Report on R&D
Performance analysis and benchmarking of different databases for robotic applications

Rubanraj Ravichandran[*]
Matrikel Nr.: 9029355

Master Studies in Autonomous Systems

Bonn-Rhein-Sieg University of Applied Sciences

Advisors:
Prof. Dr. Manfred Kaul[**]
Nico Huebel[††]
Sebastian Blumenthal[‡‡]

January 22, 2018

[*] rubanraj.ravichandran@smail.inf.h-brs.de
[**] Manfred.Kaul@h-brs.de
[††] nico.huebel@kuleuven.be
[‡‡] blumenthal@locomotec.com

Declaration

I, **Rubanraj Ravichandran**, confirm that the research work titled: **"Performance analysis and benchmarking of different databases for robotic applications"** was solely carried out by myself. Any reference to work done by any other person or institution or any material obtained from other sources have been duly cited and referenced.

Date	Rubanraj Ravichandran

Abstract

Robotic applications generate huge amount of data from sensors and sometimes data are stored locally in the robot and most of the times data has been discarded after extracting meaningful information from it. People in the robotic community store these sensor data in the form of text files, log files or ROS bags and use them in future for navigation, manipulation, fault diagnosis, etc,. But there are potential disadvantages in storing data locally and, the way data are stored and queried.

To address this issue, we could opt the existing solutions from available databases to handle robot sensor data. In this research work, a set of databases will be selected from different categories like RDBMS, NoSQL, Graph, and Column oriented, and qualitative and quantitative analysis will be conducted under different configurable test cases. The results from this research work will give the best practices for when to use which database, which data model should be considered and, insights of best database architectures, replication strategies and conflict resolution policies.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Modern robotics applications consume and generate already lots of data. However, this will further increase with the rise of the ubiquitous sensing provided by IoT and Industry 4.0 technologies. Most of the time, perceived data from sensors has been processed for decision making and disposed after the use. In the future, applications will require the sharing of data for the coordination of fleets of robots among themselves and with sensors in their environment. Typically, these data comes from different sources and requires different treatment. It can roughly be split into two categories:

- Streams of raw sensor data: This data is typically produced at high frequency and can be small (like the "tick" of an encoder) or large (like a point cloud produced by a laser scanner).
- Meta-data: This is connecting different types of sensor data together with the information and knowledge required to interpret them (like which robot has produced that data using which sensor with which settings at which position, applying which algorithm while performing what task). E.g., a point cloud of a laser scanner is not very useful if it is unknown from which position it was taken with which type of laser scanner and which settings.

As mentioned in these papers [1, 2, 3, 4], databases used in robotics for logging system, SLAM, storing sensor data for fault analysis and performance evaluation. But, the questions are which database is fit for which task, how efficiently we can use Relational database, NoSQL[5], Graph database[6], Multi modal database for different tasks in robotics. This project focuses on qualitative analysis of a reasonable subset of currently available databases and benchmark their performance metrics under different scenarios.

1.2 Structure

- Section 2 briefly describes the background about the topics which are relevant to this work.
- Section 3 shows the related work which has been done earlier in the field of database in robotic applications and also highlights the mechanism they followed to evaluate databases.

- Section 4 and 5 discuss about problem formulation for this research work and how we are going to approach the performance evaluation.
- Section 6 includes detailed feature analysis, advantages and disadvantages of databases which are selected for Qualitative analysis.
- Section 7 discuss about how and why the databases selected for Quantitative analysis, and section 8 and 9 shows the test environment and results comparing databases under different scenarios.

2 Background

This section gives brief description/definitions of concepts relevant for this research work with respect to distributed databases.

2.1 Data model

A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of the real world entities. For instance, a data model may specify that the data element representing a car be composed of a number of other elements which, in turn, represent the color and size of the car and define its owner.[7]

In the context of DBMS, data model represents how the data need to be stored and accessed from the database. Every database designed with their own data models and users can adopt any of these data model according to the application requirement.

2.1.1 Type of data models

- Relational data model - Data stored in rows and columns in the form of tables.
- Graph data model - Data stored in the form of tree like structure.
- Document data model - Data stored as documents or semi-structured data.
- Column family data model - Data stored in columns and similar columns category can be grouped into column family.
- Multi model - Database that supports more than one data model (E.g, Graph, Document, Relational).

These data models are popular in terms of flexibility and most used in new generation databases, and there are also other well-known database models available, but less frequently used are Hierarchical model, Network model, Object-oriented database model, Object-relational model, Entity-relationship model.

2.2 Database schema

Database schema is a structure described in a formal language supported by the database management system (DBMS). The term ”schema” refers to the organization of data as a blueprint of how the database is constructed.[8]

2.3 Query Language

Query language is a mechanism to read/access stored data from the database. Each database should be provided with their own query language with additional data filtering capabilities. For example, Mysql database have SQL (Structured Query Language), Cassandra have CQL (Cassandra Query Language), ArangoDB have AQL (Arango Query Language).

2.4 Distributed database

In distributed database, all elements in a single database is replicated in multiple database in different locations. Even if one database fails, we can still access our data from other available database.

Advantages

- High availability.
- Scalability

Disadvantages

- High overhead in making changes in all database instance
- High cost in terms of buying additional servers

2.4.1 Availability

Availability is a dominant property in distributed database which represents, whenever an user request for a data, database should be always available to serve the request quickly without fail.

2.4.2 Scalability

Scalability means, a system should be capable of handling the varying traffic workload by enhancing its resources (CPU core, RAM, Memory) or adding additional systems to split the workload. We can scale distributed systems in two ways, vertical and horizontal scaling.

Vertical scalability

In vertical scaling, we increase hardware resources in a single database server to compensate the current traffic. But this will not be an optimal solution always, because if the workload reaches critical point there is a chance that database might fail and no users can write/read data into database. This scenario is called single point of failure.

Horizontal scalability

In horizontal scaling, we add multiple database instance and distribute data equally among them and route incoming write/read requests to different database using load balancer. But the important thing is, the data should be consistent in all the databases and by doing this we can achieve 100% consistency.

2.5 Immutable and Mutable objects

In object-oriented and functional programming, an immutable object (unchangeable object) is an object whose state cannot be modified after it is created. This is in contrast to a mutable object (changeable object), which can be modified after it is created. [9]

Choosing between immutable and mutable objects are completely dependent on the application. Let consider two different scenarios, first scenario with more frequently updating data in the database and second one with few/no updation of data. For scenario one, mutable objects are best because instead of creating new memory space during every update, change the value in the existing memory address. For scenario two, immutable objects are best because it will not allow the user to change the existing value in the memory, but instead store the updated value in new memory address and delete the old memory space. Immutable objects are strongly suggested to use to achieve high performance and data consistency.

2.6 Replication

Replication is a process of creating multiple copies of same data and storing them in different database instances to ensure the data availability and improve the system fault tolerant state. If same data is available in multiple geographical locations, users can query data from their nearest location and, it will minimize the network traffic and latency.

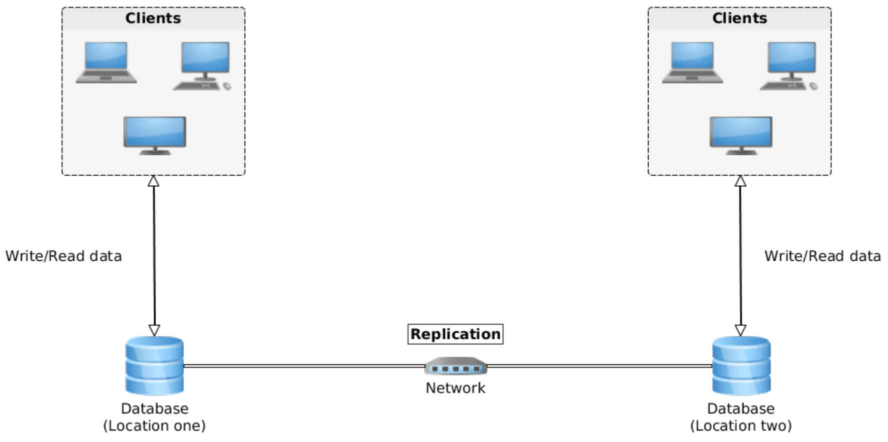


Figure 1: Generic Replication Architecture

2.6.1 Master-Slave Architecture

In master slave architecture Figure 2, one database instance will act as a primary (master) node and all other database instances will act as a secondary (slave) nodes. Primary node can handle write and read requests, but slave nodes are restricted to accept write/update requests. It shows that, slave nodes are considered as backup nodes and if we have more slave nodes we can reduce the workload in master node.

2.6.2 Master-Master Architecture

In master master architecture Figure 3, all data nodes in the cluster will act as a master. Masters can be present in different physical location and any master can serve the client with its updated data. The advantages of using multi master replication are, high availability and fast server processing time. The disadvantages of multi master replication are, latency issues, loosely consistent (lazy and asynchronous) and violating ACID properties.[10]

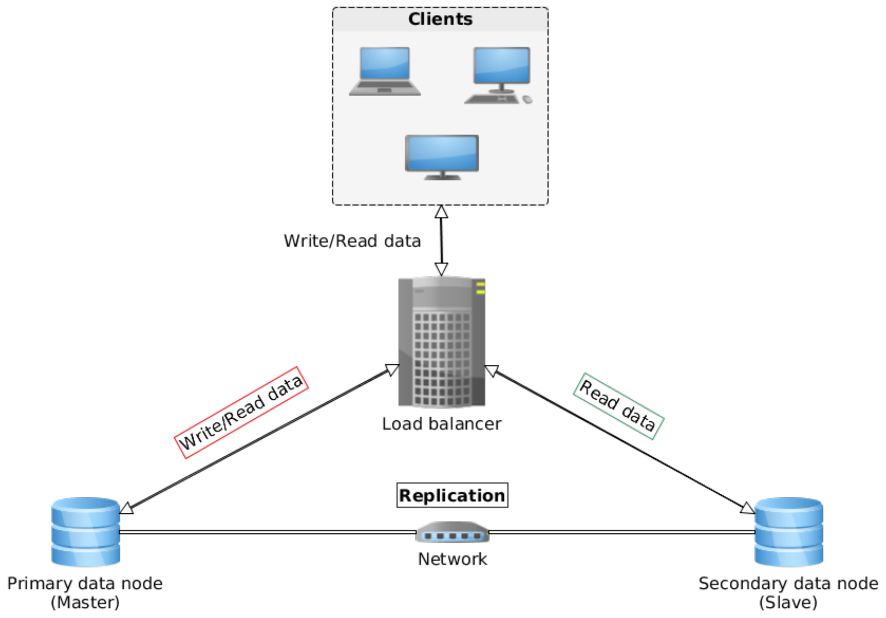


Figure 2: Master-Slave Replication Architecture

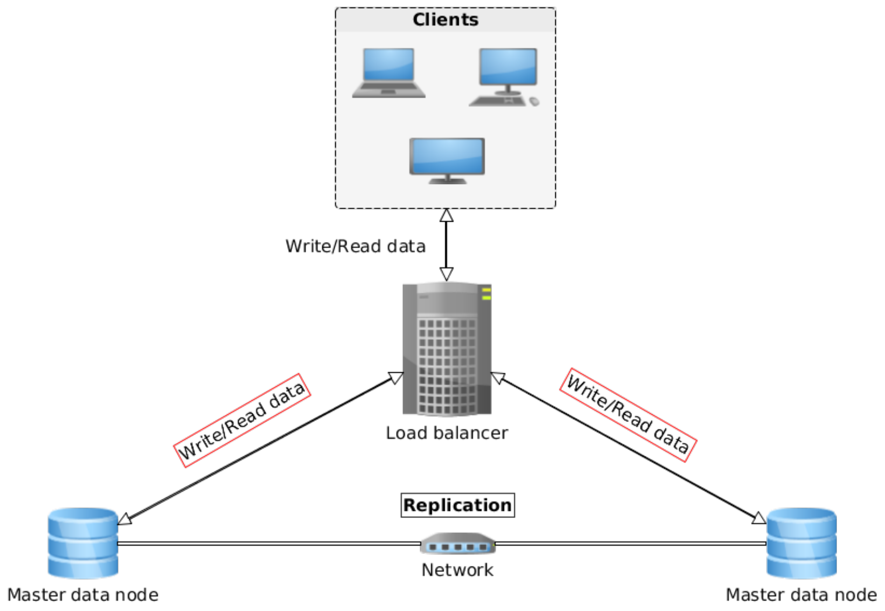


Figure 3: Master-Master Replication Architecture

2.7 CAP theorem

CAP theorem also called as Brewer's theorem named after computer scientist Eric Brewer[11], states that all distributed database systems can only provide two out of three properties which are shown in figure 4.

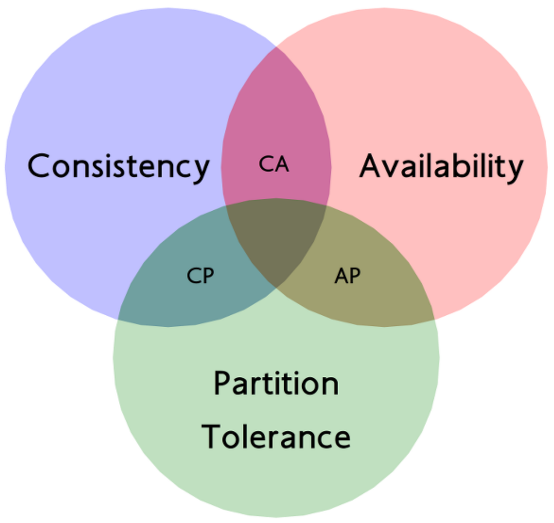


Figure 4: CAP Theorem [12]

Consistency

Whenever user reads data from any database from the cluster, user will get the same version of data as a result. Consistency is achieved by updating several nodes before allowing further reads.

Availability

All nodes in the distributed system should answer for user requests(write as well as read). Availability is achieved by replicating the data across different data node instances.

Partition tolerance

Here partition does not mean the disk/data partitioning, instead it means connection between the nodes partitioned (or disconnected) due to certain reasons like network failure, poor configuration, cable problem, etc.,

Distributed systems are somehow partitioned because of the problems which is mentioned above, but not so often. So, developers should keep partition tolerance in mind before developing systems. In partition tolerance mode, the system should choose either availability or consistency. How to pick anyone of these? It is simple, it depends on what our application required. If our application requires high availability, then the system should allow reads before updating all nodes and if application needs strong consistency, then the system should lock all nodes before allowing reads.

2.8 Docker

Docker is an open source platform for developing, shipping and running applications. [13] Important components of docker are, images, and containers. Images are nothing but a software package or operating system like Ubuntu, Nginx, etc., Using docker platform, one can quickly combine images and necessary software packages to build a container. Container is an isolated component runs individually in the docker engine, and multiple containers can be connected and communicated with each other in the same network. Docker engine shares the host machines resources (CPU,Memory) with the running containers. We can write a new image file on top of other base image file to build our customized containers.

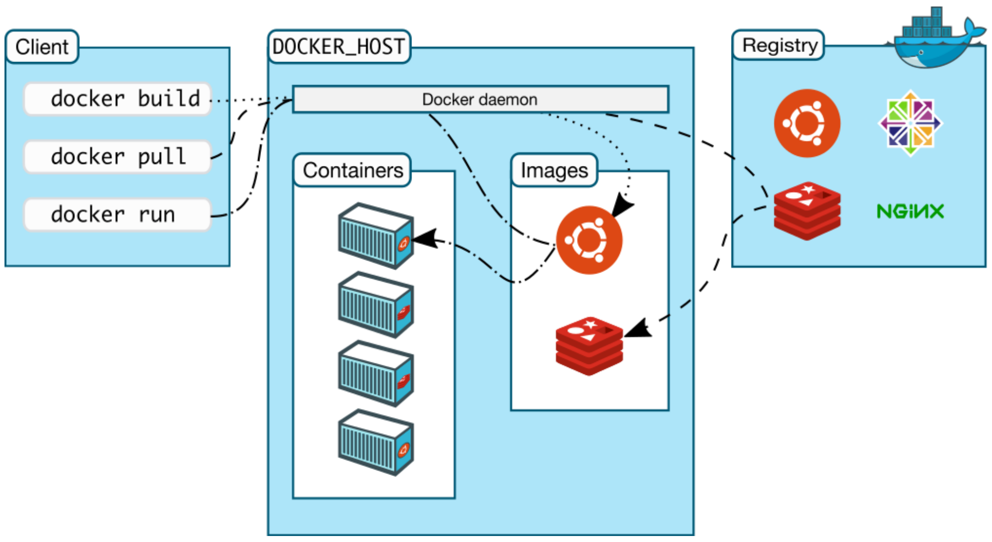


Figure 5: Docker architecture [14]

3 Related Works

People in robotics community already started to explore and use few database solutions to handle sensor data. Many works has been done prior related to evaluating databases and data formats in terms of robot applications[3, 15], comparison between SQL[16], NoSQL [17, 18], Graph databases[4], and use cases where database could be used in robot applications[1]. This section describes about the previous works related to databases used in robotic applications till now, performance evaluation of different databases and also this section concludes why this research work is necessary than the existing work.

3.1 Database in Robotic applications

Niemueller et al. [1] integrated a well known NoSQL database Cassandra with Robotic Operating System(ROS) to handle the data in smart environments. Evaluated this approach in two scenarios, within a realistic robot exploration and with a pessimistic benchmark using randomly generated data. The final solution is compared with two other solutions that are commonly used in ROS applications(rosbag and mongo_ros). The research results shows that, Cassandra can handle terabytes of data and their time-stamp mechanism allows querying and retrieving data without additional efforts. But this approach compared their results only with two other storage mechanisms.

Fourie et al. [4] discussed briefly about the importance of memory recall mechanism for robots. Also, implemented a two level database layer in the centralized server, to store simple data(odometric values) in graph database (Neo4j) and larger sensor data like RGB and depth image and laser scans in key value store (MongoDB). With this architecture, they can achieve sharing of work load between robot and database server. As a result, for executing this query server took between 10 to 100 milliseconds. In this work, they have given a initial motivation to robotic community that using databases is possible and advisable, but author didn't specified why Neo4j and MonngoDB for data storing even though there are lot of database technologies available.

Fiannaca and Huang [3] briefly explains the difficulties in handling the log messages from ROS(Robotic Operating System) and alternate ideas to store ROS logs for utilizing it in future. To overcome the difficulties in storing logs in flat file system, they wanted to use relational database or NoSQL database. So a benchmarking test has been conducted to evaluate and find which one is best. Three databases (MongoDB, PostgreSQL, SQLite3) from NoSQL and relational database family has been chosen for benchmarking. But in recent years, PostgreSQL and SQLite3 databases are little outdated and MongoDB is not only the NoSQL database available. There are new NoSQL databases emerging every month and might perform better than MongoDB. So this evaluation results are not useful to choose best NoSQL database to store sensors logs and benchmarking only one NoSQL database is not an optimal choice.

3.2 Performance Evaluation of databases

Li and Manoharan [16] analyzed each NoSQL databases which are considered for evaluation with the SQL database. Comparison is done based on, Read, Write, Delete, and Instantiate operations on key-value stores. Additionally, investigated the performance of iterating over all keys and experimental results includes the time consumed by these operations. This paper considered only few NoSQL databases (Couchbase, MongoDb, RavenDB) for comparison with SQL database and the scenarios used for evaluation seems very simple and if we consider complex scenarios in robotic applications like geographical range queries, text search based on dialogs and writing/reading blob files, this rankings and metrics may fail. Also this experiment conducted under single instance, so this performance results may differ in case of multi agent systems.

3.3 Conclusion

From all this approaches discussed above, they focused on very specific applications/settings and are always comparing only very few databases within this limited setting. Moreover, they benchmarked the performance with single robot in most of the cases. This benchmarking may not be helpful in multiple robot scenarios. Our intention of this work is, to go beyond that by comparing different model databases (SQL, NoSQL, Graph and Multi-model) in multiple settings/robots with the help of docker containers. This paper [4] seems to go in the direction of what we expect to be the best solution (combination of graph + data storage) and we will be further explored.

4 Problem formulation

The problem addressed in this work is, there is no specific databases available in the market for robotic applications and our task is to identify a set of databases and benchmark them to decide which database will be fit for robotic applications. When data about different robots and their environment combined together called as world model. These days, developers in robotics often use self-written, ad-hoc solutions for handling robotic data. However, in recent years many commercial as well as open-source databases became available that perform similar tasks with promising better performance (wrt features, performance, and querying) as well as additional features compared to the ad-hoc solutions from robotics. However, they all come with advantages and disadvantages and were not designed for robotic applications. So the robotic specific structures and data models need to be imposed on these databases to find out which databases performing better.

