

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§

December 22, 2017

*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.

For example, autonomous mobile robots need to share their data with the other robots in the environment to solve tasks in a team. If the data is not stored in a central place, or the robots are not connected in a network, then it will be a bottle neck situation for robots to get other robots sensor data. Even though the data is stored and shared between group of robots, if the data are not organized properly with no querying capabilities, then developers have to consider writing their own solutions to filter the required data from the log files.

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 databases, which data model should be considered and, insights of best database architectures, replication strategies and conflict resolution policies.

Contents

1	Introduction	7
1.1	Motivation	7
1.2	Structure	7
2	Background	8
2.1	Data model	8
2.1.1	Type of data models	8
2.2	Database schema	8
2.3	Query Language	9
2.4	Distributed database	9
2.4.1	Availability	9
2.4.2	Scalability	9
2.4.2.1	Vertical scalability	10
2.4.2.2	Horizontal scalability	10
2.5	Immutable and Mutable objects	10
2.6	Replication	10
2.6.1	Master-Slave Architecture	11
2.6.2	Master-Master Architecture	12
2.7	CAP theorem	12
2.7.1	Consistency	12
2.7.2	Availability	13
2.7.3	Partition tolerance	13
2.7.4	Docker	13
3	Related Works	15
3.1	Database in Robotic applications	15
3.2	Performance Evaluation of databases	16
3.3	Conclusion	16
4	Problem formulation	17
5	Approach	17
6	Qualitative Analysis	18
6.1	Neo4j	18
6.1.1	Casual clustering	18
6.1.2	High available cluster	19

6.1.3	Cluster components and responsibilities	19
6.1.4	Sharding and Cache based sharding	19
6.1.5	Performance tuning in Neo4j	20
6.2	Apache Accumulo	21
6.2.1	Data model	21
6.2.2	Components of accumulo [1]	21
6.2.3	Replication	22
6.3	Apache HBase	23
6.3.1	Data model	23
6.3.2	Architecture	23
6.3.3	Advantages and disadvantages	24
6.4	Apache CouchDB	24
6.4.1	Data model	24
6.4.2	Replication [2]	25
6.4.3	Conflict management [3]	26
6.5	Apache cassandra	26
6.5.1	Architecture [4]	27
6.5.2	Replication [4]	27
6.5.3	Write and Read work flow	28
6.5.4	Rules for Cassandra data model	29
6.6	OrientDB [5]	29
6.6.1	Clusters and its uses [6]	30
6.6.2	Graphs in OrientDB	30
6.6.3	Replication vs Shrading	31
6.7	PipelineDB	31
6.7.1	Replication and high availability	32
6.8	Druid	32
6.8.1	Data model	32
6.8.2	Types of nodes [7]	33
6.9	ArangoDB	34
6.9.1	Data model	34
6.9.2	Write mechanism	34
6.9.3	Indexing	35
6.9.4	Scalability	36

List of Figures

1	Generic Replication Architecture	11
2	Master-Slave Replication Architecture	11
3	Master-Master Replication Architecture	12
4	CAP Theorem	13
5	Docker architecture	14
6	Accumulo data model	21

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 [8, 9, 10, 11], 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[12], Graph database[13], 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 ...

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.[14]

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.[15]

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.

2.4.2.1 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.

2.4.2.2 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. [16]

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.

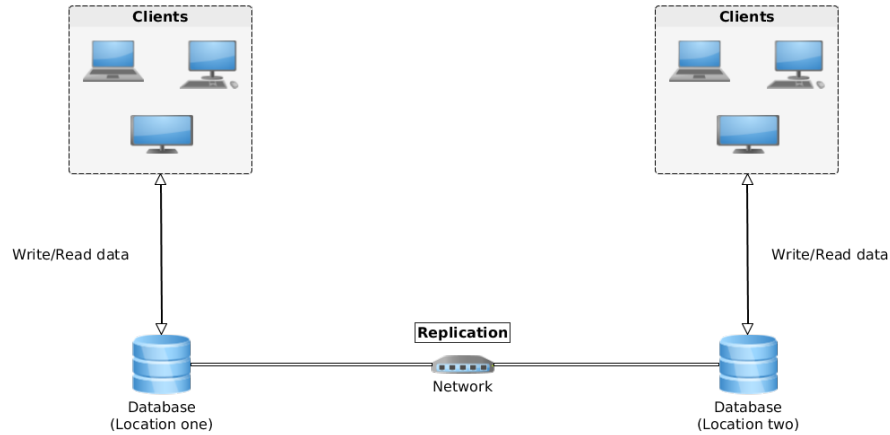


Figure 1: Generic Replication Architecture

2.6.1 Master-Slave Architecture

In master slave architecture, 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.

