nearest node and store data in it. Later a system will synchronize the data with all other nodes.
- Network failure: In this scenario PouchDB is deployed on a local machine to cache all the data. Once the machine re-connects to the Internet, PouchDB synchronizes the local data with CouchDB nodes.

MongoDB is a document store like CouchDB and it has similar features like schema less data modelling, eventual consistency, high availability and partition tolerance. Moreover, MongoDB can be configured with different storage engine options. In contrast to CouchDB, MongoDB offers geo-spatial queries out of the box. MongoDB is a key player in document store databases and has been previously used in robotic projects and database evaluations.

Other document store databases (e.g., Couchbase) provide some similar features but lack in others (like HTTP access, Sync, language support, easy configuration) and community support.

3) Wide column store: For the wide column store database category, Accumulo, HBase, and Cassandra were considered for closer investigation. All three databases make similar trade-offs w.r.t. consistency, availability, and partition tolerance. Furthermore, they are comparable in memory caching, scalability and immutability. However, **Apache Cassandra** was chosen because it has more language support, is easy to configure, and its independent data storage.Also, the ROS (Robotic Operating System) has an existing library called *cassandra_ros* which will be useful for interfacing with a Cassandra database.

4) Multi model database: Recently, a new paradigm is evolving called *multi model databases*. It means that a single database supports key value pairs, document stores and graph data models. From this multi model database category, **OrientDB** and **ArangoDB** were chosen. There are many similarities in features between OrientDB and ArangoDB, but in the aspect of storing blob files OrientDB has additional mechanisms to handle huge files compared to ArangoDB. ArangoDB does not have native support to handle blobs but it can be extended by using *Foxx micro services*. Using Foxx micro services allows to store files in the file system (not in database) and save the reference in database. OrientDB provides multiple options out of the box to handle blob data:

1) Store the blob data in the file system and store the reference in a database document.
2) Store the blob data (up to 10MB) in the database itself, but with the risk of degrading performance, run time cost, and waste of space + 33% [4].
3) Store blob data via *ORecordBytes*. It is a special class designed to handle large blobs without additional conversions.

5) Time series database: Time series databases are meant to store streaming information from sensors, monitors, etc., along with a time stamp. In most cases they store numeric values. SiriDB, InfluxDB and Druid were considered.

Out of these three databases, SiriDB and Druid were neglected for quantitative comparison because SiriDB is a new database lacking some features (like support programming languages like C, C++ and Java, community support and documentation). Druid and InfluxDB provide mostly similar features in terms of consistency, availability, schema less design and immutability. However, Druid supports less programming language interfaces than InfluxDB. Futhermore, InfluxDB has a *Time Structured Merge Tree* engine that improves high speed ingestion and data compression. So **InfluxDB** was chosen for quantitative analysis from the time series databases.

6) Relational database: For a relational database **MySQL** was picked over other familiar SQL databases for e.g. MariaDB and PostgreSQL, because MariaDB is forked from MySQL and both of them likely to have similar features like ACID property, replication methods, data durability and concurrency [16]. Fiannaca and Huang [9] analyzed the performance of PostgreSQL and SQlite against monogoDB but MySQL was not included in this work. Hence we concluded to use MySQL in our research work to check whether MySQL is performing better than other databases.

[3]`https://neo4j.com/blog/dark-side-neo4j-worst-practices/`

[4]http://orientdb.com/docs/last/Binary-Data.html

C. Experimental setup

The experimental evaluation was carried out on a laptop (8GB RAM, 256GB SSD drive, 7th Generation Intel i5). The Docker configuration files available on the project repository (see above) include the versions of the database. Three robots/containers were used for the analysis due to the limitations of the used hardware. As frequencies f_{write} 30Hz, 60Hz and 120Hz were chosen. 30Hz represents common frequencies for, e.g., camera data. The values were doubled to be able to discover trends with increasing frequency.

D. Benchmarking results

The workbench has been applied to the selected database candidates Neo4J, Apache CouchDB, MongoDB, Apache Cassandra, OrientDB, ArangoDB, InfluxDB and MySQL. The results can be found in the tables I to VI.

1) Write only and Write with read: An observation of tables I and II is that Neo4j's write query execution time reduces constantly if the frequency increases (except for RGB images with blob data - in RGB images with blob data, there is a rise in time at 60 Hz and again drop at 120 Hz). Other databases behave as expected: the write query execution time increases proportional to frequency of generating events. Neo4j and OrientDB perform worse compared to all other databases. But surprisingly, OrientDB handles BLOB files better than Neo4J.

It turns out, writing (a) images as complete blob data is on average 200 times slower than (b) storing the image as a file and keeping a pointer to it in the database.