5 Approach

To address this problem, a set of Relational, NoSQL, Graph and multi-model databases will be compared qualitatively based on their features. For comparison, databases will be collected from state-of-the-art techniques and current top ranked databases. The selected features include common database criterias into account. Furthermore, features will be derived from the below robotic usecases. Benchmarking scenarios will be defined and implemented from robotic research projects(e.g. ROPOD). Finally, a selection of the databases will be constructed using docker containers and compared against the benchmarking scenarios. This comparison is based on the performance evaluation metrics(e.g. throughput, query execution time, message size, frequency, data replication time and number of robots involved in the scene). The result will be a suggestion for the best practice when to use which database in robotics.

6 Qualitative Analysis

This section contains an extensive qualitative analysis report for a collection of databases. Analysis has been done based on database architecture and other features such as data model, consistency, availability, partition tolerance, query language, language support, schema, storage options, distributed architecture, immutable data and license type.

6.1 Neo4j

Neo4j is a graph based database which supports ACID transactions, native graph storage and processing. It is implemented based on Java and can be accessible from other languages using CQL (Cypher Query Language) via HTTP endpoint or binary bolt protocol. Neo4j stores data in nodes and edges. Each node can have more than one property which represents the real data and edges store relationship type between two nodes. It supports casual consistency and availability and doesn't support partition tolerance. Neo4j is a schema free database, means users are allowed to add/delete attributes to nodes and schema can be modifiable even after database initialization. A single instance of Neo4j can hold 34 billion nodes, 34 billion relationships, and 68 billion properties.

6.1.1 Clustering

Neo4j offers two cluster solutions to ensure redundancy and performance, casual and high availability clustering.

Casual clustering

In casual clustering mode, core servers run all the time(always available) to provide fault tolerant platform which ensures safety. Read replicas scale the whole platform that enables the heavy workloads to be executed in a distributed way. Clients are assured to read at least their own writes.

How casual consistency works?

- Applications send data to the best available Core Servers to write the data.
- System asynchronously replicates the date to the Read Replica servers.
- If Core Server failed, application will at least read its own writes from the Read Replica servers.
- There is no master-slave architecture in casual clustering.

High available cluster

High available cluster architecture includes single master and zero or more slaves. Basic high available cluster contains three instances and each instance have logic for data replication and election management. Once data is written to master, then the same data will be replicated to all slaves automatically. Till the replication process ends, write operation will be locked in master to ensure eventually consistency. If a cluster fails, other cluster will be able to serve the user with the same data, which increases the consistency of data. HA (High Availability) cluster, replicate complete graph to each instance of other clusters, means full dataset is replicated across all cluster. We can make the system highly available for read operations by scaling databases in the cluster.

6.1.2 Cluster components and responsibilities

Each cluster is built with two important components. First component is Neo4j database itself, and the second one is cluster management component which is responsible for data synchronization between master and slave clusters. If the current master goes offline, then new master will be automatically selected from the running slaves by cluster management component.

6.1.3 Sharding and Cache based sharding

A key feature of graph database is "index free adjacency property". This means, graph database can find its neighbors without searching the complete graph. As a result, even the graph is huge the execution time will be constant. But if you consider any relational database, it is directly depend on the total number of rows in the table being queried. But obviously, at some point Neo4j face the resource constraints such as limitations in the amount of RAM available. So, the performance will decrease because of swap caching. To overcome this problem, other databases like relational database use sharding concept.

Sharding is nothing but, partitioning the data across multiple servers. By doing this, number of rows in single database will be decreased and it intern increase the query execution performance. But, the mathematical problem of optimally partitioning a graph across multiple servers is nearly impossible (NP complete) for larger graphs [19]. This is a hard problem in case of graph database. Neo4j offers an optimal way to handle sharding problem in graph databases called cache-based sharding.

Cache based sharding [20] is very simple and it do consistent request routing. For example, Robot 1 request always sent to server 1 and Robot 20 request always sent to server 3. It means, the neighborhood data of robot 1 will be cached in server 1, so whenever a request received from robot 1, the load balancer will route the request to server 1. This strategy is highly effective for managing a large graph that does not fit in RAM.

6.1.4 Performance tuning in Neo4j

Neo4j provides a road map to improve the overall system performance (e.g, faster query execution) in two ways, first one is memory based and second one is disk based optimization.

Memory based [21]

- Configuring page cache and heap size equal to RAM memory, which enables OS to start swapping to disc which in turn reduce the performance of Neo4J.
- Page caching copy the graph from the disc to memory to reduce the cost of disc access.
- One can set the page caching memory size in the neo4j.conf file and larger the memory size, increase the read performance and reduce the query execution latency.
- How to estimate page caching size?
  - For an existing Neo4j database, sum all the size of the available database (which can fit in the page caching size) and add another 20% for growth.
  - For a new Neo4j database, run an import with a fraction (e.g. 1/100th) of the data and multiply the result with that fraction (x 100) and add another 20% for growth.
- Large enough heap sizing will be helpful for concurrent operations. To run Neo4j reliably, between 8GB and 16GB heap size is required.

Disk based [22]

- If there is multiple disks, it divide the store files and transaction logs across those disks.
- We can achieve average seek time of 5 milliseconds with current mechanical drive, but with SSD we can achieve average seek time of less than 100 microseconds this means 50 times faster. Better disks equals better performance.
- But still, SSD is tens or hundreds of times slower than accessing data from RAM.
- To avoid accessing disk, we can install high memory RAM. e.g, few 10's of millions of primitives (nodes, relationship and properties) can be handled with 2 - 3 GBs of

RAM. Hundreds of millions of primitives can be handled with 8 - 16 GBs of RAM. Billions of primitives can be handled with 16 - 32 GBs of RAM.

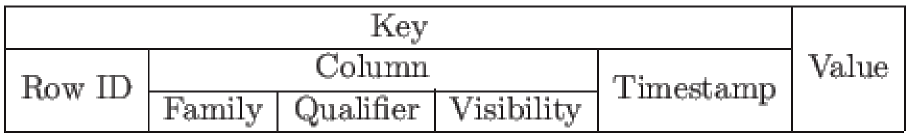
- If we invest in a good SSD, we only need less RAM to handle larger graphs.

6.2 Apache Accumulo

Apache accumulo belongs to wide column store family and it is developed based on Java. It provides eventual consistency and availability. Data can be accessed through declarative query language, API based and REST/HTTP based queries. Language support is very minimal compared to other databases. It is schema less for database, but it has physical schema for key-value pair. Accumulo stores data in memory and it also provides other reliable options like HDFS (Hadoop File System).

6.2.1 Data model

Accumulo data model looks very similar to key value pairs, but key is divided in to three sub elements. RowID specifies the row number for the specific value, and it is not unique. Column specifies the column name of the value. Timestamps represent the time when the value is stored. Accumulo sorts keys by element and lexicographically in ascending order and timestamps are sorted in descending order [23]. Column (value) again divided into three sub elements, Family which is used to group similar column names, Qualifier represents the column name itself and Visibility defines the rules for who can access this value.



Key					Value
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		

Figure 6: Accumulo data model [24]

6.2.2 Components of accumulo

There are multiple components [23] configured together in accumulo to achieve the distributed architecture, and the responsibilities of each components as follows,

Tablet server

Tablet server take care of read and write requests from client and stores the data initially in write-ahead log. Once the memory reaches it threshold, all data will be moved to new files in HDFS.

Garbage collector Accumulo process stores temporary files and objects in HDFS and garbage collector periodically checks those garbage chunks that area no longer useful for accumulo process and delete them. For robustness, multiple garbage collectors possibly run in standby mode. If current garbage collector fails, then based on election a new garbage collector will be assigned.

Master

Master plays an important role in the accumulo architecture. It is responsible for detecting TabletServer failure and assign the tablets to different Tabletserver. Master assigns the tablets to the Tabletserver and unload the tablets from Tabletserver when necessary. [23] Master is also responsible for recovery management if a TabletServer fails. To ensure availability, multiple masters can be initiated and one master will be chosen based on election process. Other masters act as a backup servers.

Monitor

Accumulo monitor is a web application which comes with accumulo package and it helps to constantly check the wealth information of an accumulo server like read/write rates, cache hit/miss rates, scan rate, active/queued compaction's. For debugging accumulo database, monitor is the first entry point to find problems. Like other components have backup option, we can have multiple monitor instance which can be used in case of monitor failure.

Client

Accumulo includes a client library that is linked to every application and it contains logic for finding servers and communicating with them to write and read key-value pairs. [23]

Fault tolerance

In case of TabletServer failures, master automatically moves the tablets to other TabletServer. If any data stored in to WAL during the failure, fault tolerance system also moves that data to newly created TabletServer, to ensure consistency.

6.2.3 Replication

Replication in accumulo copies data to other instances in the cluster automatically, due to the purpose of disaster recovery, high availability or geographic locality [25]. The instance which is handling the current read/write request is called as primary/local instance and the replicated servers are called as peers. Accumulo is eventually consistent in cluster mode, but it is strongly consistent in single instance mode. Using ZooKeeper, we can lock TabletServer during replication of files to the peer servers and later Master and Garbage Collector remove records from meta data and replication tables and files from HDFS respectively.

6.3 Apache HBase

HBase is a Distributed, ColumnFamily oriented data store and it is a type of NoSQL database. HBase provides Consistency and Partition tolerance from CAP theorem. It was developed based on Googles BigTable paper. Since HBase is built on top of HDFS, it provides replication and horizontal scalability out of the box. It provides native API as well as REST API to access the data from HBase, and additional client libraries available for other languages. It uses in memory storage for caching data.

6.3.1 Data model

HBase belongs to column family data model and it's table consists of multiple rows. Each row is associated with a row key and one or more columns associated with values [26]. By default rows are sorted by the row key alphabetically and row keys are not mandatory to be unique. Column in HBase includes a column family and a column qualifier (nothing but column name). Column Family consists of a set of column and its values, often for performance reasons. Each column family have a set of properties such as, whether its values should be cached in memory, how the data should be compressed and encoded [27]. Each row should have the same column families, but the row might not store any data in a given column family. Column qualifier is nothing but column names and it is fixed during table creation. Eg., laser_sensor:id & laser_sensor:manufacturer. In the given example, laser_sensor represents column family and, id and manufacturer represents column qualifier. Cell is a combination of row, column family, column qualifier, and contains a value and a timestamp and timestamps are unique.

6.3.2 Architecture

HBase architecture is based on master slave model. There can be many master running to make sure availability. Table is splitted into multiple regions and regions are stored in RegionServer. The data about where the regions are stored is handled by master instance. At very first, client contacts zookeeper and then zookeeper ask master to find which RegionServer have the requested data. Once the RegionServer returns the location, zookeeper give the lookup information to client and then client will directly access the data. The data are stored in HDFS, and frequently accessed data are cached in a memory block which is resides on RegionServer. Since HDFS provides replication and fault tolerance out of the box, so HBase does not need to manually configure replication setup.

6.3.3 Advantages and disadvantages

- HBase supports ACID in limited ways, namely Puts to the same row provide all ACID guarantees.
- HBase itself doesn't contain MapReduce, but it uses Apache MapReduce which is a software framwork used often with Apache Hadoop.
- HBase supports both linear and modular scaling and clusters can be expanded by adding RegionServers.
- HBase is strongly consistent, so it is suitable for operations such as high-speed counter aggregation.
- It supports automatic shrading and automatic RegionServer failover.
- HBase supports massively parallel processing using MapReduce framework.
- Performance is poor in handling large blobs (>3GB).

6.4 Apache CouchDB

Apache CouchDB is a document based NoSQL data store which is developed based on Erlang. It stores the data in JSON format and it used javascript to query the data. Read performance is really amazing because it is auto indexing the primary key whenever there is a new data. It uses multi version concurrency control, means it does not lock the database even though there is a write operation. If there is any conflicts, databases will resolve them.

6.4.1 Data model

CouchDB stores data in the form of documents. Each database in CouchDB consists of collection of documents. Each document is uniquely named in the database and stored in the form of key value pairs. Each document can contain any number of fields, so that we can dynamically add different fields in each document. This helps users to not give importance to database schema. CouchDB document model is lockless and optimistic. User can get the document to client side and make changes in the document and save them back to the database. For example, if another user editing the same document and saves it first, then the first user will get edit conflict while saving the document.

6.4.2 Replication

CouchDB is a peer-based distributed database system and it allow multi master replication architecture in the cluster [28], so that all the nodes in the cluster can process read/write request. Even there is a network disconnection between two replicated servers, each server allow users to add/update changes in their database and once the network connection is back, the changes will be replicated bidirectionally. The replication process is incremental, it means replication starts with the documents updated since the last replication. For example, robots which are working under low/no network coverage areas, they can store data with their local CouchDB instance and once robot connected with network, it can replicate their data with the remote servers.

There are two flavors of replication available in CouchDB, continuous and triggered replication. If continuous replication is set to true between server one and server two , for every new changes in server one will trigger a replication request to server two to update new data in server two database. Replication system will not trigger right away after the changes, instead couchDB's complex algorithm chooses the right moment to replicate to achieve maximum performance. But this is unidirectional, also called as master slave replication, that means if there is any changes in server two, it will not update in server one. To make the continuous replication bidirectional, we need to set continuous replication to true also from server two to server one. From now on, for every changes in server one or server two, it will replicate the data bidirectionally also called as master-master replication.

Execute the command in server one: [28]

```
curl -X POST -H "Content-Type: application/json" http://host1:5984/_replicate -d
 '{"source":"cmr","target":"http://host2:5984/fs","continuous":true}'
```

The above command implements unidirectional continuous replication. Execute the below command [28] in server two to achieve bidirectional continuous replication.

```
curl -X POST -H "Content-Type: application/json" http://host2:5984/_replicate -d
 '{"source":"cmr","target":"http://host1:5984/fs","continuous":true}'
```

Triggered replication replicates only once between the mentioned servers. After that, servers will not replicate from source to target continuously.

6.4.3 Conflict management

CouchDB resolves conflicts based on the _rev id which is tagged to each document. Lets have a look at how _rev id looks like,

```
3-5d0319b075a21b095719bc561def7122
```

The first part is (3-) an integer with dash and the integer represents number of the times the document has been revised (updated). So for every update the revision number increases by one number. The second part is an md5-hash over a set of document properties. If two servers have two different version of same document, then it will replicate the recent version document to the other server (with old version document). Let say if two servers have different version document with same rev number and different md5 hash id, then _rev values are compared in ASCII sort order and the highest wins. For example, consider the below scenario, two different _rev id's but the version number is same. In this case, sorting them based on ASCII helps to find the winner [29].

```
2-de0ea16f8621cbac506d23a0fbbde08a
2-7c971bb974251ae8541b8fe045964219
```

6.5 Apache cassandra

Casandra is a well known NoSQL database management system. Cassandra able to handle large amount of data across many low cost servers. Cassandra provides high availability and no single point of failure. It is highly robust because it runs many clusters in multiple data centers. It does not support master slave architecture. Cassandra cluster is decentralized, so there is no network bottlenecks and all the nodes are homogeneous in the cluster. Read and write throughput increases linearly with adding new nodes in the cluster. We can choose our replication strategy between synchronous and asynchronous replication for each update and it is eventually consistent. Query language used in Cassandra is simple compared to SQL language used in Relational Databases. Cassandra supports all data formats such as structured, semi-structured and unstructured. Data structures can be dynamically changeable depends on the application needs. Cassandra tested and proved the scalability with the following organizations [30],

- Netflix with 2500 nodes, 420 Terra Bytes, and over 1 trillion requests per day.
- Chinese search engine Easou with 270 nodes, 300 Terra Bytes, and over 800 million requests per day.
- eBay with 100 nodes and 250 Terra Bytes.

6.5.1 Architecture

Cassandra architecture [31] is designed in such a way that, it can handle big data workloads without single point of failure. Cassandra architecture designed based on peer to peer distributed systems. The components and responsibilities of components in the architecture as follows,

Node

Data are stored in nodes and it is the basic component of cassandra. All the nodes in a cluster plays same role, means there is no master slave role. Each node is independent of other nodes, but all nodes are connected internally. When a node is down, read/write requests will be handled from other nodes in the cluster.

Data center

Collection of nodes are called as data centers.

Cluster

Collection of many data centers is called cluster.

Commit log

When a write request comes, Cassandra first write the data in the commit log and it will be used for data recovery. Commit log is an append only data system, means we can write data sequentially, but we cannot go back and update old records.

Mem table

Once Cassandra writes the data in commit log, then it writes the data in mem table, but this data is not persistent.

SSTable

If the data limit reached in mem table, then Cassandra flush the data from Mem table to SSTable (in disk).

6.5.2 Replication

Replication make sure no single point of failure. In case of hardware failure, or link disconnection between nodes, data which is replicated in other nodes still serve the requests [31]. Cassandra replicates all data in different nodes to assure 100% availability and it replicates data based on two important factors,

- *Replication strategy* tells where to replicate the data next in the cluster.
- *Replication factor* tells number of required replicated copies. For example, three replication factor means, same data will be replicated in three different nodes. To make sure no single point of failure, the chosen replication factor should be 3.

There are two types of strategy we can use for replication,

Simple strategy is used when we have only one cluster in the network. Partitioner chose the first node to write data and remaining replicas will be written to next consecutive nodes in clockwise direction.

Network topology strategy is used when we have multiple datacenters (multiple node rings connected internally) in a cluster. Network topology strategy writes the first data in one node and make replication in clockwise direction till it reaches the first node in the next ring. So, if a datacenter (node ring) is down, we can get the data from other datacenter.

6.5.3 Write and Read work flow

Write work flow

At first incoming data will be written into commit log and memTable. Mem table is a temporary data storage place, and once mem table reaches it's threshold limit then data will be flushed to SSTable which is a permanent storage area. A node which receives write request will act as a coordinator node.

Coordinator node then forward the data to the nearby nodes for replication. Coordinator node choose the number of nodes for replication based on the replication factor. For example, if the replication factor is 3 then coordinator node tries to replicate the data in 3 nearby nodes. It responds to client with success acknowledgment once the QUORUM condition is satisfied. QUORUM is a way to tune consistency level. For example, if there is four node in the cluster ring and the QUORUM is set to 2, then the coordinator node have to wait till it receives two success response from nearby replica nodes. Once coordinator node get successful response back from two nodes (QUORUM = 2), then it respond client with success acknowledgment. To achieve strong consistency, the QUORUM condition should set to 'ALL', this means coordinator node have to wait till the data successfully replicated in all available nodes, then it will acknowledge client.

Read work flow

Cassandra coordinator sends three types of read request to replicas:

- Direct Request - Coordinator send request to anyone of the replicas and get the data.
- Digest Request - Then, it send digest request to remaining replicas and checks whether the returned data is consistent with other replicas.
- Read Repair Request - If the data is not consistent with other replica nodes, read repair request will be triggered to update the recent version of data in all replica nodes. This make sure the consistency and consistency level is configurable.

6.5.4 Rules for Cassandra data model

Cassandra does not support joins, group by, OR clause, aggregations, etc., by default, So developers have to store our data in such a way that data should be retrieved completely. To improve the read performance and availability, increase the number of writes and writes are not expensive in Cassandra. If data is replicated in many clusters, it ensures high data availability to clients and no single point of failure. Disk space are cheaper than memory, CPU processing and IO operations. Cassandra suggests that, spread data evenly around the cluster and minimize the number of data partitions [32]. If a data is partitioned in to many clusters, then server will take time to retrieve those data which in turn affects read query performance.

6.6 OrientDB

OrientDB [33] is a document graph database (aka multi model database) which gives all graph capabilities with other databases features. OrientDB uses Record as an element of storage and each record will be assigned with an unique ID. Document is nothing but a key value pairs, and it refers to fields and properties belong to a class. OrientDB works in the way of Class works from Object-oriented programming paradigm. We can define our data model as a class and can add properties and constraints to each data belongs to the model. OrientDB also supports SQL syntax and keywords like SELECT, INSERT, UPDATE, DELETE, WHERE, ORDER BY, LIMIT, etc.,

How class works in OrientDB?

Class represents a specific data model, for example Actuator can be a class. Unlike relational database, classes can be scheme-less, schema-full or mixed. Class can inherit properties from other class. When we create a class, by default a cluster (let say a space to store records of this class) will be created for this class. e.g, for class Actuator, by default a cluster will be created in the name of 'actuator'. There are enough commands provided by OrientDB to create classes, adding properties to class, adding restrictions, etc.