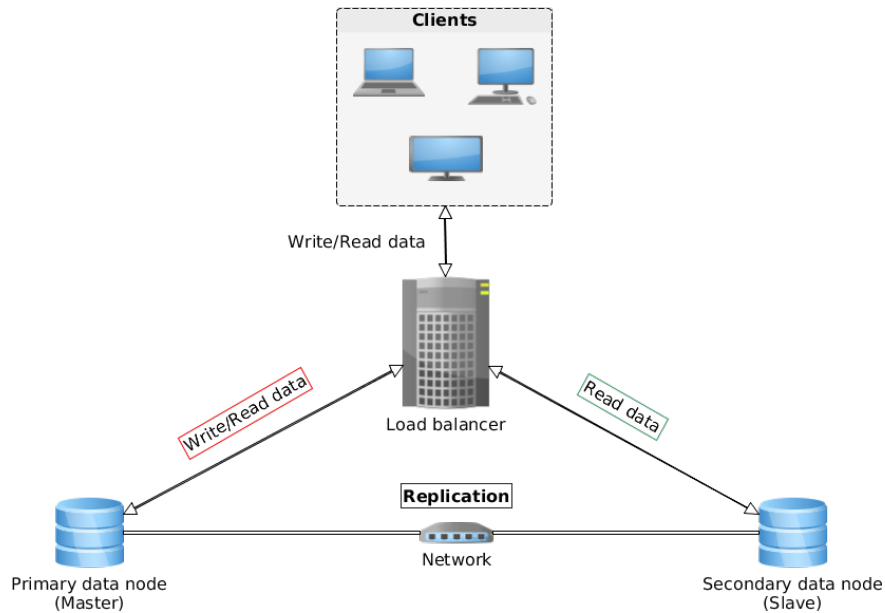


Figure 2: Master-Slave Replication Architecture

2.6.2 Master-Master Architecture

In master master architecture, 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.[17]

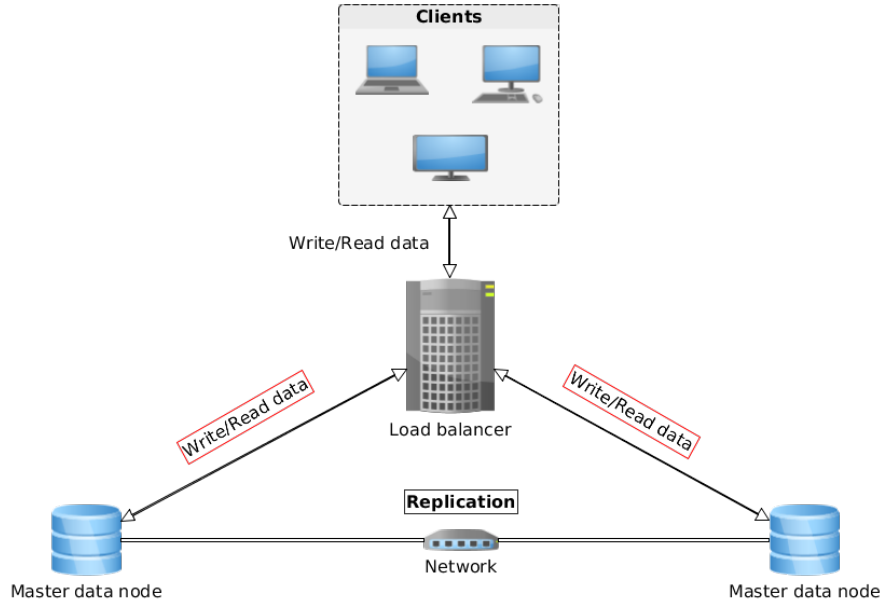


Figure 3: Master-Master Replication Architecture

2.7 CAP theorem

CAP theorem also called as Brewer's theorem named after computer scientist Eric Brewer[18], states that all distributed database systems can only provide two out of three properties which are shown in figure 4.

2.7.1 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.