For example, the Neo4j database takes 0.063s to store a pointer reference in the database at 120 Hz while 0.541s to store the blob data. But even for storing blob data directly into MongoDB and ArangoDB took only 0.01 and 0.0461 seconds. This shows that MongoDB and ArangoDB are more efficient in storing blob files directly to the database compared to other candidates.

In most cases, Cassandra, OrientDB and MySQL databases show comparable write timings given the three different frequencies. ArangoDB's write performance is constant until 60 Hz but at 120 Hz it shows double the write time in all events. Still write timings are smaller than other databases. As a overall observation, MongoDB and ArangoDB perform good in case of writing all three events under different frequencies. Please note, MongoDB supports only master-slave architectures, so there is no replication workload on MongoDB master node, but ArangoDB is tested in master-master mode. Apart from these two databases, Cassandra, OrientDB and MySQL are in the same range and perform better than Neo4j and CouchDB.

2) Read only and Read with write: CouchDB shows stable and good read query execution time as seen in tables III to V. It is better than all other databases in all four scenarios. The reason is, CouchDB uses B-trees to generate *key sorted views* that is built once and available to clients for efficient lookup. If clients add, delete, or update a document, then B-tree will be indexed automatically and reflect the current state of database.

We can divide the performance of the tested databases into two sets: The first set (ArangoDB, OrientDB, InfluxDB, and MySQL) took significantly longer for executing queries compared to the second set of databases (Neo4j, CouchDB, MongoDB, and Cassandra). In all four query scenarios the second set of databases executes read query much faster than the first set.

To illustrate the difference: In the query scenario 1 (Get RGB events (without blob) for the last 10 seconds), the best performance from the first set of databases is only 0.025 seconds by ArangoDB, while the worst performance from the second set is just 0.0013 seconds by Cassandra. databases.

Another interesting observation drawn from the majority of the databases is, that the time for reading data along **with** writing is double the time faster than only reading data. We hypothesize, the databases cache recently written data and while reading them, the queries read the data from those caches. This improves the over all read query performance. Still this is not the case for in all databases. MySQL and OrientDB do not show this behaviour.

The MySQL database is slower (on average up to 6 seconds) than others for complex queries to retrieve blob data. For simpler queries there is no significant difference.

As a conclusion, CouchDB and MongoDB read times are faster compared to other databases. Followed by Neo4j and Cassandra. ArangoDB, InfluxDB, OrientDB and MySQL databases consume more query execution time in case of read and read with write scenarios compared with all four databases from second set of performers.

3) Average replication time: Since InfluxDB community edition does not support cluster architectures, so it has been has been excluded for the replication test.

As indicated in table VI, the replication time of MongoDB looks impressive, but since it supports only master-slave architectures it has less applicability for multi-robot scenarios. The second good performer is MySQL (0.0715 seconds), but the performance for storing and retrieving blob data is poor. In case of an application where blob files do not need to be stored or shared, MySQL is a good candidate . Neo4j, ArangoDB, OrientDB consum nearly equal time to replicate the data. In this group of databases, ArangoDB is a good performer in handling blob files as well as write/read operations compared to Neo4j and OrientDB. So, after MySQL, ArangoDB will be a good candidate for replication. Finally, CouchDB and Cassandra took nearly same time to replicate data. Even though, these two databases performed well in handling blob files and write/read operations, they performed poor in replication test.

Overall, we can say that ArangoDB, MongoDB and CouchDB will be more flexible candidates for multi robot systems to handle sensor data. However there is no clear winner in all performance categories. Given the results from our workbench and the tested candidates, we do not find enough evidence to support our **hypothesis** h_0. Thus, we derive, **there is no single database suitable for multi-robot scenarios**.

	neo4j	orientdb	couchdb	mongodb	cassandra	arangodb	influxdb	mysql
write(30hz)	0.0874	0.0069	0.0788	0.0034	0.0050	**0.0016**	0.0113	0.0115
write_with_read(30hz)	0.0841	0.0069	0.0857	**0.0027**	0.0101	0.0257	0.0264	0.0129
write(60hz)	0.0812	0.0094	0.0797	0.0035	0.0040	**0.0017**	0.0115	0.0133
write_with_read(60hz)	0.0828	0.0098	0.0877	0.0044	0.0117	**0.0019**	0.0332	0.0142
write(120hz)	0.0715	0.0104	0.0833	**0.0037**	0.0111	0.0126	0.0067	0.0129
write_with_read(120hz)	0.0630	0.0089	0.0952	**0.0021**	0.0241	0.0157	0.0428	0.0160

TABLE I: Average write execution timings in s for RGB image references (without blob data). Note, the results are similar for the other data sources: pose, location, ultrasonic data or motor voltage - thus they are not shown in the paper. The complete results are available on the project repository.