How Relationship between records works in OrientDB

Not like relational database, OrientDB does not support Joins. Instead, they use Links to point other records. Source record have address (Record ID) of target Record, which works like pointers in memory.

Types of storage's that OrientDB supports

- plocal which is a Persistent disk storage.
- Memory storage which is local and alive as long as the JVM is running. This is a good choice for reading data frequently.
- Remote which is a storage from other location.

Types of relationships

- LINK, which points to single target record.
- LINKSET, which points to multiple records (without order).
- LINKLIST, which points to multiple records (with order).
- LINKMAP, which points to multiple records with key stored in the source records.

6.6.1 Clusters and its uses

From version 2.2, OrientDB creates multiple clusters for each class to improve parallelism. Number of clusters created is equal to number of CPU's cores available in the server. The advantage of running multiple clusters on multiple locations are, Optimization, Indexes, Parallel Queries, and Shrading [34]. Adding clusters to a specific class is simple with OrientDB commands.

6.6.2 Graphs in OrientDB

Graphs works based in network like structures consisting of vertices and edges. Vertices have its own properties and edges are used to connect two vertices. By default, OrientDB provides a persistent classes for vertex (V) and edge (E). But we can create our own vertices using given commands (CREATE VERTEX V) and to work with existing graphs it provides other helpful commands like DELETE VERTEX, CREATE EDGE, UPDATE EDGE, and DELETE EDGE. We are allowed to extend V and E classes to our normal class and it allows object oriented inheritance in graph elements. There is other interesting topic called Lightweight edge and it works as like normal edge but only the difference is lightweight edge does not have properties. Light weight edge improves performance and reduce space usage.

6.6.3 Replication vs Shrading

OrientDB supports multi master replication architecture. Multiple nodes that have same copy of data will increase high availability and improve the performance of reads. But it is not good for writes, since it write operation have to replicate the same data to all nodes and this increase the latency. In this case, shrading the database across multiple nodes improves write performance. Also, using asynchronous replication improve the performance and reduce the latency issues.

6.7 PipelineDB

PipelineDB runs SQL statements continuously on streaming data. The output from the continuous SQL statements stored in tables. As new data streams in, the SQL queries result updates the old values in the table. PipelineDB built on top of PostgreSQL.

Continuous views

PipelineDB's fundamental abstraction is called as continuous views [35]. Continuous view is nothing but the regular view data stored in the disk and if new input stream comes in then continuous view will be updated. Once the stream of data is read by the continuous view, the data stream will be discarded.

Streams

Streams are similar to table rows. Streams allow client to push data through Continuous views [36]. Multiple aggregate functions has been provided by PipelineDB that can be applied to streamlined data before storing it in to database and only the results will be stored into database. Since PipelineDB is built on top of PostgreSQL, all client libraries that supports PostgreSQL also works with PipelineDB.

Sliding window

Continuous views updates their data continuously from input stream over time. PipelineDB provides option to restrict the update between time frame. We can set multiple sliding windows for same query. For example, keep track of robot events for last 5 minutes, 10 minutes, one day, etc,.

6.7.1 Replication and high availability

PipelineDB supports Postgre's streaming replication, both synchronous and asynchronous out of the box [37], but it doesn't come with high availability options, if the primary node fails, then user have to manually switch to the next hot standby server. Pipeline documentation says that, one can use "governor" application to achieve high availability, but "governor" program github page says the program is deprecated and archived [38]. We can integrate Apache Kafka and Amazon Kinesis streams with PipelineDB. PipelineDB supports parallel processing, means multiple workers can be created to handle input streams. But, they have poor recovery management. If a worker or node fails, there is potential possibility of losing data. PipelineDB user community is not strong and number of users are less. Initially PipelineDB allocates lot of space in memory to store the incoming data streams, and if the memory becomes full then clients have to wait to stream data.

6.8 Druid

Druid is an open source database mainly used for real time analytics on large data sets. It is a column oriented distributed data store and built based on shared nothing architecture. In Shared Nothing architecture, each node is independent and they don't share memory. Druid built on top of open source RDBMS MySQL. Druid provides different types of nodes that are partially independent with each other. All these nodes combined together to form complete Druid system. It does not support joins. Druid supports simple aggregation methods such as sum of floating point and integers, minimum, maximum and complex aggregation such as cardinality and quantile estimation. To achieve availability, we should run minimum of 2 nodes of each node type [39].

6.8.1 Data model

Druid can store data in many formats such as JSON, CSV or custom format in the data table. Data tables are collection of time stamped events and partitioned in to multiple segments. Each segment holds 5 to 10 million rows. Each and every row should contain timestamps. Druid segments are stored in a column orientation. Druid executes query based on its own query language and accepts queries as POST requests.

6.8.2 Types of nodes

In Druid system, there are four types of nodes used to ingest and store continuous stream of data events in the database and the detailed description about each nodes as follows [40],

Real time nodes

Real time nodes used to ingest and query event streams and events are available once these real time nodes complete the indexing. After ingestion, nodes store the event streams in in-memory storage and to avoid out of memory exceptions, events are flushed to disk based on threshold memory limit which is configurable. Events stored in in-memory as well as disk are immutable. On a periodical basis, all the events stored in the disk are merged together as a block which is called as segments. These segments are then stored in to permanent backup storage such as S3, HDFS and druid call it as deep storage. Kafka has been setup in between event stream producer and consumer to prevent event data loss. Kafka stores the events in a buffer and then transfers them to real time nodes. During system restoring from failure, it can easily start from the last event index received. Kafka can send same events to multiple real time nodes for replication which ensures availability and it can also split (partition) the events to different real time nodes. This model able to consume raw data at approximately 500 MB/s (150,000 events/sec or 2 TB/hour) [40].

Historical nodes

Historical nodes load and serve the immutable data segments created by real time nodes. Historical nodes follow shared nothing architecture, so one node does not know about events stored in other nodes. Zookeeper is used to find what events are stored in which historical nodes. Historical nodes serve the data from deep storage and also from cache. During node startup, historical nodes serve the available data from the cached memory. Historical nodes can support read consistency, because they only deal with immutable data. Historical nodes depend on Zookeeper for segment load and unload, if Zookeeper fails system cannot respond for the new data.

Broker nodes

Broker nodes act as a query routers to historical and real-time nodes. Zookeeper help broker nodes to find where the segments are stored in the historical node. Before returning the final query result, broker nodes merge the results from historical and real-time nodes. Segments can be found from cache to reduce the query execution time. Caching only contains historical node data, and no data from real time nodes. If historical node fails, cache still handles the request. Data is still queryable in case of Zookeeper outage. Broker node use the last known view of cluster to find where the segments are stored.

Coordinator nodes

Coordinator nodes used to load new data, drop outdated data, replicate data and move data to load balance [40]. There can be a multiple coordinator nodes for backup purpose.

6.9 ArangoDB

ArangoDB is a native multi model database that supports documents, graphs and key-value pairs [41]. It supports ACID transactions but it's optional. ArangoDB have its own query language called ArangoDBQueryLanguage(AQL). It is possible to run application and database together in application server provided by ArangoDB. ArangoDB uses modern storage hardware like SSD and large caches. Easy to setup clusters in few clicks. The traditional way of connecting to database is an application running in the server to connect to database and serve data to client in the front end. ArangoDB uses FOXX micro services which is a javascript program running directly within the database with native access to in-memory data.

6.9.1 Data model

Documents are initial components to store data and it looks closely to JSON format and stored in binary format called VelocityPack. VelocityPack was invented by ArangoDB [42] to overcome few disadvantages like compactness, platform independence, fast access to subobjects and rapid conversion from and to JSON. Documents may contain zero or more attributes. Documents are collectively grouped in to collections. Collections may have zero or more documents. We can compare collections to tables, documents to rows and attributes to columns in RDBMS. In RDBMS, columns are fixed called fixed schema and in ArangoDB attributed are not fixed called as schema less. It means, each document may have different number of attributes and all documents can store in same collection.

6.9.2 Write mechanism

ArangoDB creates a new data with new version number instead of updating existing data and old data will get deleted during compaction. All data will be stored in memory mapped files and Write-ahead log. Data in the WAL is used for data recovery during crash and replication setup when slaves need to replay the same sequence of master. The default size of WAL log file is 32MB but the size is configurable and a new log file will be created once the previous log file reaches its limit.

6.9.3 Indexing

ArangoDB provides different types of indexing mechanism which includes hash index, skip index, persistent index, geo index and fulltext index. Geo index might be helpful in robot navigation system in case of finding documents from nearby locations or in a range (in meters) with the help of Geo index functions provided by AQL.

AQL Geo index functions [43]

- Near(coll, latitude, longitude, limit, distanceName) - Returns a list of limited documents near to the given co-ordinates.
- WITHIN(coll, latitude, longitude, radius, distanceName) - Returns a list of documents with in the given radius.
- WITHIN_RECTANGLE(coll, latitude1, longitude1, latitude2, longitude2) - Returns a list of documents with in the bounded co-ordinates.
- IS_IN_POLYGON(polygon, latitude, longitude) - Returns true/false based on the given co-ordinates present inside the polygon or not.

6.9.4 Scalability

ArangoDB can scale horizontally by adding multiple data nodes and vertically by upgrading server hardware. Horizontal scaling used to achieve resilience by replication and automatic fail-over. ArangoDB have Synchronous and Asynchronous replication options. The Architecture of ArangoDB has master/master model with no single point of failure. From CAP theorem, ArangoDB chose Consistency and Partition tolerance.

6.10 MongoDB

MongoDB is a document database that provides high performance, high availability and automatic scaling. MongoDB stores data in the form of document and it consists of field and value pairs which is similar to JSON objects. The value may consists of other documents, arrays, and arrays of documents. A set of documents belongs to a collection.

Advantages of documents [44]

- Documents (i.e. Objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

MongoDB query language supports Data Aggregation, Text Search, and Geospatial Queries out of the box. MongoDB replication facility provides automatic failover, data redundancy and increasing data availability. It also provides map-reduce option to perform aggregation operations on documents.

Geospatial queries [45]

Like ArangoDB, MongoDB supports Geospatial Queries which will be useful for storing and querying robot co-ordinates.MongoDB stores geospatial data as GeoJSON objects or legacy co-ordinate pairs. GeoJSON objects embed with a field named type (E.g. Point) and a field named co-ordinates (Longitude and Latitude).

6.10.1 Storage engine

MongoDB provides different storage engines [46] and we can choose one that suits our usecase.

- GridFS is one of the storage engine that is used to store large files which exceeds 16MB document size limit.
- WiredTIger is the default storage engine and provides document level concurrency, checkpointing and compression.
- Data writes initially in memory and then flushed in to disk when the size reaches the limit.
- MMAPv1 is original storage engine of MongoDB and it will used for heavy read and write operations.

MongoDB write to the data files on disk every 60 seconds and writes to the journal files every 100 milliseconds. The limits are configurable. With MMAPv1, automatically MongoDB use systems free memory for caching and contiguous disk space for storing documents. In-Memory storage engine stores all data in the memory and it is non-persistent storage space. So it is better to have application data and system data in In-Memory storage engine. By using In-Memory storage engine, we can reduce usage of disk I/O operations and latency.

6.10.2 Replication and high availability

To increase high availability data from the primary server will be replicated to multiple secondary servers. Number of secondary servers determines the availability factor. Data will be replicated asynchronously to secondaries. Automatic failover mechanism always hear heart beat from primary servers. If there is no heart beat for 10 seconds, then it will elect a secondary server as primary. Primary node always get write and read request, but we can customize the preference and change it to secondary nodes (read requests only). But it is not sure that user will get updated data as result since the replication is asynchronous. MongoDB also support shrading data in to multiple data nodes.

6.11 InfluxDB

InfluxDB is a time series database which is used to handle large amount of timestamped data. InfluxDB is currently used for DevOps monitoring, application metrics, collection of IoT sensor data and real time analysis [47]. InfluxDB offers many ways to write/read data into database which includes, HTTP API, command line interface, client libraries and plug-ins for common data formats such as Graphite. Storing thousands of readings from sensors into database every second and this might lead to storage issues at some instance. To handle storage issues, we could utilize Downsampling and Data Retention which were offered out of the box by InfluxDB.

Downsampling

Downsampling [48] is achieved by "Continuous Query" that runs automatically and periodically in the database under specified function. For example, calculating mean and variance for a specified field automatically and store the results in a new DB and delete the raw data.

Data Retention

Data Retention [48] is achieved by "Retention Policies", means we can define a policy that describes how long we can keep a data in InfluxDB.

Single node InfluxDB is completely open source but, clustering requires closed-source commercial product. Single node instances does not offer redundancy and availability. If a node fails, users cannot write or read from the database. Clustering nodes offers high availability and redundancy (replication). To achieve high availability, InfluxDB suggest that the replication factor should be 3 and the cluster should have an odd number of replication factor.

6.11.1 Data model

In InfluxDB, data will be stored in measurements as rows. Each row should contain one time stamp, multiple field values and tag values. Each measurement should have "time" field which denotes time stamps and multiple field keys and tag keys. Fields are used to store real data which has been recorded at each time stamp and Tags are used to store meta data about something (e.g. location, device name, sensor type, etc ..)

Example: Measurement name: RGB

Timestamp	R	G	B	robot_id	location
2015-08-18T00:00:00Z	100	120	90	1	3
2015-08-18T00:01:00Z	103	126	98	1	3

Here, measurement name is RGB and field names are R,G,B and tag names are robot_id and location.

InfluxDB is a schema-less database which means, we can add measurements, tags, fields on the go. But still, InfluxDB documentation includes good recommendations for how to handle schema to give good performance. For example [49], InfluxDB tags are indexed and fields are not indexed. So if we have more series in tags, indexing will take more time which leads to poor response time.

InfluxDB is not a full CRUD

InfluxDB receives series of data from many sources like sensors, transactions, data from websites and the data has been used for analysis and visualization. So, InfluxDB priorities the CR-ud and it focus on create once and read multiple times, and rarely use update and delete which improves the performance.

6.11.2 Features

- InfluxDB uses TSM (Time Structured Merge tree) engine for high speed ingestion and data compression.
- It does not require external dependencies.
- Expressive SQL like query language.
- Continuous queries do the aggregation operation on data automatically to make frequent queries more efficient.
- Influx Enterprise edition is configurable with Chronograf which is a visualization tool to monitor query execution, nodes in cluster, manage users, and explore/visualize data.

6.12 TitanDB

TitanDB has been shutdown and currently TitanDB code base is forked by JanusGraph [50]. JanusGraph is working in incorporating new functions and improving performance and scalability [51]. So we are skipping TitanDB from the performance evaluation.

6.13 MariaDB

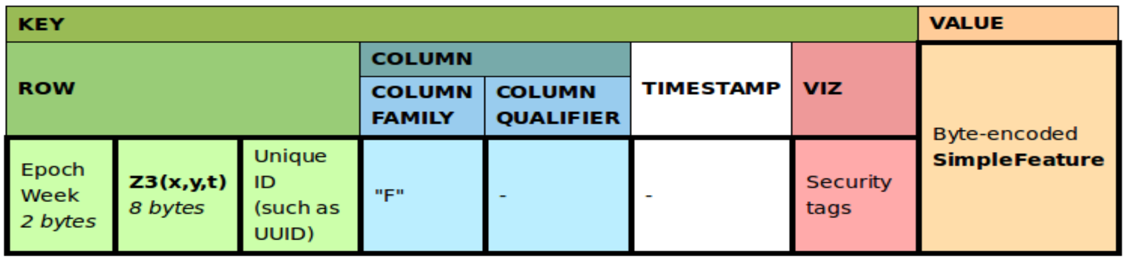
MariaDB was forked from MySQL database and there is no difference in query language and architecture. MariaDB is maintained by the original developers of MySQL and current MySQL is maintained by Oracle [52]. Since both databases have very similar architecture and features [53], we are skipping MariaDB from the performance evaluation.

6.14 GeoMesa

Geomesa is a powerful tool to do analytics on the huge set of geo spatial data stored on cloud and distributed computing systems. Geomesa works on top of anyone distributed systems. Currently they have support for Accumulo, HBase, Bigtable, Cassandra, Kafka and Spark. Geomesa stores data in the form of Geomesa formatted tables in all other databases, to keep the same table format. Then, using Geoserver application we can connect with any datastore that was created and stored by Geomesa.

6.14.1 Data model

Geomesa stores data as key value store. To store normal data, key value stores create a unique identifier for each values(data) and we can query the data based on the unique key. When using a key-value database, good design of the keys themselves can lead to more efficient applications. To store spatio-temporal data, Geomesa designed the key which represents the time/space and location of the record. Geomesa uses latitude, longitude and time-stamp to generate key for a value(may be a simple features like line, polygon, image,etc..). The line connects between the points(keys which contain latitude/longitude/time-stamp) represented as z-curve. This line visits each point(key) exactly once, and the points(keys) should be in ordered way. GeoMesa will be a good choice if and only if there is a huge dataset based on location, since it indexes data based on location key.



KEY							VALUE
ROW			COLUMN		TIMESTAMP	VIZ	Byte-encoded **SimpleFeature**
			COLUMN FAMILY	COLUMN QUALIFIER			
Epoch Week *2 bytes*	**Z3(x,y,t)** *8 bytes*	Unique ID (such as UUID)	"F"	-	-	Security tags	

Figure 7: Geomesa data model [54]

6.15 SiriDB

SiriDB is a time series database which supports cluster for scaling and redundancy [55]. Collection of data will be stored under a uniquely identifiable name. Each data is stored as points. Each point consists of key which is nothing but timestamp and value could be any number. SiriDB allows users to insert data in any timestamp order, which means one can store old data and new data at anytime. It supports numeric data types to store timestamps (integer or float). Data can be queried by name of the collection, regular expressions or dynamic groups.

6.15.1 Scaling and redundancy

SiriDB can be scaled horizontally by adding pools (clusters). Each pool can have upto two servers [55]. Initailly SiriDB starts with single pool with single server, and during expansion we can add one more server in a single pool or multiple pools with two servers in it. If one server fails in a pool, the other one will be online to serve requests. SiriDB algorithm will only move part of data from old pools to a newly added pool, which makes the old pools to store data continuously and distribute load on all pools by avoiding traffic only on new pool. SiriDB runs the replication with low priority in the background, so the old databases remains fully operational until a new pool is being added.

6.15.2 Replication and shrading

If a pool receives new point, then it will first randomly determine which server to store new point and stores them in buffer file and buffer memory. In buffer file, points are stored not in order and this points are only used to rebuild after a restart or failure. In buffer memory, points are stored in order (sorted) for faster access. If the buffer is full, then SiriDB creates shrad file and move the old data (only between certain time ranges) to shrad file. Each shrad file have chunks of ordered points with index. The pool which receives the read request, find all points from other pools and send back the final result to client [55].

6.16 GraknAI

GraknAI is a hyper-relational database for knowledge oriented systems [56]. In GraknAI, realationships between attributes should be pre-defined to ensures information integrity. It uses Enhanced-Entity-Relational model to understand the relationships and works intelligently. Graql is a reasoning and analytics query language used to query data based on context oriented. GraknAI is a suitable option to represent real world objects and relationships. In case of robotic application, we can define the realtion between multiple robots and objects in environment, but it is questionable that how efficient GraknAI can handle stream of sensor data. GraknAI is a newbie in database market and it have more dependency on other databases, for example it depends on Cassandra for replication. Also it supports primitive data types, and it will be a blockade for storing large images in the database. Yes, the database might be useful in robotics for knowledge representation and understanding context, but not fit for storing all sensor data and replicating data between robots.

6.17 MySQL

MySQL is a well-known Relational Database Management System. It stores data in tables and relations can be defined between tables using primary and foreign keys to represent the connection between data. It indexing data based on the primary key which improves the speed of read query execution. Data model or schema should be defined before inserting data into tables. It provides ACID property to provide strong consistency on the data and it also available depends on the chosen configuration.

It supports to store 50 million rows or more in a table and the default size for a table is 4GB and configured to the limit of 8 million terrabytes [57]. MySQL database performs robust in handling web applications data, but it shows a poor performance on ingesting robot generated data and large blob files. It also supports only master slave architecture, so it would be a constraint in case of multi robot systems. In state of the art analysis, while comparing relational database with NoSQL database, relational database shows deficient performance in handling stream of sensor data.