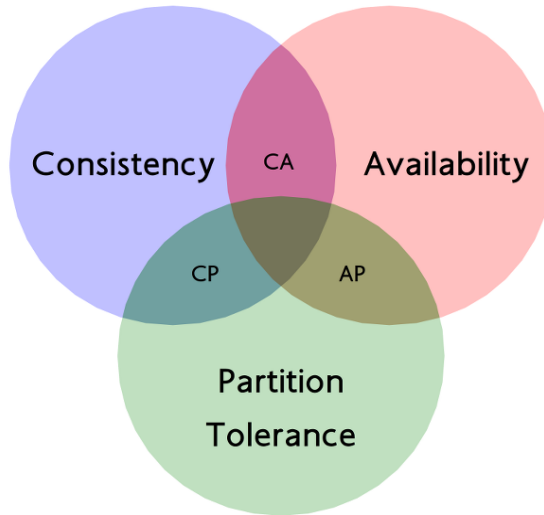


Figure 4: CAP Theorem [19]

2.7.2 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.

2.7.3 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.

2.7.4 Docker

Docker is an open source platform for developing, shipping and running applications. [20] 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.

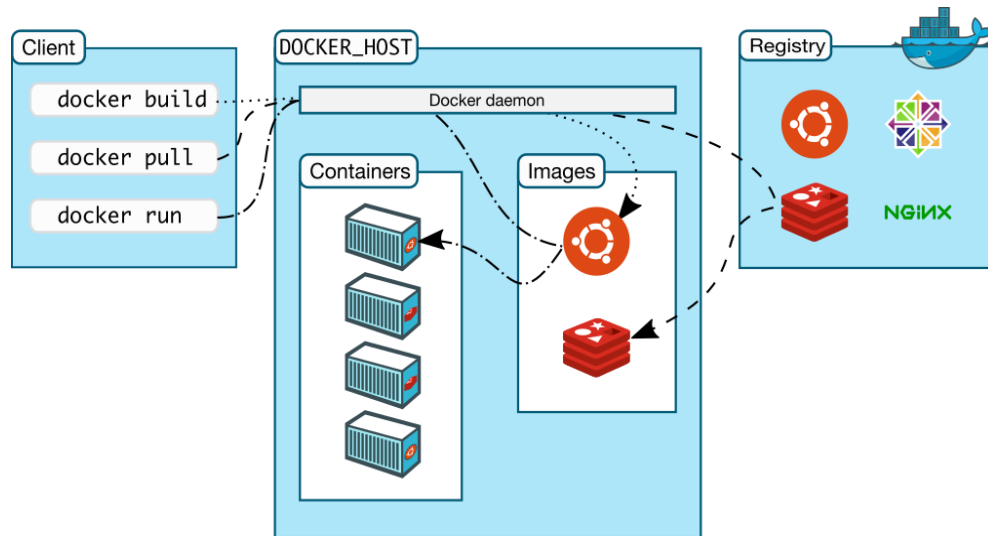


Figure 5: Docker architecture [21]

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[10, 22], comparison between SQL[23], NoSQL [24, 25], Graph databases[11], and use cases in robot applications where database could be used[8]. 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

In this paper [8], a well known NoSQL database Cassandra has been integrated 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.

In this paper [11], author 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.

This paper [10], 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

The scope of this paper [23] is to analyze 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 [11] 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. Neo4j offers two cluster solutions to ensure redundancy and performance.

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 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 [26]. 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 [27] 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.

6.1.5 Performance tuning in Neo4j

Memory based [28]

- 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 [29]

- 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.
- wIf 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 [1]. 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 [30]

6.2.2 Components of accumulo [1]

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. [1] 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. [1]

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 [31]. 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 [32]. 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 [33]. 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 framework 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 sharding 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 [2]

CouchDB is a peer-based distributed database system and it allow multi master replication architecture in the cluster, 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: [2]

```
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 [2] 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 [3]

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.