	neo4j	orientdb	couchdb	mongodb	cassandra	arangodb	influxdb	mysql
write(30hz)	0.5494	0.0344	0.1471	0.0218	0.0353	**0.0100**	0.0371	0.0870
write_with_read(30hz)	0.4613	0.0343	0.1564	**0.0156**	0.0841	0.0763	0.0697	0.0617
write(60hz)	0.5804	0.0453	0.1526	0.0150	0.0395	**0.0130**	0.0304	0.0601
write_with_read(60hz)	0.6069	0.0483	0.1723	**0.0122**	0.0591	0.0132	0.0797	0.0623
write(120hz)	0.5163	0.0718	0.1640	**0.0133**	0.0530	0.0380	0.0259	0.0554
write_with_read(120hz)	0.5410	0.0765	0.1833	**0.0105**	0.0593	0.0461	0.1003	0.0641

TABLE II: Average write execution timings in s for RGB images with blob data

	neo4j	orientdb	couchdb	mongodb	cassandra	arangodb	influxdb	mysql
read(30hz)	0.0027	2.4472	**0.0002**	0.0014	0.0041	0.9106	1.5382	2.9981
read_with_write(30hz)	0.0011	2.5056	**0.0001**	0.0009	0.0009	0.1264	2.0431	4.2811
read(60hz)	0.0022	2.4387	**0.0002**	0.0010	0.0045	0.9111	1.7041	3.1920
read_with_write(60hz)	0.0009	3.4057	**0.0001**	0.0008	0.0009	0.0259	2.8656	5.1232
read(120hz)	0.0013	2.4607	**0.0002**	0.0007	0.0048	0.9209	1.7285	3.4621
read_with_write(120hz)	0.0012	3.1530	**0.0001**	0.0005	0.0014	0.0700	3.6152	9.8847

TABLE III: Average read query execution timings in s to get RGB images (without blobs) for the last 10 seconds

	neo4j	orientdb	couchdb	mongodb	cassandra	arangodb	influxdb	mysql
read(30hz)	0.0027	2.2152	**0.0002**	0.0014	0.0041	0.9205	1.5291	2.9967
read_with_write(30hz)	0.0009	2.2788	**0.0001**	0.0009	0.0009	0.1264	2.4808	3.7208
read(60hz)	0.0022	2.2299	**0.0002**	0.0010	0.0044	0.9113	1.7215	3.2154
read_with_write(60hz)	0.0012	3.1863	**0.0001**	0.0008	0.0009	0.0259	2.8464	5.1120
read(120hz)	0.0013	2.2410	**0.0002**	0.0007	0.0048	0.9261	1.7187	3.4758
read_with_write(120hz)	0.0013	2.9127	**0.0001**	0.0005	0.0014	0.0699	3.4412	8.9265

TABLE IV: Average read query execution timings in s to get RGB images (with blob) for the last 10 seconds

	neo4j	orientdb	couchdb	mongodb	cassandra	arangodb	influxdb	mysql
read(30hz)	0.0019	2.9490	**0.0002**	0.0011	0.0028	1.3262	2.2636	3.4127
read_with_write(30hz)	0.0008	4.8820	**0.0001**	0.0007	0.0006	0.6426	3.5922	4.7775
read(60hz)	0.0016	2.9402	**0.0002**	0.0007	0.0030	1.2658	2.5796	4.5453
read_with_write(60hz)	0.0007	3.9180	**0.0001**	0.0006	0.0006	0.0692	6.3047	6.1391
read(120hz)	0.0009	2.9240	**0.0002**	0.0005	0.0031	1.2705	2.5908	4.7684
read_with_write(120hz)	0.0007	3.6608	**0.0001**	0.0004	0.0008	0.4061	6.8796	10.9272

TABLE V: Average read query execution timings in s to get all location events within a bounding box as defined by latitude and longitude

	Average data replication time in s
neo4j	0.1918
orientdb	0.2378
couchdb	0.4733
mongodb	**0.0472**
cassandra	0.4904
arangodb	0.2372
mysql	0.0715

TABLE VI: Average replication time in s

V. Conclusion

This work presents initial insights in usability of existing databases for robotic scenarios with a focus on multi-robot systems.

Different database architectures have been discussed that can fit the needs of robotic scenarios. However, the focus of this discussion was on networked multi-robot systems. For these systems the *decentralized master-master* architecture seems to be the most appropriate. The local database node on each robot fosters availability of the data, which is in most robotic scenarios more important than having consistent data.

An extensible workbench for benchmarking databases for robotic use-cases is introduced. It is based on Docker containers to ease extensibility and improve reproducibility of the performed experiments. Example scenarios with a set of databases are provided as an initial comparison. Such scenarios can be vastly different in robotics and, therefore, the intention is to allow easy replacement with user specified scenarios.