6.18 Summary of qualitative analysis

Name	Data model	C	A	P	Query language	Schema		Storage options	Distributed	RA
						Full	Less			
Geomesa	Key-Value pair	✓	✓	×	Common Query Language	×	✓	Cloud based storage	✓	-
SiriDB	Time-series	✓	✓	×	Siri Query Language Siri prompt Siri HTTP	✓	×	Memory & disk	×	-
Grakn.AI	Graph	✓	✓	×	Graql, REST API	×	✓	Disk storage	×	-
Neo4j	Graph	✓	✓	✓	Cypher query language, REST/HTTP API, TinkerPop 3, API Based	×	✓	Memory & Disk	✓	[4]
Apache Accumulo	Wide column store	✓	✓	✓	Declarative query language API Based REST/HTTP API	✓	✓	Memory,Disk & HDFS	✓	-
Apache HBase	Column family	✓	✓	✓	HBase native APIs, REST API, Thrift API	×	✓	Memory,Disk & HDFS	✓	-

CAP - Consistency, Availability, Partition tolerance | RA - Robotic Applications

Table 1: Qualitative analysis summary

Name	Data model	C	A	P	Query language	Schema		Storage options	Distributed	RA
						Full	Less			
Apache CouchDB	Json Key value pair	✓	✓	✓	RESTful HTTP, JSON API	×	✓	Memory,Disk, HDFS & Cloudant	✓	-
Apache Cassandra	Wide column store	✓	✓	✓	Cassandra Query Language (CQL)	×	✓	Memory & Disk	✓	[2]
OrientDB	Multi-model	✓	✓	✓	Java API, Gremlin API, & HTTP API	✓	✓	Memory & Disk	✓	-
PipelineDB	RDBMS	✓	✓	×	SQL	✓	×	Memory & Disk	✓	-
Druid	Column Oriented	✓	✓	×	HTTP API & SQL	✓	×	Memory, Disk & HDFS	✓	-
InfluxDB	Time Series	✓	✓	×	HTTP API & Command line interface	×	✓	Memory & Disk	✓	-
MongoDB	Document store	✓	✓	✓	Mongo shell & JSON query	×	✓	Storage engine, Journal & GridFS	✓	[1, 3, 4] [58, 59]

CAP - Consistency, Availability, Partition tolerance | RA - Robotic Applications

Table 2: Qualitative analysis summary

Name	Data model	C	A	P	Query language	Schema		Storage options	Distributed	RA
						Full	Less			
ArangoDB	Multi-model	✓	✓	✓	HTTP API & JSON query	×	✓	Storage engine, Memory & Disk	Yes	-
Couchbase	Document store	✓	✓	✓	N1QL & (pronounced as ”nickel”)	×	✓	Memory & Disk	Yes	-
MySQL	RDBMS	✓	✓	✓	SQL	✓	×	Memory & Disk	Yes	-

CAP - Consistency, Availability, Partition tolerance | RA - Robotic Applications

MySQL additionally supports ACID.

Table 3: Qualitative analysis summary

7 Quantitative analysis

In the previous section, we have seen the features of all databases and in this section we sort a set of databases for quantitative analysis and reasons behind why we chose them. As a fair comparison, we decided to pick at least one database from each data model and the database should be suitable for robotic applications. Based on the qualitative comparison of databases, concluded that the following databases would be considered for quantitative analysis,

- Neo4J
- OrientDB
- Apache CouchDB
- MongoDB
- Apache Cassandra
- ArangoDB
- InfluxDB
- MySQL

7.1 Graph database

As there are other graph databases (e.g., GRAKN.AI) available in the initial set of selected databases, we decided to choose Neo4J for the following features. Neo4j concerns more on high availability compared to other graph databases and easy to query data since it have multiple interface (Cypher query, REST, HTTP, TinkerPop 3, etc.,) to access data. Also Neo4j supports many languages (more than ten which includes the important languages like Java, Python, Scala), but other graph databases have support for very few languages. Neo4J has been already used for an evaluation in mobile robots. So, we could potentially look how they evaluated Neo4J database with robot sensor data and still the results would be compared and improved with other databases results. Dockerhub provides customized docker images which can run on multiple platforms includes Raspberry Pi.

7.2 Document store

In document store data model series, we choose Apache CouchDB and mongoDB.

Apache CouchDB

The reasons why we choose CouchDB than other document store databases are following,

- It supports simple HTTP protocol to access the data, so CouchDB does not let developers to write different conversion wrappers to access data from the database.
- CouchDB supports more languages which includes C, Pyhton, Java, etc.,
- Easy to configure distributed instances using cloudant.
- It supports MapReduce.
- Immutable property helps to add new data every time and deletes the old data based on revision number during compaction process.

CouchDB have an unique feature called Sync, means synchronization between different nodes. Sync overcomes network latency problems. We might face issues in two scenarios, where couchDB helps,

- First one is latency issue, if there are multiple nodes located in different places around the world, we could choose the nearest one and store data in it. Later system will sync the data with all other nodes.
- Second one is no network coverage issue, in this scenario we could use PouchDB (written in javascript, act as a local instance) in local machine to store all the data. Once the machine connect to internet, pouchDB sync the local data with CouchDB nodes.

MongoDB

MongoDB is also a document store like CouchDB and it have similar features from CouchDB like schema less, eventual consistency, high availability and partition tolerance. Moreover, MongoDB can be configured with different storage engine options (WiredTiger, Journal, GridFS). MongoDB offers Geo-spatial Queries out of the box, but CouchDB doesn't. Since MongoDB is the key player in document store databases and previously used in many robotic projects and database evaluations [1, 3, 4, 58, 59], we would like to consider MongoDB for comparison with CouchDB as well as with other data model databases.

Other document store databases (for eg., Couchbase) provide similar kind of features but lacks behind in few important features (HTTP access, Sync, language support, easy configuration) and community support. So, we skip them for quantitative analysis.

7.3 Wide Column Store

In Wide column store database series, we have chosen Accumulo, HBase, and Cassandra, and all of these three databases belongs to Apache family. This three databases mostly have unique features like consistency, availability, partition tolerance, in memory caching, licensing, scalability and immutability. But, we choose Apache Cassandra out of other databases because it has more language support, easy to configure, independent data storage (It doesn't require separate storage configuration like HDFS). Also, ROS (Robotic Operating System) have an existing library called cassandra_ros which will be useful for Cassandra database interface.

7.4 Multi model database

In recent days, there is a new data model is evolving called multi model databases which means a single database supports for key value pairs, document stores and graphs. From this multi model database series, we have chosen OrientDB and ArangoDB for qualitative comparison. There are many similarities in features between OrientDB and ArangoDB, but in the aspect of storing blob files OrientDB have additional mechanisms to handle huge blob files compared to ArangoDB.

BLOB store

As we know robots generate huge point clouds and images of huge size (approx. 10 MB and more), there are two mechanisms we could potentially follow to handle large data.

- Store pointclouds/images as blobs in the database itself. But storing blobs in database may significantly decrease the performance.
- Store blob in file system (with faster access SSD for good performance) and keep the reference in the database. But the disadvantage is, it cannot be replicated automatically by the database.

How selected Graph databases react for handling BLOB stores?

Even though Neo4j support byte-array type, it still consider storing huge blobs in database as an anti pattern [60]. ArangoDB doesn't have native support to handle blobs but it can be achievable through Foxx micro services. Using Foxx micro services we can store files in the storage (not in database) and save the reference in database. Surprisingly OrientDB provides multiple options out of the box to handle blob store.

OrientDB with BLOB

- Store the blob in file system and store the reference in document.
- Store the blob (up to 10MB) in the database itself, but with the risk of degrading performance, run time cost and waste of space + 33% [61].
- Store with ORecordBytes. ORecordBytes class is directly implemented from Record class and can handle huge blobs without conversions.

ORecordBytes creates a new record that contains blob, and reference this record to the original record. This technique improves the performance in terms of accessing blob from database. If the blob file really bigger, then OrientDB suggest developers to chunk the huge blob file into pieces and store them in multiple separate records using ORecordBytes. Then now reference all the chunks into one main record and this looks similar to one to many relationship since one main record have reference to many chunks of data.

7.5 Time series database

Time series database are meant to store streaming information from sensors, monitors, logs, etc., along with timestamp. In most cases they store numeric and float values in time series database. For qualitative comparison, we have considered SiriDB, InfluxDB and Druid.

Out of this three databases, SiriDB and Druid will be neglected for quantitative comparison because SiriDB is a new database with lack of many features (support for many important programming languages like C, C++ and Java, community support and documentation). Druid and InfluxDB provides mostly similar features in terms of Consistency, Availability, Schema less design and immutability. But Druid have less number of programming languages support than InfluxDB, also InfluxDB have Time Structured Merge tree engine that helps high speed ingestion and data compression. So we decided to consider InfluxDB from Time series databases for quantitative analysis.

8 Test environment setup

To analyze performance of the selected databases, we employ docker engine to build multiple containers and assume each container is a separate robot which generate number of sensor events. This section discuss about the following key aspects which are necessary for qualitative analysis setup,

- Architecture (Master-master/Master-slave)
- Number of events (Threads in each robot)
- Data types
- Number of robots
- Frequency of generating sensor events

8.1 Architecture

The main purpose of this research is find a good performing database for multi robots scenario. Robots can be in different locations and they might want to share their data to accomplish a task. It means, all robot should be connected in the same network always. But it isn't achievable always, because the internet connectivity can be poor or no connection available at some locations. So, the architecture should be robust enough to handle the problems in the network, even though there is no network connection robot should be able to store the data somehow until it connects to the network again. We think of three possible architectures and chose best one for the qualitative analysis setup.

8.1.1 Master slave architecture

Figure 8 shows the master slave setup. Each robot have multiple threads that produces different sensor events under varying frequency range. Generated sensor data are sent to master database which is located somewhere centrally to the robots. There can be many replica (slave) databases that copies the data from master and duplicates them in to slave databases. If a robot want to read data of other database, it should send a read request to master/slave database to get the data. But note that, all robots can write data only to the master database.

Here the problem is, if there is a network problem between robots and database nodes, they cannot write/read data from/to the database. The other potential problem is, robots generate data at high frequencies and if all robots send many write request to master node, there will be chance of data loss due to high traffic.

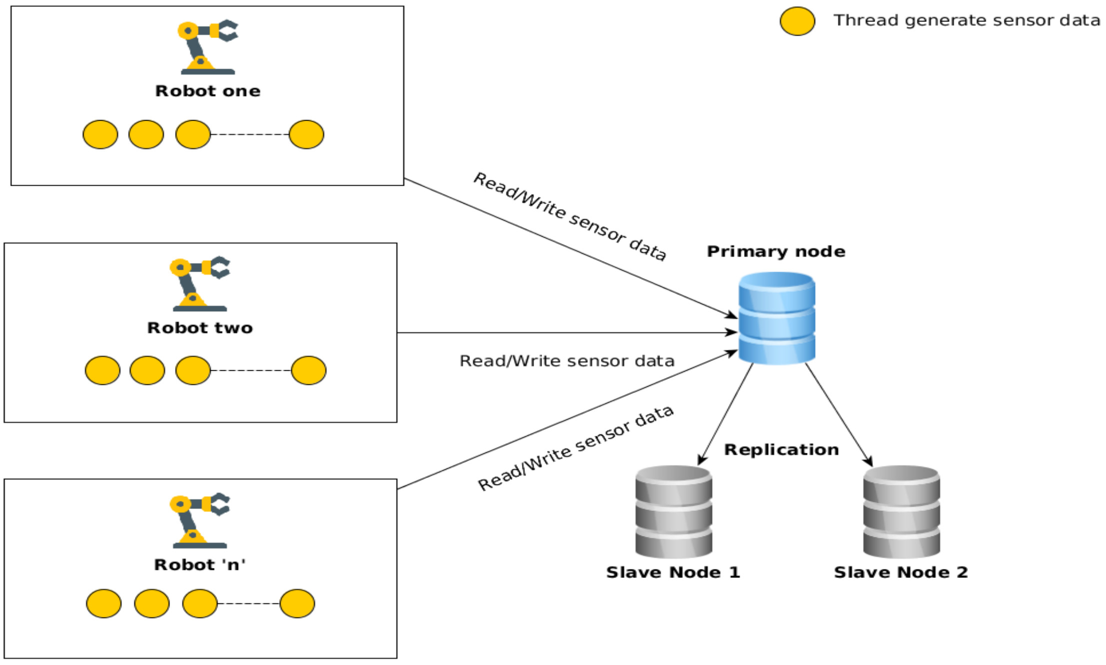


Figure 8: Master slave setup

8.1.2 Master master architecture (Without database in robot)

Figure 9 shows the master master setup without database in robots. Difference between this architecture and the previous one is, here there is no slave nodes. Instead all data nodes act as master node. So, robots can write/read data in any master node.

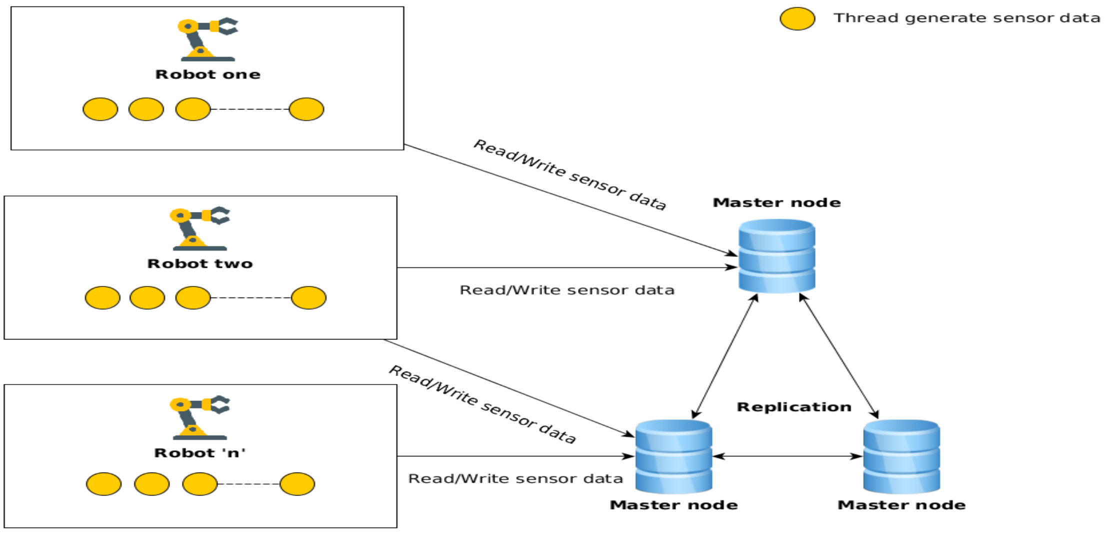


Figure 9: Master master setup (Without database in robot)

By using master master replication setup, we could solve one of the problem from master slave architecture. Here, write/read load is distributed with all master nodes. Hence we can reduce the chance of data loss compared to first architecture. But still this setup will face network issues.

8.1.3 Master master architecture (With database in robot)

Figure 10 shows the master master setup with database in each robot. This architecture follows same master master setup with small change in where the master nodes are configured. In this setup, each robot stores the data in the master node available locally in the robot and also the data will be replicated among other robots. Here, all problems from master slave architecture are addressed. Even though there is a problem in the network, robot can still store its data locally and replicate with other robots once it get the network back.

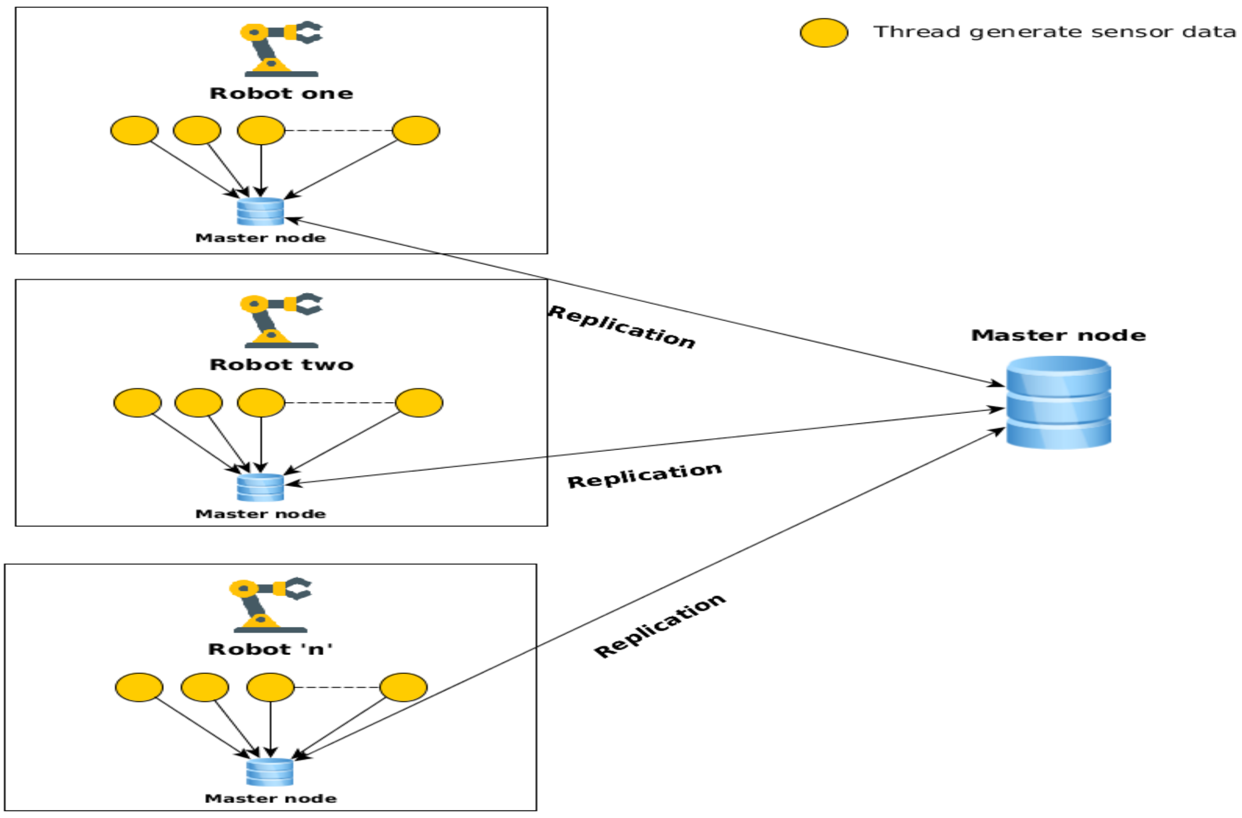


Figure 10: Master master setup (With database in robot)

Finally, we conclude that master master architecture with database in each robot is robust to write/read sensor data. So we use the same architecture for the qualitative analysis. We construct docker containers that generate and stores sensor events in the local database, as well as it will replicate the generated data with the other containers in the network.

8.2 Number of events

From each container (robot), we decided to have five sensor events that randomly generate sensor data. Five different threads will be spawned in a robot and they generate data independent to each other and send them to database. But this threads are synchronous, means once they send a data to database, they will wait till it receives response back from database and then it will generate next event. This will break the rule of generating sensor events at certain frequency. To overcome this, we are using a queue to store the generated events. Whenever a thread generate an event, it will push the event to the queue. To write these events to the database, we run separate threads that will pick an event from the queue and write it to database.

The five sensor events are,

- Location Event - Location of robot
- HandleBarVoltage Event - Voltage supplied to handle bar
- MotorBarVoltage Event - Voltage, current supplied to motors
- Pose Event - Current pose of robot
- RGB Event - RGB image

8.3 Data types

Generated events which are mentioned above consists of mixed data types such as String, Float, Integer, Boolean, Binary (Image) and DateTime.