```
2-de0ea16f8621cbac506d23a0fbbde08a
```

```
2-7c971bb974251ae8541b8fe045964219
```

6.5 Apache cassandra

Cassandra 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 [34],

- 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 [4]

Cassandra architecture 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.

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 [4]

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. 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 [35]. 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 [5]

OrientDB 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 [6]

From version 2.2, OrientDB creates multiple clusters for each class to improve parallelism [6]. 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 Sharding. 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 Sharding

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, sharding 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 [36]. 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 [37]. 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 [38], 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 [39]. 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 [40].

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 [7]

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) [7].

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 [7]. 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.

References

- [1] accumulo. Accumulo architecture, 2017. URL http://accumulo.apache.org/1.6/accumulo_user_manual.html. [Online; accessed 20-December-2017].
- [2] couchdb. Couchdb replication, 2017. URL <http://guide.couchdb.org/draft/replication.html>. [Online; accessed 21-December-2017].
- [3] couchdb. Couchdb conflic management, 2017. URL <http://guide.couchdb.org/draft/conflicts.html>. [Online; accessed 21-December-2017].
- [4] guru99. Cassandra architecture, 2017. URL <https://www.guru99.com/cassandra-architecture.html>. [Online; accessed 21-December-2017].
- [5] orientdb. Orientdb, 2017. URL <http://orientdb.com/docs/2.2.x/>. [Online; accessed 21-December-2017].
- [6] orientdb. Orientdb clusters, 2017. URL <https://orientdb.com/docs/2.1/Tutorial-Clusters.html>. [Online; accessed 21-December-2017].
- [7] druid. Architecture, 2017. URL <http://static.druid.io/docs/druid.pdf>. [Online; accessed 22-December-2017].
- [8] 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.
- [9] André Dietrich, Siba Mohammad, Sebastian Zug, and Jörg Kaiser. Ros meets cassandra: Data management in smart environments with nosql.
- [10] Alexander J Fiannaca and Justin Huang. Benchmarking of relational and nosql databases to determine constraints for querying robot execution logs.
- [11] 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.
- [12] 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.

- [13] 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.
- [14] wikipedia. Data model, 2017. URL https://en.wikipedia.org/wiki/Data_model/. [Online; accessed 10-December-2017].
- [15] wikipedia. Database schema, 2017. URL https://en.wikipedia.org/wiki/Database_schema. [Online; accessed 10-December-2017].
- [16] wikipedia. Immutable object, 2017. URL https://en.wikipedia.org/wiki/Immutable_object. [Online; accessed 10-December-2017].
- [17] wikipedia. Master-master replication, 2017. URL https://en.wikipedia.org/wiki/Multi-master_replication. [Online; accessed 10-December-2017].
- [18] wikipedia. Cap theorem, 2017. URL https://en.wikipedia.org/wiki/CAP_theorem. [Online; accessed 15-December-2017].
- [19] stackchief. Cap theorem, 2017. URL <https://www.stackchief.com/blog/CAP%20Theorem%20Explained>. [Online; accessed 12-December-2017].
- [20] docker. Docker overview, 2017. URL <https://docs.docker.com/engine/docker-overview/>. [Online; accessed 19-December-2017].
- [21] docker. Docker architecture, 2017. URL <https://docs.docker.com/engine/article-img/architecture.svg>. [Online; accessed 19-December-2017].
- [22] 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.
- [23] 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.
- [24] 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.

- [25] 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.
- [26] neo4j. Shrading, 2017. URL [http://info.neo4j.com/rs/neotechnology/images/Understanding%20Neo4j%20Scalability\(2\).pdf](http://info.neo4j.com/rs/neotechnology/images/Understanding%20Neo4j%20Scalability(2).pdf). [Online; accessed 20-December-2017].
- [27] fromdev. Cache based shrading, 2017. URL <http://www.fromdev.com/2013/10/neo4j-cache-sharding-scale-out.html>. [Online; accessed 20-December-2017].
- [28] neo4j. Performance memory configuration, 2017. URL <http://neo4j.com/docs/operations-manual/current/performance/memory-configuration/>. [Online; accessed 20-December-2017].
- [29] 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].
- [30] accumulo. Accumulo data model, 2017. URL https://accumulo.apache.org/1.3/user_manual/img1.png. [Online; accessed 20-December-2017].
- [31] accumulo. Accumulo replication, 2017. URL <https://accumulo.apache.org/docs/2.0/administration/replication>. [Online; accessed 20-December-2017].
- [32] hbase. Hbase data model, 2017. URL <http://hbase.apache.org/book.html#datamodel>. [Online; accessed 20-December-2017].
- [33] 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].
- [34] cassandra. Cassandra, 2017. URL <http://cassandra.apache.org/>. [Online; accessed 21-December-2017].
- [35] datastax. Cassandra data modeling rules, 2017. URL <https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling>. [Online; accessed 21-December-2017].
- [36] piepelledb. Continuous views, 2017. URL <http://docs.pipelinedb.com/continuous-views.html>. [Online; accessed 22-December-2017].

- [37] pipelinedb. Streams, 2017. URL <http://docs.pipelinedb.com/streams.html>. [Online; accessed 22-December-2017].
- [38] pipelinedb. replication, 2017. URL <http://docs.pipelinedb.com/replication.html>. [Online; accessed 22-December-2017].
- [39] github. governor, 2017. URL <https://github.com/compose/governor>. [Online; accessed 22-December-2017].
- [40] druid. design, 2017. URL <http://druid.io/docs/0.10.0/design/index.html>. [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. velocity 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].