Several databases have been discussed and were quantitatively evaluated. The results show that MongoDB and ArangoDB performed well compared to the other databases. As a best practice, it is suggested to store blob data files in the file system, along with references in the database, even though MongoDB and ArangoDB can handle blob files. Unfortunately, MongoDB does not support the decentralized master-master replication. CouchDB showed lesser and stable query execution time for reading and, while it performed ordinary in write operations. So if an application requires more reading than writing, it is a good choice. Further results and the used schemas can be found in the repository of the project[5].

Of course, evaluation of databases spans further dimensions then the quantitative comparison. Our findings here are that most databases offer bindings for commonly seen programming languages including C/C++, Python and Java. However, they differ a lot in further aspects like data modelling, redundancy treatment, scalability or consistency. Therefore, we can only give an idea about these properties and provide a framework to test a particular application instead.

Note that some databases provide dedicated tools to tune them for application dependent (frequent) queries. Such tuning has not been fully explored.

Finally, it should be mentioned that the results can change for different queries and data models. So the presented quantitative results are only indicative. This is one of the reasons for providing this workbench: to be able to test such changes and compare multiple databases rather quickly for new robotic applications.

In the future, extensions for network simulations, like bandwidth limitations, latency, or transmission errors, are planned. This can be added to the workbench by adapting the docker compose files or using tools like Docker Swarm or Kubernetes. Since there is a plethora of models and various connection types (like WiFi, radio, or LTE to name just a few) appropriate choices will be required to come up with a set of examples that can then be extended by the community.

[5]`https://github.com/rubanraj54/research-and-developement`

ACKNOWLEDGMENT

This work was supported by the European Union's Horizon 2020 projects ROPOD (grant agreement No 731848) and RobMoSys (grant agreement No 732410).

References

[1] T. Niemueller, G. Lakemeyer, and S. S. Srinivasa, "A generic robot database and its application in fault analysis and performance evaluation," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 364–369.

[2] J. Winkler, M. Tenorth, A. Bozcuoglu, and M. Beetz, "Cramm–memories for robots performing everyday manipulation activities," *Advances in Cognitive Systems*, vol. 3, pp. 47–66, 2014.

[3] F. Balint-Benczédi, Z.-C. Márton, M. Burner, and M. Beetz, "Storing and retrieving perceptual episodic memories for long-term manipulation tasks," in *Advanced Robotics (ICAR), 2017 18th International Conference on*. IEEE, 2017, pp. 25–31.

[4] M. Tenorth, J. Winkler, D. Beßler, and M. Beetz, "Open-ease: A cloud-based knowledge service for autonomous learning," *KI-Künstliche Intelligenz*, vol. 29, no. 4, pp. 407–411, 2015.

[5] T. Niemueller, S. Schiffer, G. Lakemeyer, and S. Rezapour-Lakani, "Life-long learning perception using cloud database technology," in *Proc. IROS Workshop on Cloud Robotics*. Citeseer, 2013.

[6] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, Feb 2012.

[7] A. Dietrich, S. Mohammad, S. Zug, and J. Kaiser, "Ros meets cassandra: Data management in smart environments with nosql," in *Proc. of the 11th Intl. Baltic Conference (Baltic DB&IS)*. Citeseer, 2014.

[8] D. Fourie, S. Claassens, S. Pillai, R. Mata, and J. Leonard, "Slamindb: Centralized graph databases for mobile robotics," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 6331–6337.

[9] A. J. Fiannaca and J. Huang, "Benchmarking of relational and nosql databases to determine constraints for querying robot execution logs," *Computer Science & Engineering, University of Washington, USA*, pp. 1–8, 2015.

[10] 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.

[11] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah, "A comparative study: Mongodb vs. mysql," in *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*. IEEE, 2015, pp. 1–6.

[12] 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 *Database Theory and Application (DTA), 2015 8th International Conference on*. IEEE, 2015, pp. 14–17.

[13] V. Abramova and J. Bernardino, "Nosql databases: Mongodb vs cassandra," in *Proceedings of the international C* conference on computer science and software engineering*. ACM, 2013, pp. 14–22.

[14] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 195–204.

[15] C. Boettiger, "An introduction to docker for reproducible research, with examples from the R environment," *CoRR*, vol. abs/1410.0846, 2014. [Online]. Available: http://arxiv.org/abs/1410.0846

[16] K. Gupta, "Mariadb vs. mysql vs. postgresql in-depth comparison," 2017, [Online; accessed 30-December-2017]. [Online]. Available: https://www.freelancinggig.com/blog/2017/07/22/mariadb-vs-mysql-vs-postgresql-depth-comparison/