Location Event

```
{
        'robot_id' : _robot_id #integer 24bytes
        'latitude' : random.uniform(1.0, 100.0), #float 24bytes
        'longitude' : random.uniform(1.0, 100.0), #float 24bytes
        'offset' : random.uniform(1.0, 100.0), #float 24bytes
        'accuracy' : random.uniform(1.0, 100.0), #float 24bytes
        'timestamp' : datetime.datetime.now() #datetime 48bytes
}
```

HandleBarVoltage Event

```
{
        'robot_id' : _robot_id #integer 24bytes
        'voltage' : random.uniform(1.0, 100.0), #float 24bytes
        'timestamp' : datetime.datetime.now() #datetime 48bytes
}
```

MotorBarVoltage Event

```
{
        'robot_id' : _robot_id #integer 24bytes
        'motor_id' : random.randint(1, 10), #float 24bytes
        'voltage' : random.uniform(1.0, 100.0), #float 24bytes
        'current' : random.uniform(1.0, 100.0), #float 24bytes
        'timestamp' : datetime.datetime.now() #datetime 48bytes
}
```

Pose Event

```
{
        'robot_id' : _robot_id #integer 24bytes
        'x' : random.uniform(1.0, 100.0), #float 24bytes
        'y' : random.uniform(1.0, 100.0), #float 24bytes
        'z' : random.uniform(1.0, 100.0), #float 24bytes
        'theta' : random.uniform(1.0, 100.0), #float 24bytes
        'timestamp' : datetime.datetime.now() #datetime 48bytes
}
```

RGB Event (Without blob)

```
{
        'robot_id' : _robot_id #integer 24bytes
        'image_base64': big_image_base64_path, #string >60bytes
        'blob' : False, #boolean 24bytes
        'timestamp' : datetime.datetime.now() #datetime 48bytes
}
```

RGB Event (With blob)

```
{
        'robot_id' : _robot_id #integer 24bytes
        'image_base64': big_image_base64, #string (Buffer) 1 Mega byte
        'blob' : False, #boolean 24 bytes
        'timestamp' : datetime.datetime.now() #datetime 48 bytes
}
```

All events are provided with 'robot_id' and 'timestamp' attribute by default to identify which robot generated the event at specific time. The other attributes such as robot location, current, voltage, current post are generated randomly. In RGB event, there are two sub events in which one stores RGB image in the database and the other one stores only the path where RGB image is stored in file system. We would like to evaluate database performance with storing images within the database itself as a blob.

8.4 Number of robots

Number of robots considered for qualitative analysis is completely dependent on the system (laptop) configuration since we are running all robots (containers) in a single docker engine. The configuration of the system which we are using for qualitative analysis is, 8GB RAM, 256GB SSD drive, intel CORE i5 7th Generation processor, and Ubuntu 14.04. Initially we started analysis with 5 containers, but RAM is not sufficient for 5 containers, then we reduced it to 3 containers. Finally we decided to use 3 containers (robots) for quantitative analysis.

8.5 Frequency of generating sensor events

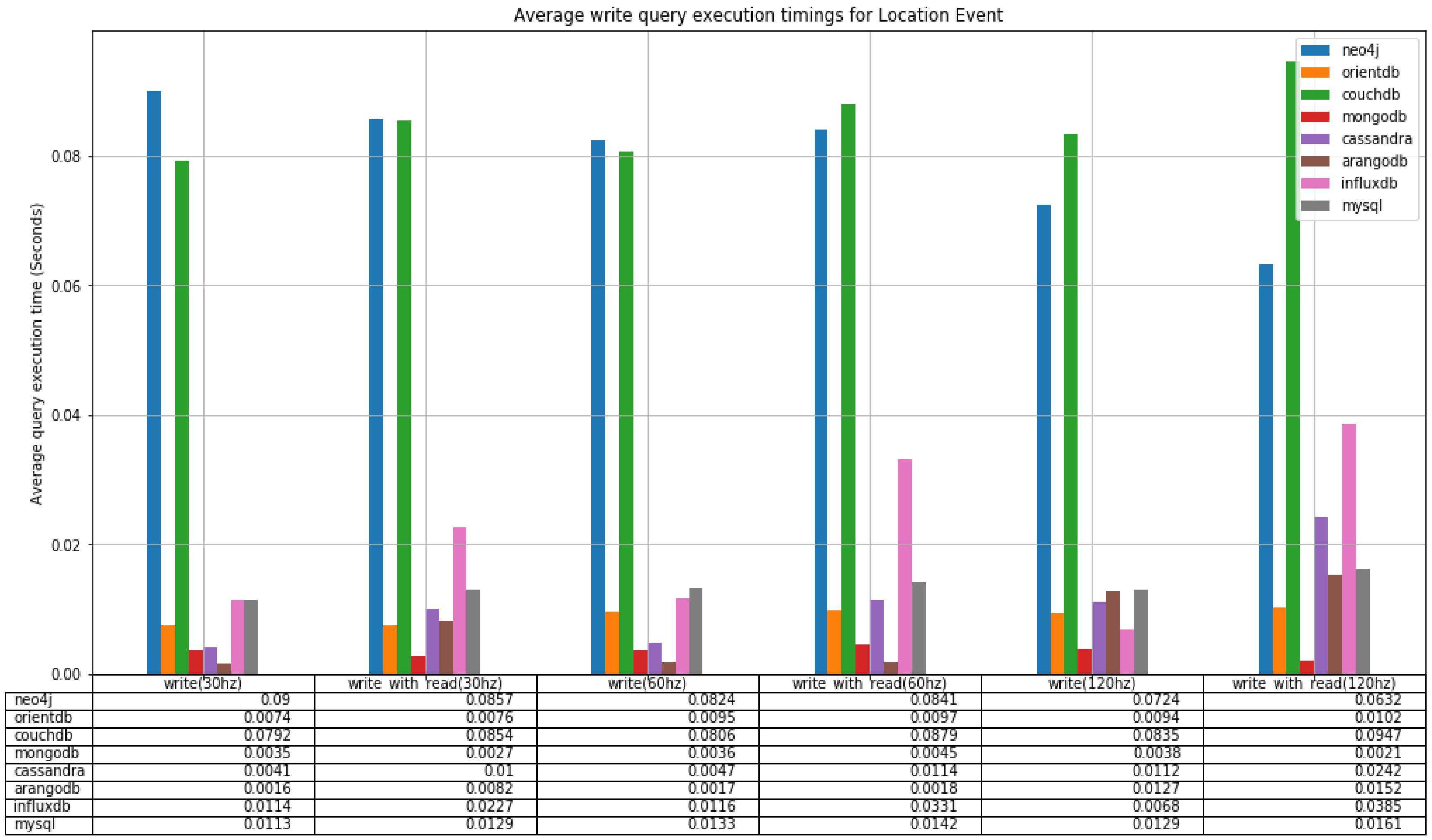
To evaluate databases performance under different traffic loads, we have decided to stream sensor events at 3 different ranges are, 30 Hz, 60 Hz and 120 Hz.

9 Results

This section discuss about the observed results from quantitative analysis. We divide the results in to three parts as follows,

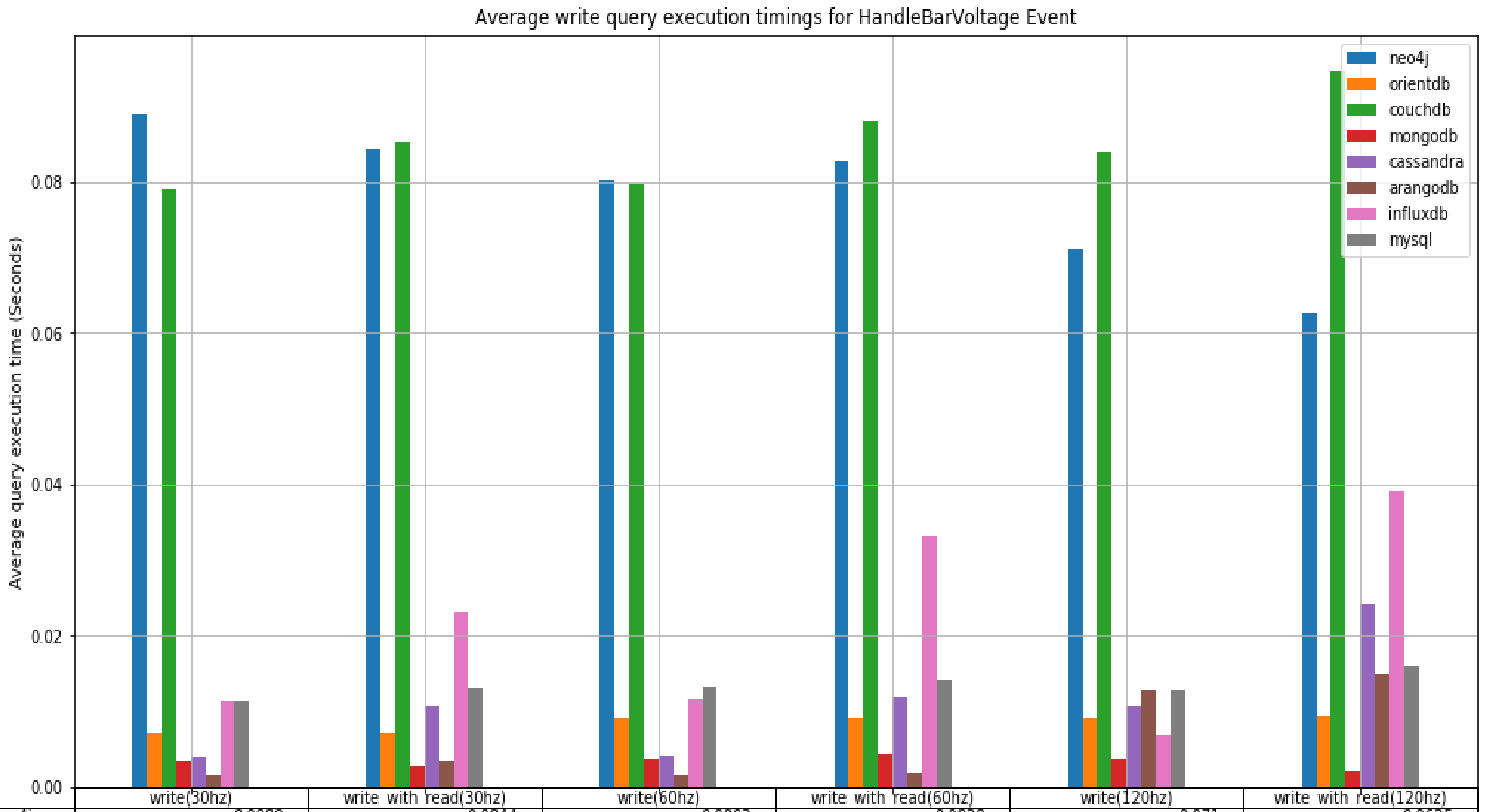
- Write only and Write along with read query execution at different frequency range
- Read only and Read along with write query execution at different frequency range
- Average replication time

9.1 Write only and Write along with read



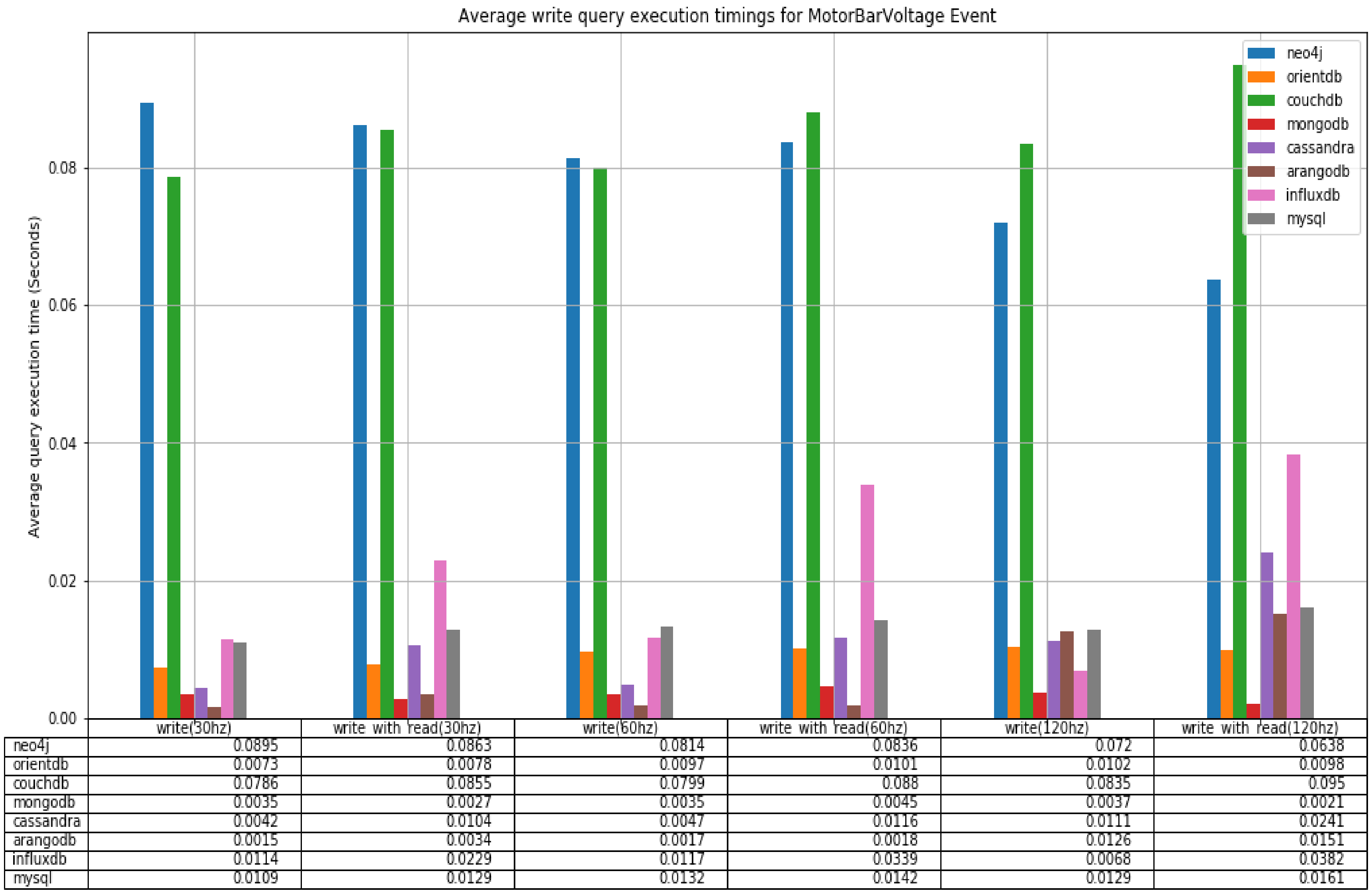
	write(30hz)	write with read(30hz)	write(60hz)	write with read(60hz)	write(120hz)	write with read(120hz)
neo4j	0.09	0.0857	0.0824	0.0841	0.0724	0.0632
orientdb	0.0074	0.0076	0.0095	0.0097	0.0094	0.0102
couchdb	0.0792	0.0854	0.0806	0.0879	0.0835	0.0947
mongodb	0.0035	0.0027	0.0036	0.0045	0.0038	0.0021
cassandra	0.0041	0.01	0.0047	0.0114	0.0112	0.0242
arangodb	0.0016	0.0082	0.0017	0.0018	0.0127	0.0152
influxdb	0.0114	0.0227	0.0116	0.0331	0.0068	0.0385
mysql	0.0113	0.0129	0.0133	0.0142	0.0129	0.0161

Figure 11: Average write query execution timings for Location event



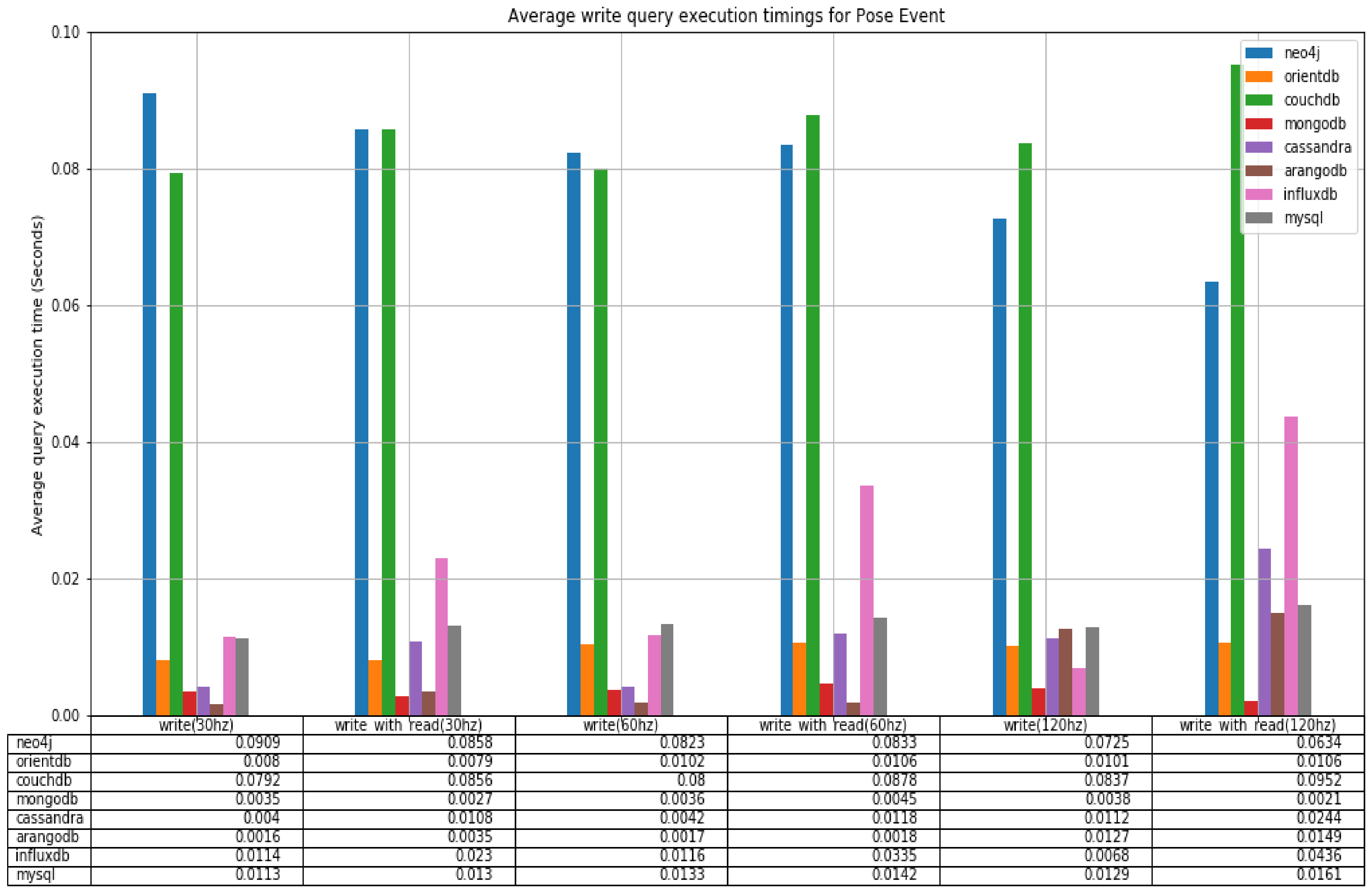
	write(30hz)	write with read(30hz)	write(60hz)	write with read(60hz)	write(120hz)	write with read(120hz)
neo4j	0.0888	0.0844	0.0803	0.0828	0.071	0.0625
orientdb	0.0071	0.007	0.009	0.0092	0.0092	0.0094
couchdb	0.079	0.0853	0.0798	0.0879	0.0838	0.0947
mongodb	0.0034	0.0026	0.0035	0.0044	0.0037	0.0021
cassandra	0.0039	0.0107	0.0041	0.0118	0.0107	0.0243
arangodb	0.0015	0.0034	0.0016	0.0018	0.0127	0.0148
influxdb	0.0113	0.023	0.0116	0.0332	0.0068	0.039
mysql	0.0113	0.0129	0.0133	0.0142	0.0128	0.016

Figure 12: Average write query execution timings for HandleBarVoltage event



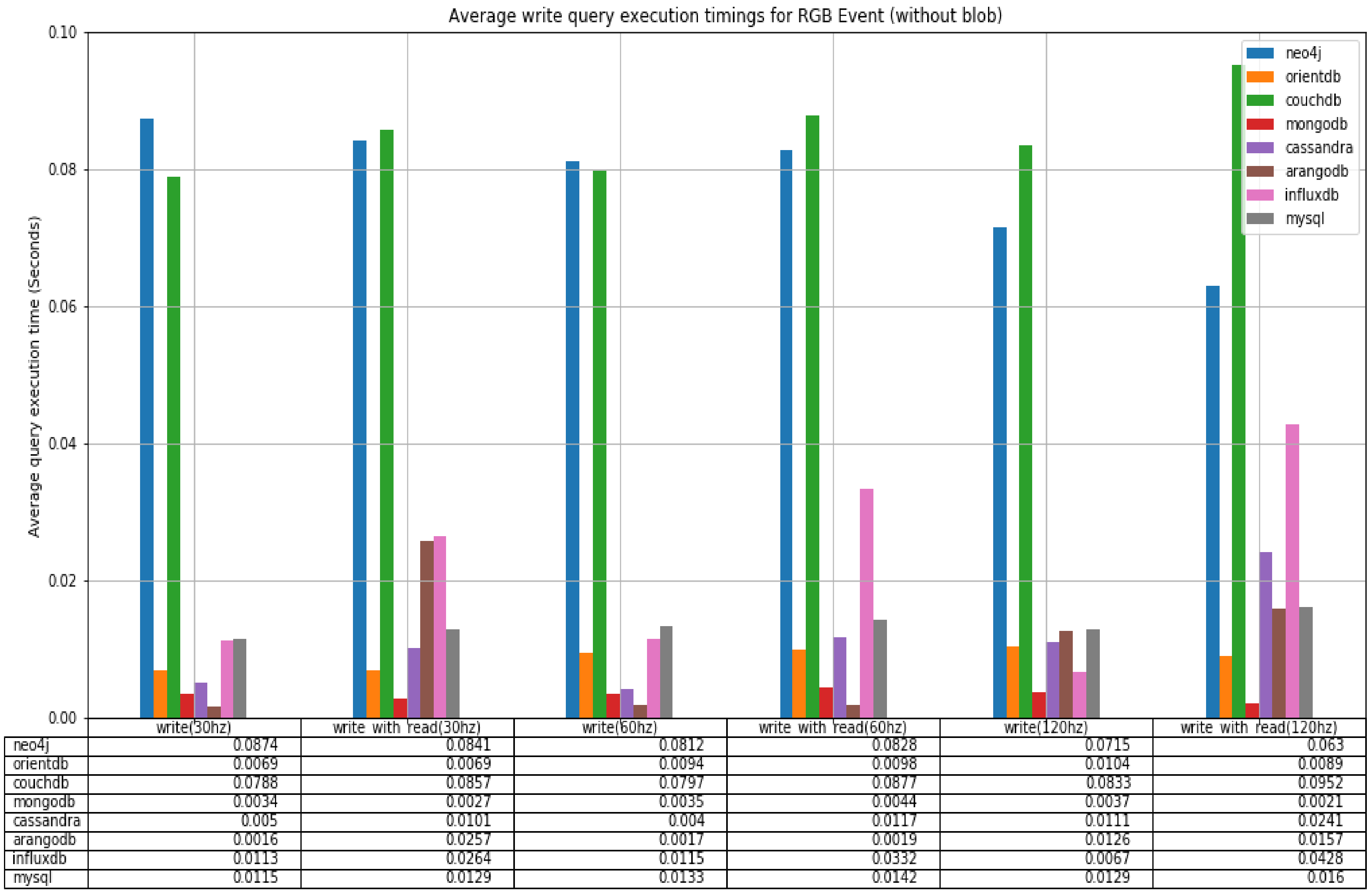
	write(30hz)	write_with_read(30hz)	write(60hz)	write_with_read(60hz)	write(120hz)	write_with_read(120hz)
neo4j	0.0895	0.0863	0.0814	0.0836	0.072	0.0638
orientdb	0.0073	0.0078	0.0097	0.0101	0.0102	0.0098
couchdb	0.0786	0.0855	0.0799	0.088	0.0835	0.095
mongodb	0.0035	0.0027	0.0035	0.0045	0.0037	0.0021
cassandra	0.0042	0.0104	0.0047	0.0116	0.0111	0.0241
arangodb	0.0015	0.0034	0.0017	0.0018	0.0126	0.0151
influxdb	0.0114	0.0229	0.0117	0.0339	0.0068	0.0382
mysql	0.0109	0.0129	0.0132	0.0142	0.0129	0.0161

Figure 13: Average write query execution timings for MotorBarVoltage event



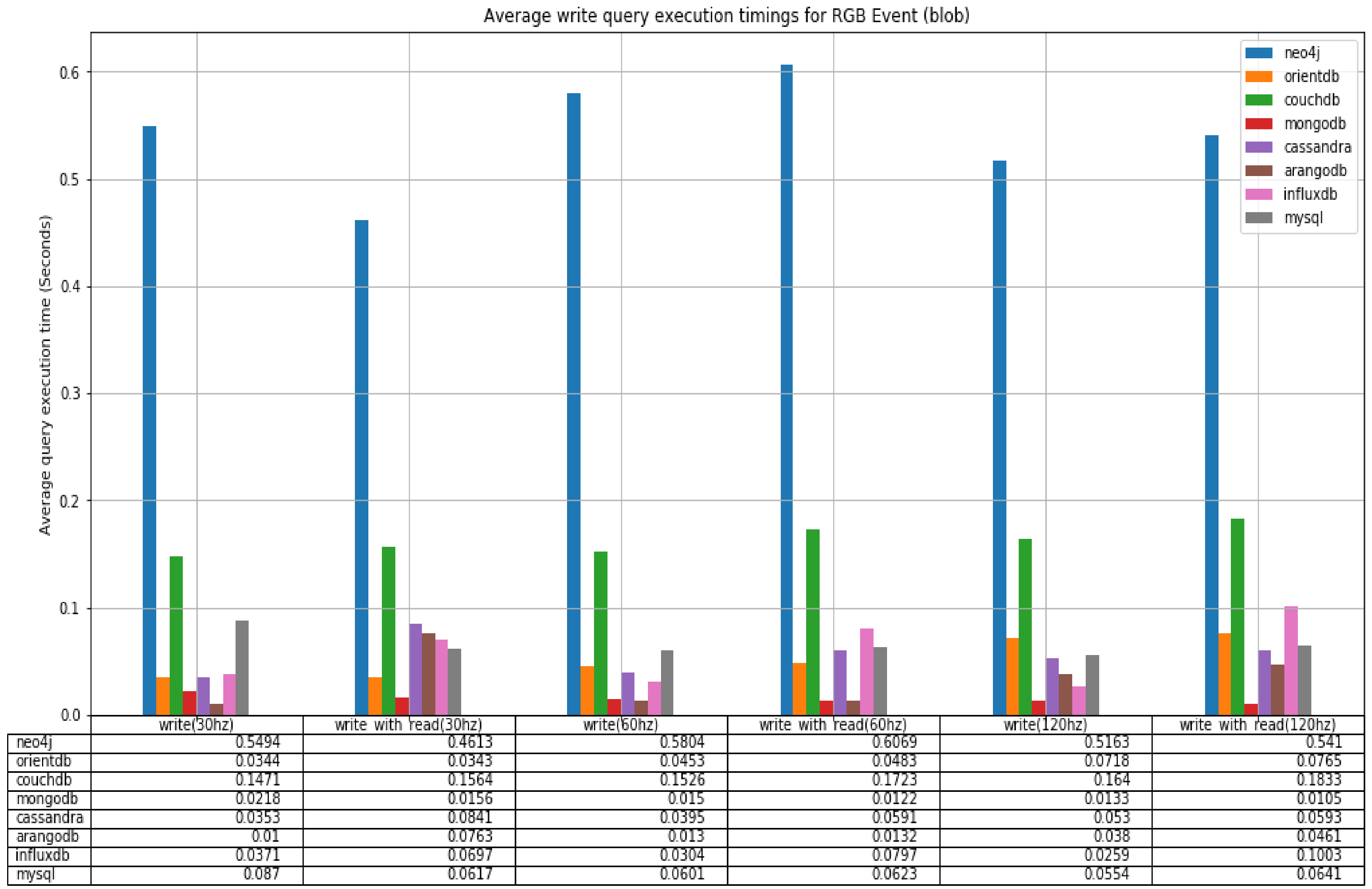
	write(30hz)	write with read(30hz)	write(60hz)	write with read(60hz)	write(120hz)	write with read(120hz)
neo4j	0.0909	0.0858	0.0823	0.0833	0.0725	0.0634
orientdb	0.008	0.0079	0.0102	0.0106	0.0101	0.0106
couchdb	0.0792	0.0856	0.08	0.0878	0.0837	0.0952
mongodb	0.0035	0.0027	0.0036	0.0045	0.0038	0.0021
cassandra	0.004	0.0108	0.0042	0.0118	0.0112	0.0244
arangodb	0.0016	0.0035	0.0017	0.0018	0.0127	0.0149
influxdb	0.0114	0.023	0.0116	0.0335	0.0068	0.0436
mysql	0.0113	0.013	0.0133	0.0142	0.0129	0.0161

Figure 14: Average write query execution timings for Pose event



	write(30hz)	write with read(30hz)	write(60hz)	write with read(60hz)	write(120hz)	write with read(120hz)
neo4j	0.0874	0.0841	0.0812	0.0828	0.0715	0.063
orientdb	0.0069	0.0069	0.0094	0.0098	0.0104	0.0089
couchdb	0.0788	0.0857	0.0797	0.0877	0.0833	0.0952
mongodb	0.0034	0.0027	0.0035	0.0044	0.0037	0.0021
cassandra	0.005	0.0101	0.004	0.0117	0.0111	0.0241
arangodb	0.0016	0.0257	0.0017	0.0019	0.0126	0.0157
influxdb	0.0113	0.0264	0.0115	0.0332	0.0067	0.0428
mysql	0.0115	0.0129	0.0133	0.0142	0.0129	0.016

Figure 15: Average write query execution timings for RGB event (Without BLOB)



	write(30hz)	write with read(30hz)	write(60hz)	write with read(60hz)	write(120hz)	write with read(120hz)
neo4j	0.5494	0.4613	0.5804	0.6069	0.5163	0.541
orientdb	0.0344	0.0343	0.0453	0.0483	0.0718	0.0765
couchdb	0.1471	0.1564	0.1526	0.1723	0.164	0.1833
mongodb	0.0218	0.0156	0.015	0.0122	0.0133	0.0105
cassandra	0.0353	0.0841	0.0395	0.0591	0.053	0.0593
arangodb	0.01	0.0763	0.013	0.0132	0.038	0.0461
influxdb	0.0371	0.0697	0.0304	0.0797	0.0259	0.1003
mysql	0.087	0.0617	0.0601	0.0623	0.0554	0.0641

Figure 16: Average write query execution timings for RGB event (With BLOB)

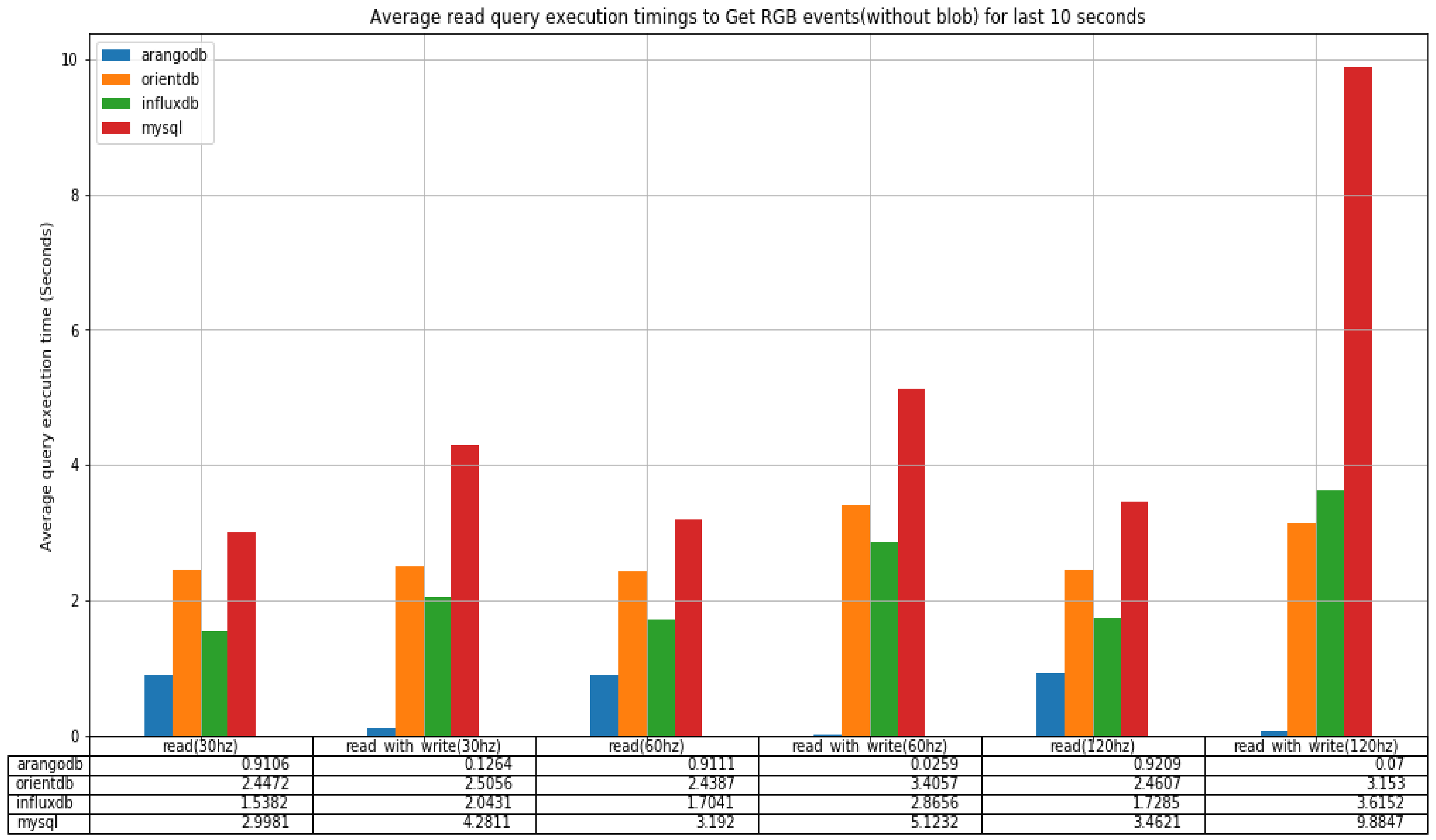
Above graphs shows the average write query execution time comparison between the databases considered for quantitaive analysis. There are 6 set of bar plots in each graph and each set shows execution time under different frequencies. For example, in figure 11 the fisrt 2 set of bar plots shows execution time with 30 Hz (write only) and 30 Hz (write along with read). Likewise, remaining bar plots in the same graph shows write execution time for other two frequency range of 60 Hz and 120 Hz and each with write only and write along with read query execution.

From figure 16 observed that, Neo4j database write query execution time reduces constantly if the frequency increases except RGB event with BLOB. In RGB event with BLOB, there is a raise in time at 60 Hz and again drop at 120 Hz. Contradicting to this, other database write query execution time increases proportional to frequency of generating events. Neo4j and OrientDB performs very bad compared to all other databases. But surprisingly, OrientDB handle BLOB files better than Neo4J.

While comparing RGB event with and without storing BLOB in database, writing BLOB in database showed increase in write time by more than 200 percent compared to storing only the pointer to BLOB. For example in figure 15 and 16 , consider Neo4j database consumes 0.063 seconds to store a BLOB pointer in database at 120 Hz and in case of storing BLOB itself it consumes 0.541 seconds. But even for storing BLOB files directly in database MongoDB and ArangoDB took only 0.01 and 0.0461 seconds, see Figure 16 respectively. This shows that MongoDB and ArangoDB is more efficient in storing BLOB files directly to the database compared to other candidates.

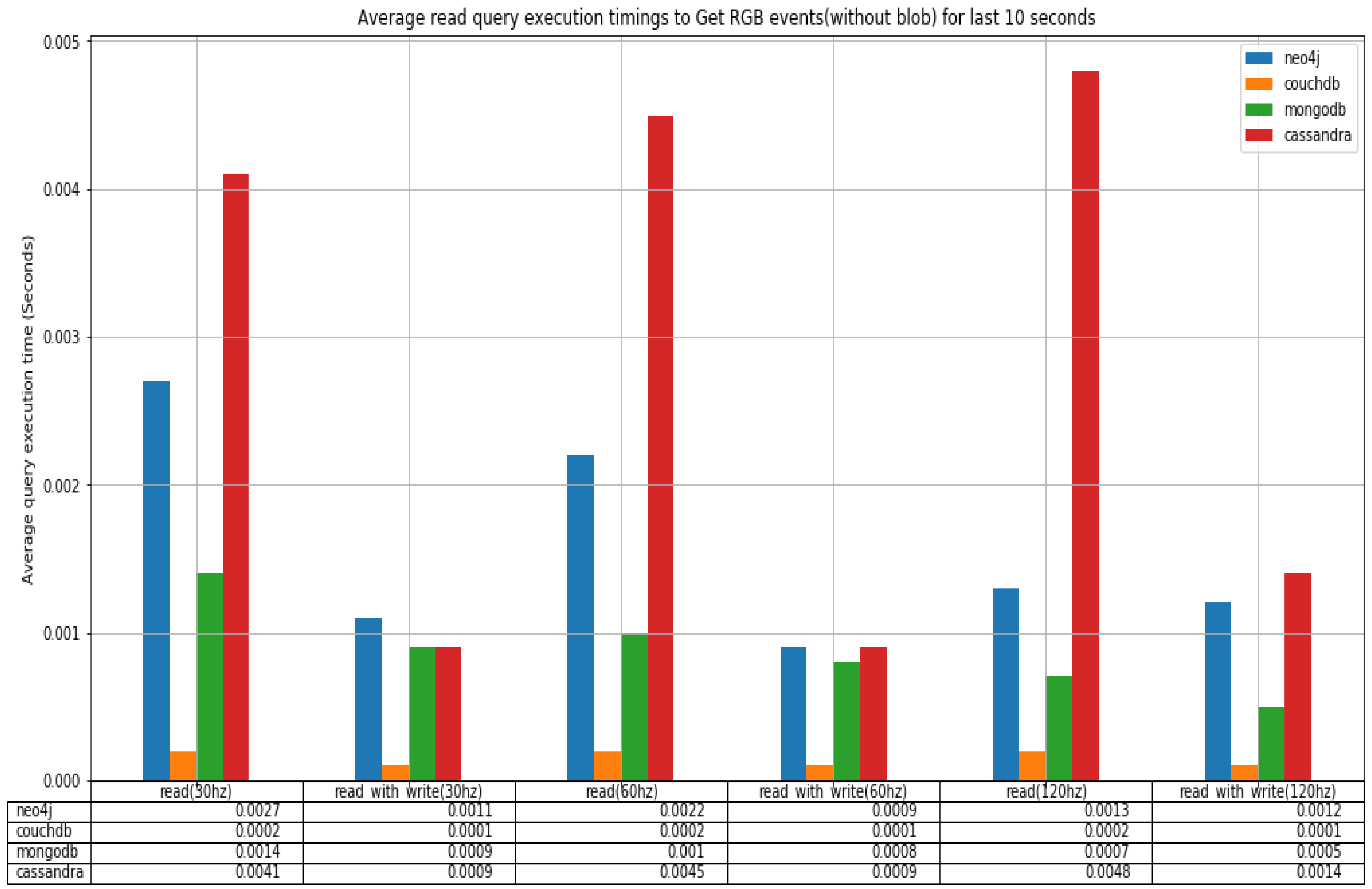
In most cases, Cassandra, OrientDB and MySQL databases shows more or less similar write timings under all three different frequencies respectively. ArangoDB write timings are also constant till 60 Hz but at 120 Hz it shows double the write time in all events. But still ArangoDB write times are smaller than other databases. As a overall observation, MongoDB and ArangoDB performed good in case of writing all three events under different frequencies. Also keep in mind that, MongoDB supports only master-slave architecture, so there is no replication workload on MongoDB master node, but ArangoDB is tested under master-master architecture. Apart from this two databases, Cassandra, OrientDB and MySQL are in the same range and performs better than Neo4j and CouchDB.

9.2 Read only and Read along with write



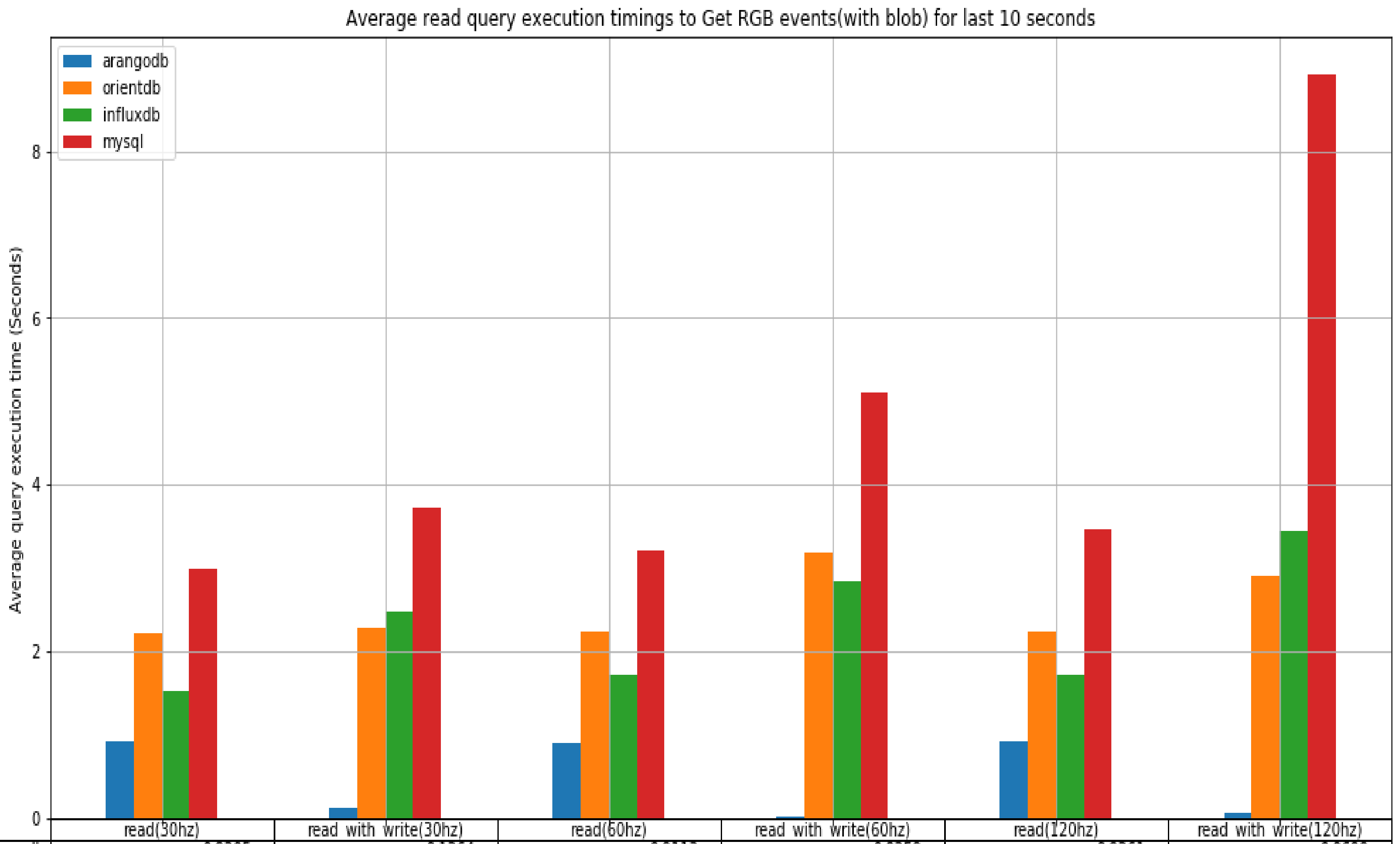
	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
arangodb	0.9106	0.1264	0.9111	0.0259	0.9209	0.07
orientdb	2.4472	2.5056	2.4387	3.4057	2.4607	3.153
influxdb	1.5382	2.0431	1.7041	2.8656	1.7285	3.6152
mysql	2.9981	4.2811	3.192	5.1232	3.4621	9.8847

Figure 17: Average read query execution timings to get RGB events (without blob) for last 10 seconds



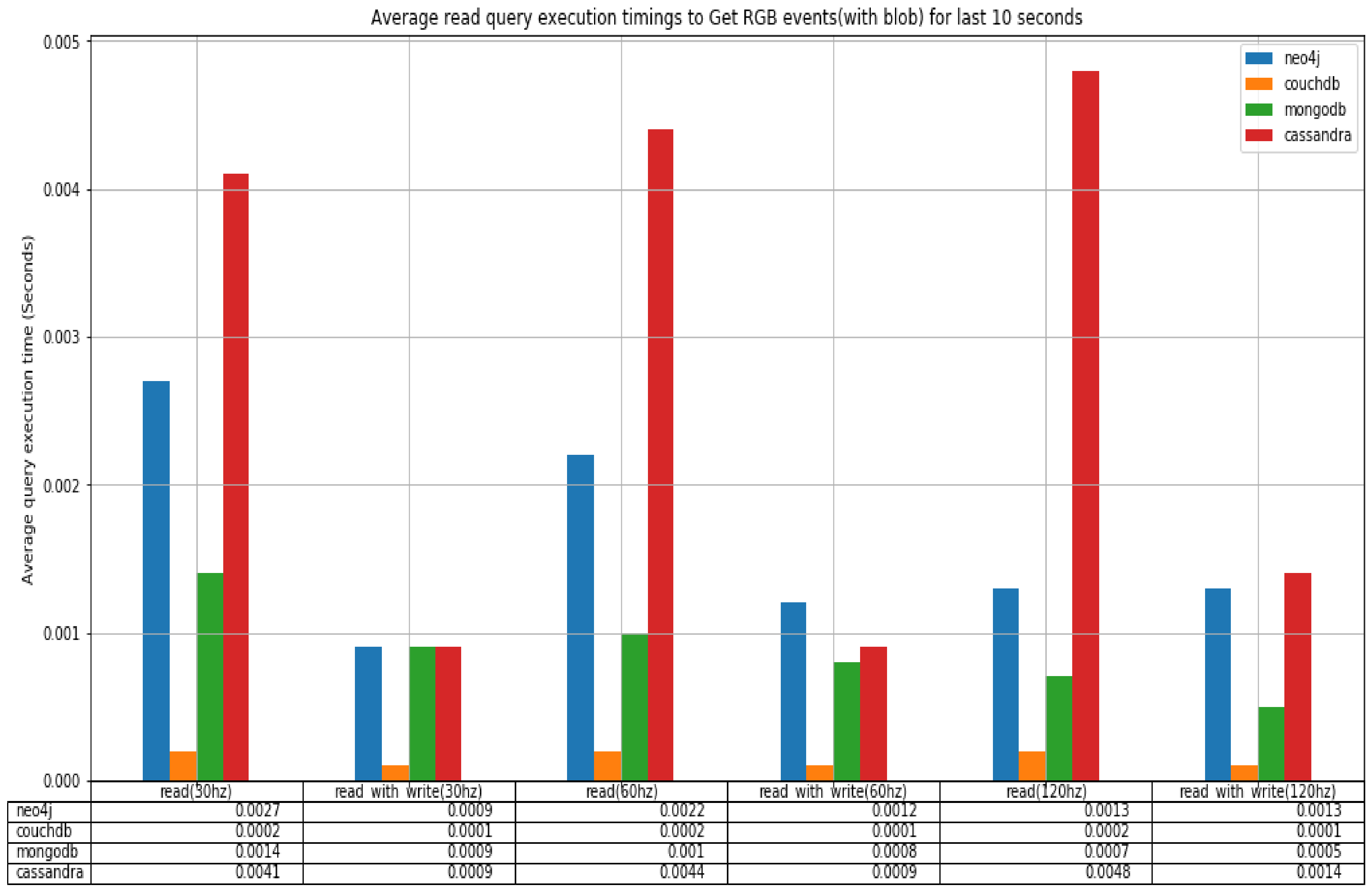
	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
neo4j	0.0027	0.0011	0.0022	0.0009	0.0013	0.0012
couchdb	0.0002	0.0001	0.0002	0.0001	0.0002	0.0001
mongodb	0.0014	0.0009	0.001	0.0008	0.0007	0.0005
cassandra	0.0041	0.0009	0.0045	0.0009	0.0048	0.0014

Figure 18: Average read query execution timings to get RGB events (without blob) for last 10 seconds



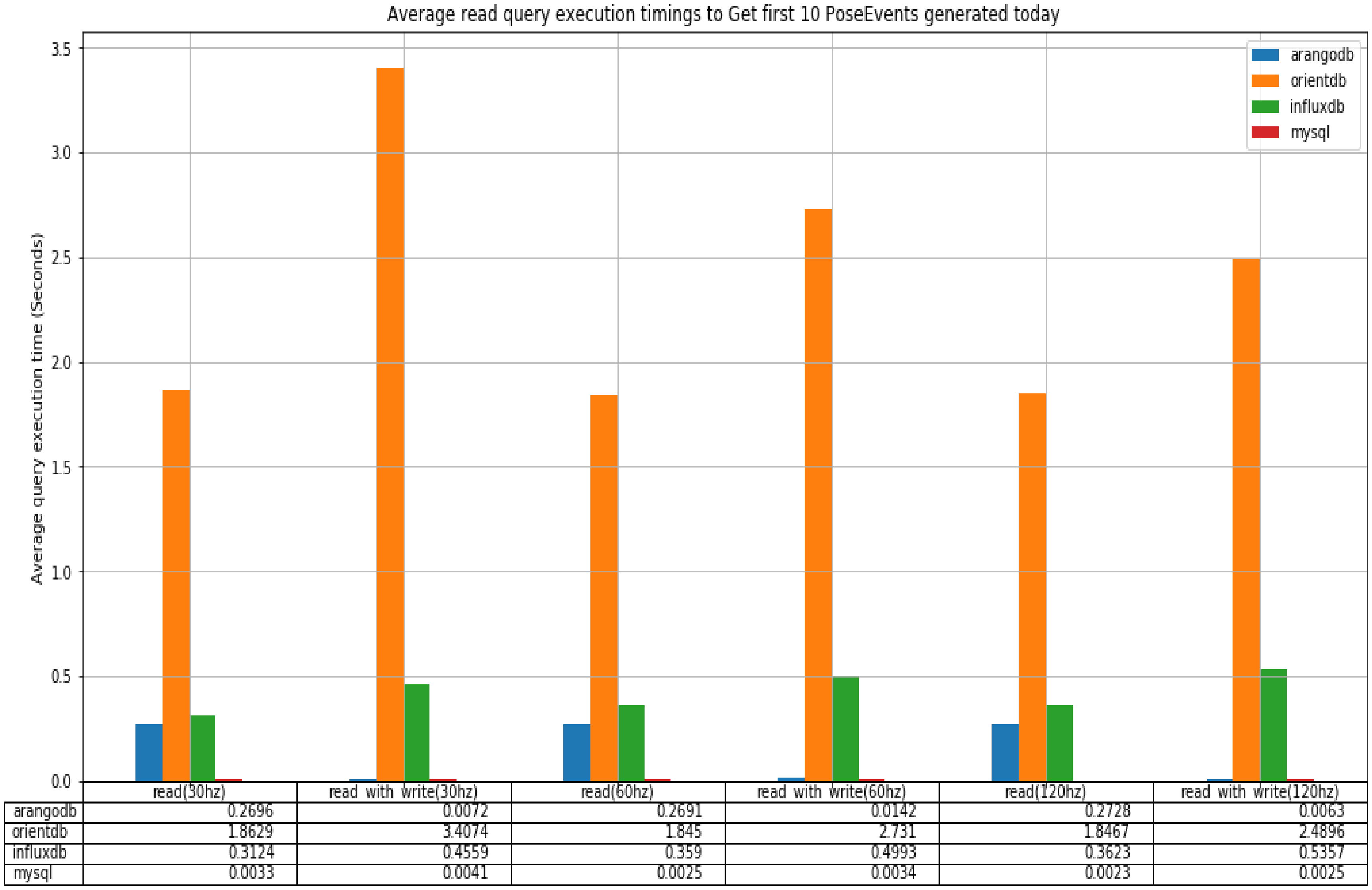
	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
arangodb	0.9205	0.1264	0.9113	0.0259	0.9261	0.0699
orientdb	2.2152	2.2788	2.2299	3.1863	2.241	2.9127
influxdb	1.5291	2.4808	1.7215	2.8464	1.7187	3.4412
mysql	2.9967	3.7208	3.2154	5.112	3.4758	8.9265

Figure 19: Average read query execution timings to get RGB events (with blob) for last 10 seconds



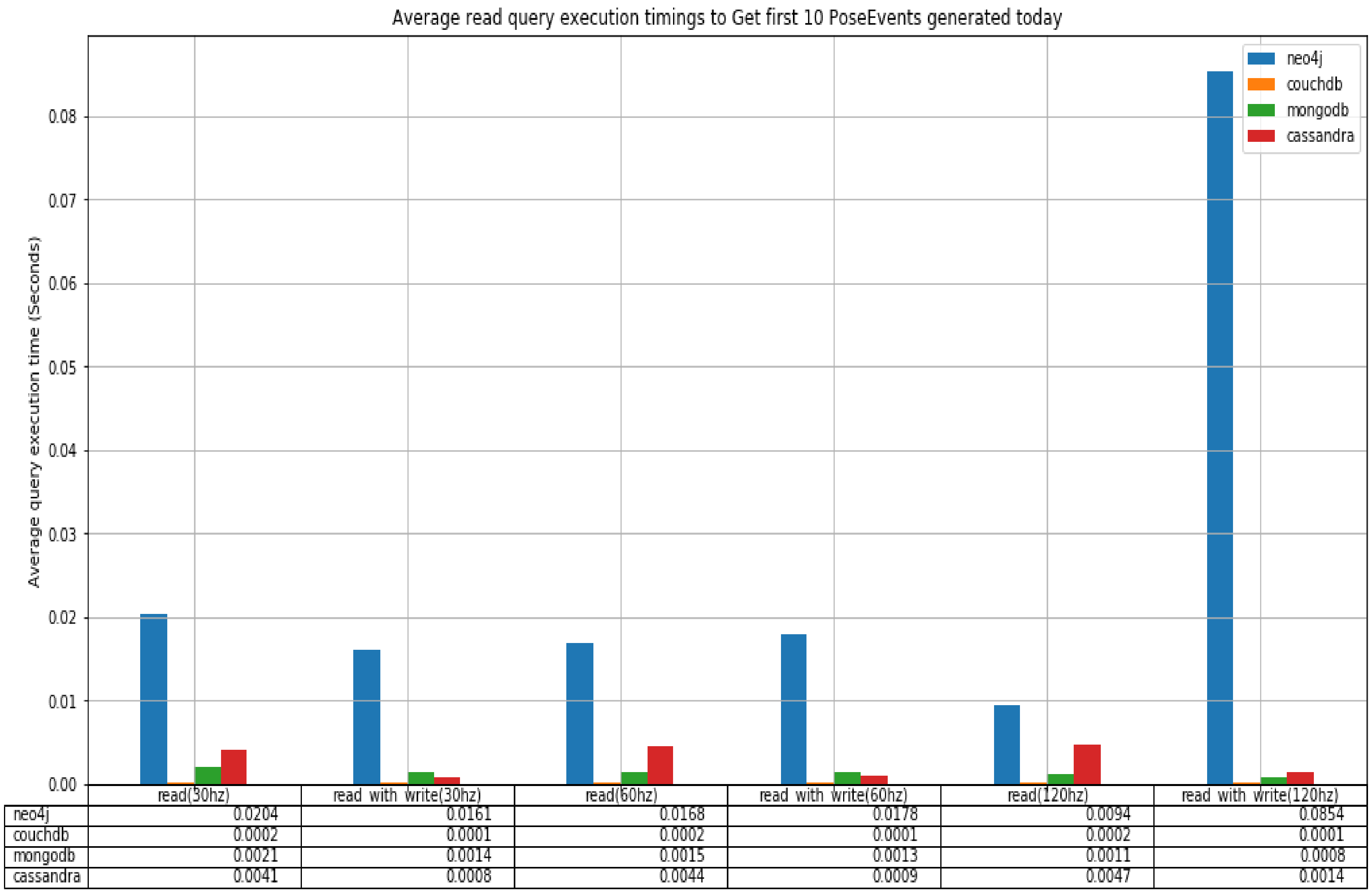
	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
neo4j	0.0027	0.0009	0.0022	0.0012	0.0013	0.0013
couchdb	0.0002	0.0001	0.0002	0.0001	0.0002	0.0001
mongodb	0.0014	0.0009	0.001	0.0008	0.0007	0.0005
cassandra	0.0041	0.0009	0.0044	0.0009	0.0048	0.0014

Figure 20: Average read query execution timings to get RGB events (with blob) for last 10 seconds



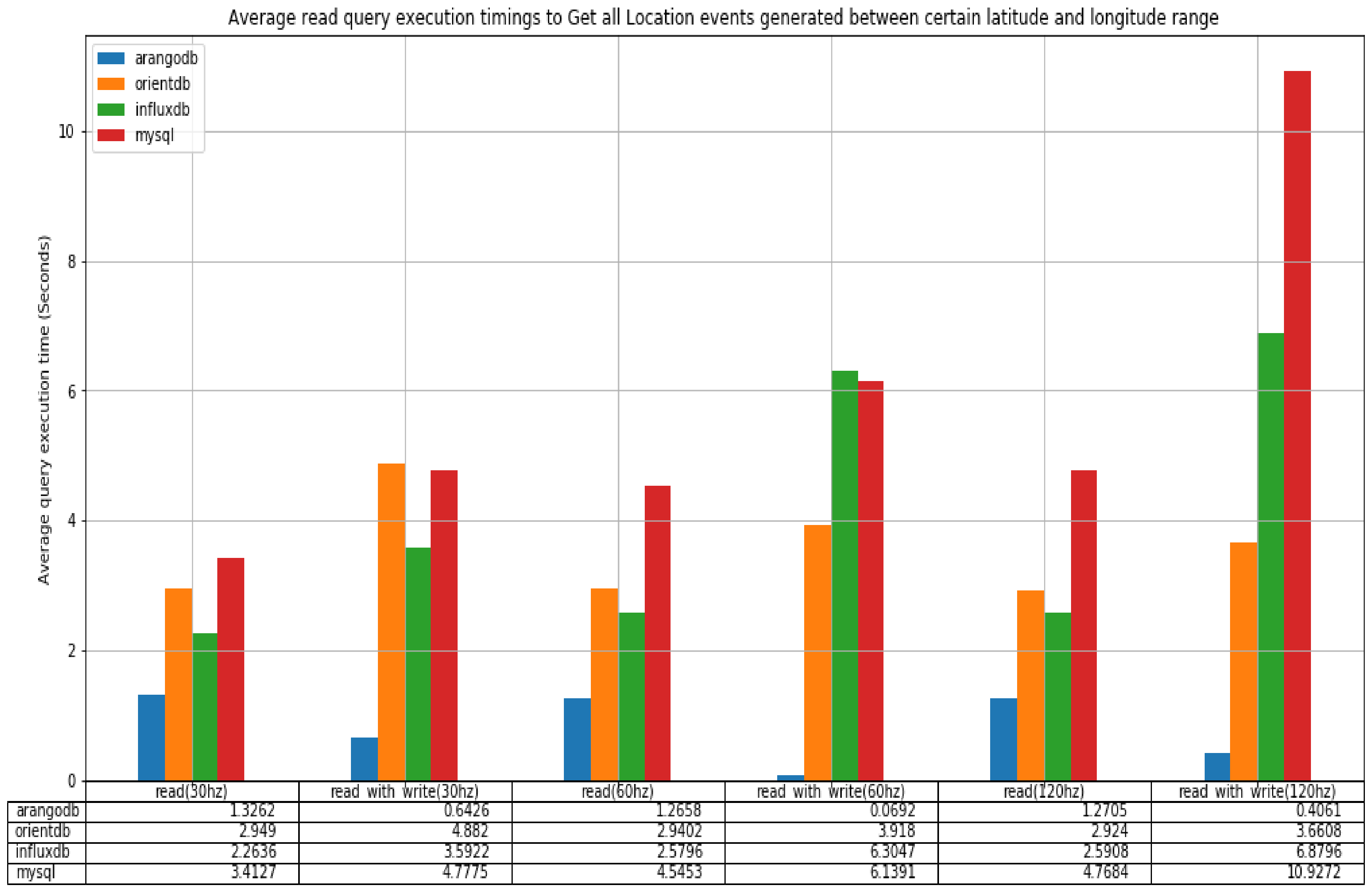
	read(30hz)	read_with_write(30hz)	read(60hz)	read_with_write(60hz)	read(120hz)	read_with_write(120hz)
arangodb	0.2696	0.0072	0.2691	0.0142	0.2728	0.0063
orientdb	1.8629	3.4074	1.845	2.731	1.8467	2.4896
influxdb	0.3124	0.4559	0.359	0.4993	0.3623	0.5357
mysql	0.0033	0.0041	0.0025	0.0034	0.0023	0.0025

Figure 21: Average read query execution timings to get first 10 Pose events generated today



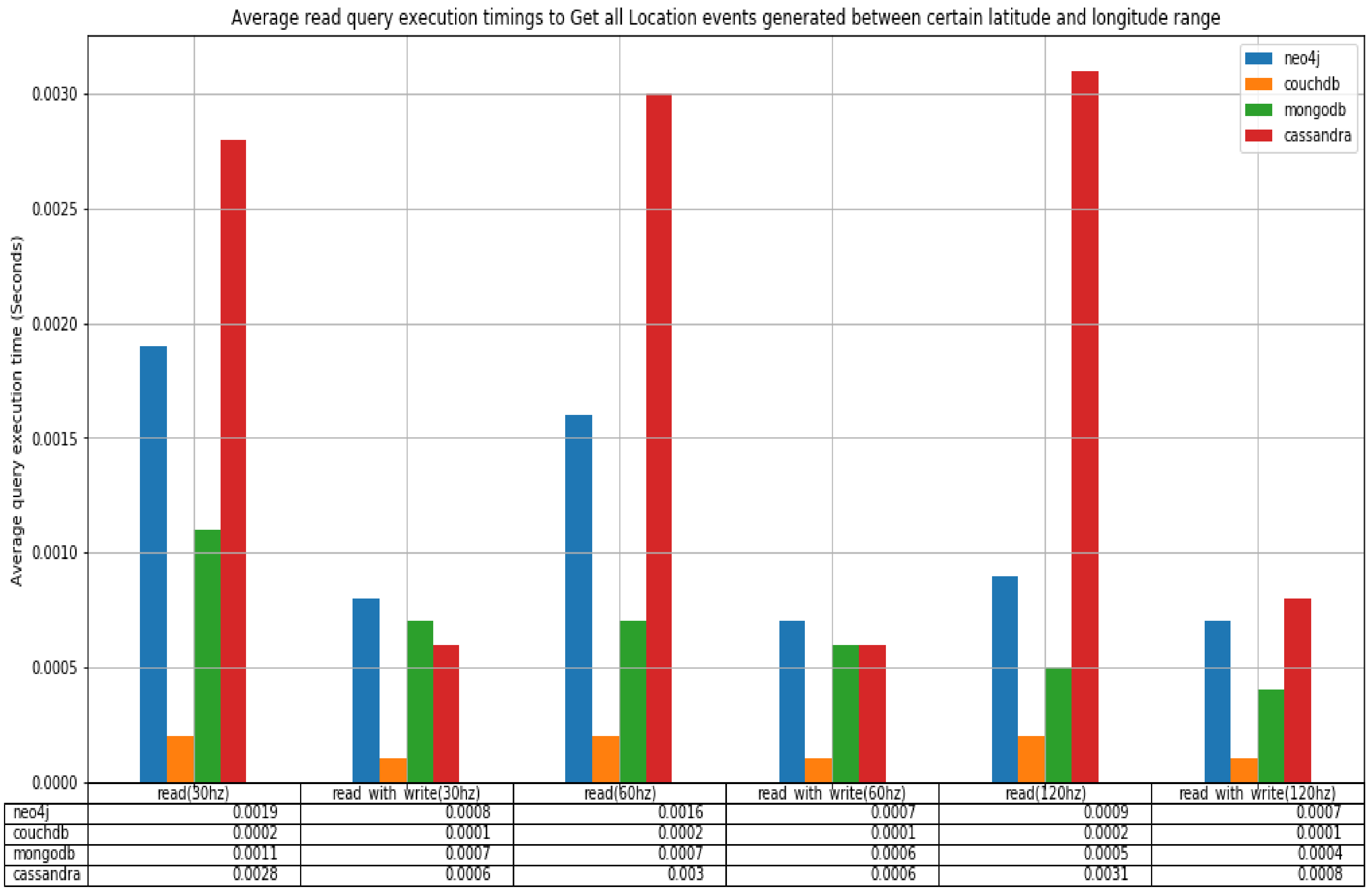
	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
neo4j	0.0204	0.0161	0.0168	0.0178	0.0094	0.0854
couchdb	0.0002	0.0001	0.0002	0.0001	0.0002	0.0001
mongodb	0.0021	0.0014	0.0015	0.0013	0.0011	0.0008
cassandra	0.0041	0.0008	0.0044	0.0009	0.0047	0.0014

Figure 22: Average read query execution timings to get first 10 Pose events generated today



	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
arangodb	1.3262	0.6426	1.2658	0.0692	1.2705	0.4061
orientdb	2.949	4.882	2.9402	3.918	2.924	3.6608
influxdb	2.2636	3.5922	2.5796	6.3047	2.5908	6.8796
mysql	3.4127	4.7775	4.5453	6.1391	4.7684	10.9272

Figure 23: Average read query execution timings to get all Location event between certain latitude and longitude



	read(30hz)	read with write(30hz)	read(60hz)	read with write(60hz)	read(120hz)	read with write(120hz)
neo4j	0.0019	0.0008	0.0016	0.0007	0.0009	0.0007
couchdb	0.0002	0.0001	0.0002	0.0001	0.0002	0.0001
mongodb	0.0011	0.0007	0.0007	0.0006	0.0005	0.0004
cassandra	0.0028	0.0006	0.003	0.0006	0.0031	0.0008

Figure 24: Average read query execution timings to get all Location event between certain latitude and longitude

Above graphs shows that the read query execution time for all databases under four scenarios. Each scenario consists of two graphs and each graph shows metrics for 4 databases. The reason for splitting into two graphs is, the first set of databases (ArangoDB, OrientDB, InfluxDB, and MySQL) have metrics ranges from 0 to 10 seconds, but the second set of databases (Neo4j, CouchDB, MongoDB, and Cassandra) shows metrics which ranges from 0 to 0.005 seconds. If we show all these databases values in a single graph, then the second set of databases values will not be visible in the graph due to very smaller values compared to first set. So, to enlarge the visibility of bars we split them in to two different graphs for each scenario.

CouchDB shows stable and good read query execution time better than all other databases in all four scenarios. The reason behind this is, CouchDB uses B-trees to generate key-sorted views that is built once and available to users to lookup efficiently. If users add, delete, or update a document, then B-tree will be indexed automatically and reflect the current state of database [62].

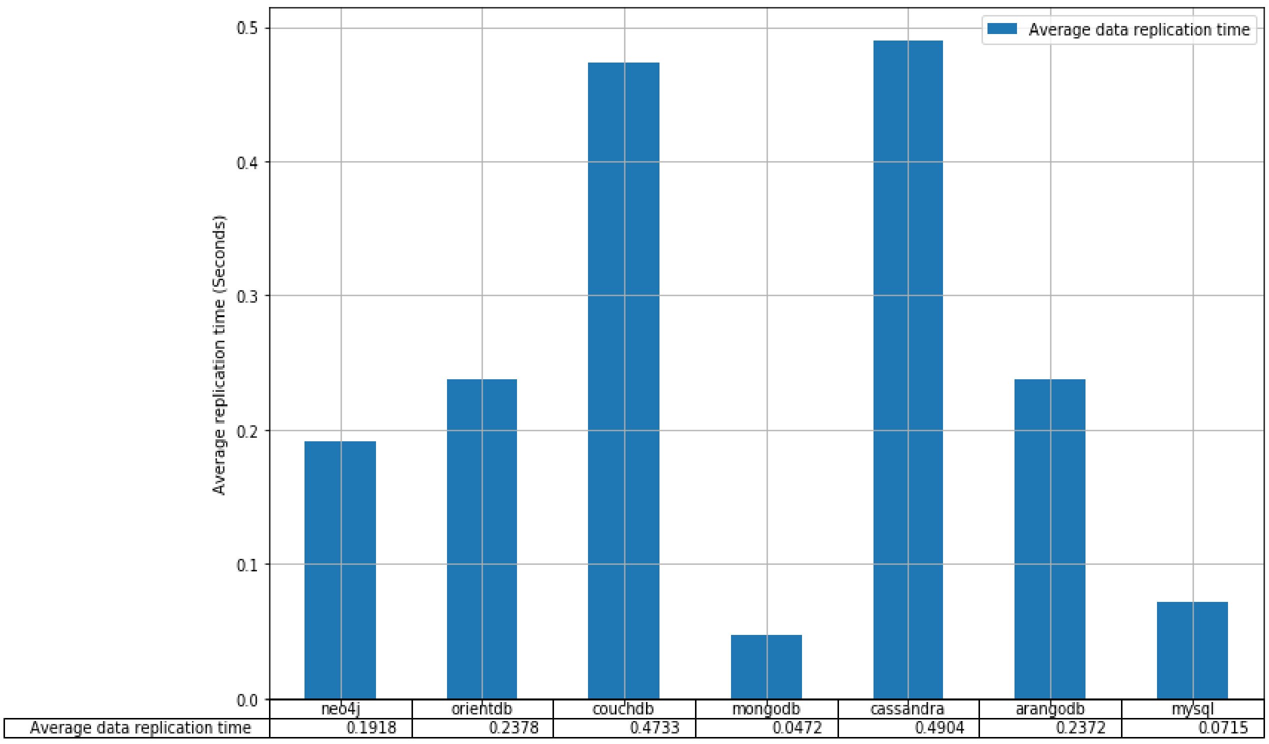
Apart from this, in all 4 scenarios the second set of databases executes read query much faster than the first set. Lets consider scenario 1 (Get RGB events (without blob) for last 10 seconds), the least number from first set of databases is only 0.025 seconds by ArangoDB, but the maximum number from second set is 0.0013 seconds by Cassandra. This differences shows clearly how bad the read query performance in first set of databases.

Another important observation from most of the databases is, the read queries time for read along with write scenario is double the time faster than read only scenario. The reason could be, the databases cache recently written data and while reading them, the queries read the data from intermediate cache memory. This improves the over all read query performance, but this is not same in all databases. Contradicting to the above fact, MySQL and OrientDB read times for read only scenario is faster than read with write scenario.

MySQL database consumes average of 5 to 6 seconds for complex queries and reading BLOB files from database, see Figure 23. But it performed relatively good than OrientDB, InfluxDB and ArangoDB in simple usecase like 'Get first 10 poses generated today'.

As a conclusion, OrientDB and MongoDB read times are stable and smaller compared to other databases. After this two, Neo4j and Cassandra performed well. ArangoDB, InfluxDB, OrientDB and MySQL databases consumes more query execution time in case of read and read with write scenarios compared with all four databases from second set.

9.3 Average replication time



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

Figure 25: Average replication time

Since InfluxDB community edition doesn't support cluster architecture, it has been excluded for replication test.

Replication time of MongoDB looks impressive, but since it supports only master-slave architecture we couldn't use it for multi robot scenarios.

The second good performer is MySQL (0.0715 seconds), but the performance in storing and retrieving blob is very poor. So, in case group of robots doesn't want to share/replicate blob files, mysql is a good candidate for fast replication.

Neo4j, ArangoDB, OrientDB consumed nearly equal time to replicate data. In this group of database, 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.

10 Conclusions

In this research work we tried to find foremost databases which are suitable for multi robot scenarios. Additionally we discussed in detail about different possible architectures to design a robust communication channel between robot and database. We also conclude that, master master replication architecture with database installed in the robot promises the availability of database over consistency.

Furthermore, the results from qualitative analysis shows the unique features, advantages and pitfalls from each selected databases. After a thorough analysis based on what each database is capable of and which database could be adopted for robotic applications, we filtered a set of candidates to explore more about them. Quantitative analysis results shows that MongoDB and ArangoDB performed well compared to all other databases. As a best practice, it is suggested to store BLOB files in file system and store the location pointers in the database. If it is mandate to store BLOB in database, use MongoDB and ArangoDB.

In terms of read scenarios, CouchDB shows lesser and stable query execution time than other databases and it performed not so bad in write scenarios. Even though MySQL is faster in replication, it performed significantly poor in specific read scenarios and especially in reading BLOBS. For robust replication, consider MongoDB, Neo4j, ArangoDB and OrientDB databases.

Overall, we can say that ArangoDB, MongoDB and CouchDB will be more flexible candidates for multi robot systems to handle sensor data.

References

[1] Tim Niemueller, Gerhard Lakemeyer, and Siddhartha 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*, pages 364–369. IEEE, 2012.

[2] André Dietrich, Siba Mohammad, Sebastian Zug, and Jörg Kaiser. Ros meets cassandra: Data management in smart environments with nosql.

[3] Alexander J Fiannaca and Justin Huang. Benchmarking of relational and nosql databases to determine constraints for querying robot execution logs.

[4] 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)*, pages 6331–6337, May 2017. doi: 10.1109/ICRA.2017.7989749.

[5] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[6] R. Angles. A comparison of current graph database models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 171–177, April 2012. doi: 10.1109/ICDEW.2012.31.

[7] wikipedia. Data model, 2017. URL https://en.wikipedia.org/wiki/Data_model/. [Online; accessed 10-December-2017].

[8] wikipedia. Database schema, 2017. URL https://en.wikipedia.org/wiki/Database_schema. [Online; accessed 10-December-2017].

[9] wikipedia. Immutable object, 2017. URL https://en.wikipedia.org/wiki/Immutable_object. [Online; accessed 10-December-2017].

[10] wikipedia. Master-master replication, 2017. URL https://en.wikipedia.org/wiki/Multi-master_replication. [Online; accessed 10-December-2017].

[11] wikipedia. Cap theorem, 2017. URL https://en.wikipedia.org/wiki/CAP_theorem. [Online; accessed 15-December-2017].

[12] stackchief. Cap theorem, 2017. URL https://www.stackchief.com/blog/CAP%20Theorem%20Explained. [Online; accessed 12-December-2017].

[13] docker. Docker overview, 2017. URL https://docs.docker.com/engine/docker-overview/. [Online; accessed 19-December-2017].

[14] docker. Docker architecture, 2017. URL https://docs.docker.com/engine/article-img/architecture.svg. [Online; accessed 19-December-2017].

[15] André Dietrich, Sebastian Zug, Siba Mohammad, and Jörg Kaiser. Distributed management and representation of data and context in robotic applications. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1133–1140. IEEE, 2014.

[16] 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)*, pages 15–19, Aug 2013. doi: 10.1109/PACRIM.2013.6625441.

[17] João Ricardo Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino. Choosing the right nosql database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1):18, 2015.

[18] B. G. Tudorica and C. Bucur. A comparison between several nosql databases with comments and notes. In *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pages 1–5, June 2011. doi: 10.1109/RoEduNet.2011.5993686.

[19] neo4j. Shrading, 2017. URL http://info.neo4j.com/rs/neotechnology/images/Understanding%20Neo4j%20Scalability(2).pdf. [Online; accessed 20-December-2017].

[20] fromdev. Cache based shrading, 2017. URL http://www.fromdev.com/2013/10/neo4j-cache-sharding-scale-out.html. [Online; accessed 20-December-2017].

[21] neo4j. Performance memory configuration, 2017. URL http://neo4j.com/docs/operations-manual/current/performance/memory-configuration/. [Online; accessed 20-December-2017].

[22] neo4j. Performance based on disk, 2017. URL http://neo4j.com/docs/operations-manual/current/performance/disks-ram-and-other-tips/. [Online; accessed 20-December-2017].

[23] accumulo. Accumulo architecture, 2017. URL http://accumulo.apache.org/1.6/accumulo_user_manual.html. [Online; accessed 20-December-2017].

[24] accumulo. Acuumulo data model, 2017. URL https://accumulo.apache.org/1.3/user_manual/img1.png. [Online; accessed 20-December-2017].

[25] accumulo. Accumulo replication, 2017. URL https://accumulo.apache.org/docs/2.0/administration/replication. [Online; accessed 20-December-2017].

[26] hbase. Hbase data model, 2017. URL http://hbase.apache.org/book.html#datamodel. [Online; accessed 20-December-2017].

[27] cloudera. Hbase data model, 2017. URL https://archive.cloudera.com/cdh5/cdh/5/hbase-0.98.6-cdh5.3.3/book/datamodel.html. [Online; accessed 20-December-2017].

[28] couchdb. Couchdb replication, 2017. URL http://guide.couchdb.org/draft/replication.html. [Online; accessed 21-December-2017].

[29] couchdb. Couchdb conflic management, 2017. URL http://guide.couchdb.org/draft/conflicts.html. [Online; accessed 21-December-2017].

[30] cassandra. Cassandra, 2017. URL http://cassandra.apache.org/. [Online; accessed 21-December-2017].

[31] guru99. Cassandra architecture, 2017. URL https://www.guru99.com/cassandra-architecture.html. [Online; accessed 21-December-2017].

[32] datastax. Cassandra data modeling rules, 2017. URL https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling. [Online; accessed 21-December-2017].

[33] orientdb. Orientdb, 2017. URL http://orientdb.com/docs/2.2.x/. [Online; accessed 21-December-2017].

[34] orientdb. Orientdb clusters, 2017. URL https://orientdb.com/docs/2.1/Tutorial-Clusters.html. [Online; accessed 21-December-2017].

[35] piepelinedb. Continuous views, 2017. URL http://docs.pipelinedb.com/continuous-views.html. [Online; accessed 22-December-2017].

[36] piepelinedb. Streams, 2017. URL http://docs.pipelinedb.com/streams.html. [Online; accessed 22-December-2017].

[37] piepelinedb. replication, 2017. URL http://docs.pipelinedb.com/replication.html. [Online; accessed 22-December-2017].

[38] githuub. governor, 2017. URL https://github.com/compose/governor. [Online; accessed 22-December-2017].

[39] druid. design, 2017. URL http://druid.io/docs/0.10.0/design/index.html. [Online; accessed 22-December-2017].

[40] druid. Architecture, 2017. URL http://static.druid.io/docs/druid.pdf. [Online; accessed 22-December-2017].

[41] arangodb. multi model, 2017. URL https://www.arangodb.com/why-arangodb/multi-model/. [Online; accessed 22-December-2017].

[42] arangodb. velovity pack, 2017. URL https://github.com/arangodb/velocypack. [Online; accessed 22-December-2017].

[43] arangodb. Geo indexing, 2017. URL https://docs.arangodb.com/3.3/Manual/Indexing/Geo.html. [Online; accessed 22-December-2017].

[44] mongodb. Mongodb manual, 2017. URL https://docs.mongodb.com/manual/introduction/index.html. [Online; accessed 24-December-2017].

[45] mongodb. Mongodb geojson, 2017. URL https://docs.mongodb.com/manual/reference/geojson/. [Online; accessed 24-December-2017].

[46] mongodb. Mongodb storage engine, 2017. URL https://docs.mongodb.com/manual/core/storage-engines/. [Online; accessed 24-December-2017].

[47] influxdb. Influxdb applications, 2017. URL https://www.influxdata.com/customers/. [Online; accessed 25-December-2017].

[48] influxdb. Downsampling and data retention, 2017. URL https://docs.influxdata.com/influxdb/v1.4/guides/downsampling_and_retention/. [Online; accessed 25-December-2017].

[49] influxdb. Key concepts, 2017. URL https://docs.influxdata.com/influxdb/v1.4/concepts/key_concepts/. [Online; accessed 25-December-2017].

[50] stackoverflow. Is it "safe" to use titandb?, 2017. URL https://stackoverflow.com/questions/35177947/is-it-safe-to-use-titandb. [Online; accessed 25-December-2017].

[51] datanami. Janusgraph picks up where titandb left off, 2017. URL https://www.datanami.com/2017/01/13/janusgraph-picks-titandb-left-off/. [Online; accessed 25-December-2017].

[52] stackexchange. What's the difference between mariadb and mysql?, 2017. URL https://softwareengineering.stackexchange.com/questions/120178/whats-the-difference-between-mariadb-and-mysql. [Online; accessed 25-December-2017].

[53] quora. What is the difference between mysql and mariadb?, 2017. URL https://www.quora.com/What-is-the-difference-between-MySQL-and-MariaDB. [Online; accessed 25-December-2017].

[54] geomesa. geomesa data model, 2017. URL http://www.geomesa.org/documentation/_images/accumulo-key.png. [Online; accessed 26-December-2017].

[55] siridb. Time series with siridb, 2017. URL http://siridb.net/blog/time-series-with-siridb/. [Online; accessed 26-December-2017].

[56] grakn. Graknai overview, 2017. URL https://dev.grakn.ai/overview/. [Online; accessed 26-December-2017].

[57] tutorialspoint. Mysql introduction, 2017. URL https://www.tutorialspoint.com/mysql/mysql-introduction.htm. [Online; accessed 26-December-2017].

[58] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*, pages 14–22. ACM, 2013.

[59] Tim Niemueller, Stefan Schiffer, and Gerhard Lakemeyer. Life-long learning perception using cloud database technology.

[60] neo4j. Worst practices for blobs in neo4j, 2017. URL https://neo4j.com/blog/dark-side-neo4j-worst-practices/. [Online; accessed 27-December-2017].

[61] orientdb. Binary data, 2017. URL http://orientdb.com/docs/last/Binary-Data.html. [Online; accessed 27-December-2017].

[62] couchdb. Couchdb views, 2017. URL http://guide.couchdb.org/draft/views.html. [Online; accessed 30-December-2017].