



An evaluation of relational and NoSQL distributed databases on a low-power cluster

Lucas Ferreira da Silva¹ · João V. F. Lima¹

Accepted: 4 March 2023 / Published online: 23 March 2023

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

The constant growth of social media, unconventional web technologies, mobile applications, and Internet of Things (IoT) devices create challenges for cloud data systems in order to support huge datasets and very high request rates. NoSQL databases, such as Cassandra and HBase, and relational SQL databases with replication, such as Citus/PostgreSQL, have been used to increase horizontal scalability and high availability of data store systems. In this paper, we evaluated three distributed databases on a low-power low-cost cluster of commodity Single-Board Computers (SBC): relational Citus/PostgreSQL and NoSQL databases Cassandra and HBase. The cluster has 15 Raspberry Pi 3 nodes with Docker Swarm orchestration tool for service deployment and ingress load balancing over SBCs. We believe that a low-cost SBC cluster can support cloud serving goals such as scale-out, elasticity, and high availability. Experimental results clearly demonstrated that there is a trade-off between performance and replication, which provides availability and partition tolerance. Besides, both properties are essential in the context of distributed systems with low-power boards. Cassandra attained better results with its consistency levels specified by the client. Both Citus and HBase enable consistency but it penalizes performance as the number of replicas increases.

Keywords Distributed databases · Raspberry Pi · Low-power · Single-Board Computers · Docker swarm · Big data · NoSQL

✉ Lucas Ferreira da Silva
lferreira@inf.ufsm.br

João V. F. Lima
jvlima@inf.ufsm.br

¹ Graduate Program in Computer Science, Universidade Federal de Santa Maria, Santa Maria, RS, Brazil

1 Introduction

Technological advances, the mobility of devices and the popularization of concepts such as IoT (Internet of Things), have caused a growth of the volume of data generated every day in proportions never seen before. The variety of digital data from the most diverse sources, combined with its large and growing quantity, represents Big Data, a term that has been gaining more and more importance both in the scientific and industry community. The survey carried out by IDC in 2018 [1] predicts that the global data volume will reach 175 ZB by the year 2025. In addition, it is estimated that 33% of this total will be data that require real-time processing.

In this scenario, with the increase in the volume of data, the demand for tools and technologies more powerful and adaptable to this new paradigm also grows. Several problems began to arise in traditional RDBMS as they were not designed to easily deal with the exponential data volume growth characteristic of Big Data. New storage technologies such as NoSQL databases appear as an option more adaptable to the Big Data scenario. Features such as capability to manage large amounts of data, easy horizontal scalability, and schema-free data model are essential requirements in cloud data systems. However, the scalability of storage and processing requires more hardware infrastructures and data centers, leading to increased investment, space and energy consumption of the infrastructure as a whole.

Several studies have proposed the adoption of commodity Single-Board Computer (SBC) clusters as a promising alternative to conventional data centers. The authors in [2] sustain that the use of low consumption devices such as SBC's can be an option to minimize the infrastructure problems of data centers. A single board can encapsulate all the resources of a functional computer and with relatively good processing power that, when interconnected as a cluster, can replicate characteristics of large data centers. In [3], the authors report that these devices can deliver a good relationship between computing power and energy consumption with compact hardware, low-power consumption, and low cost. To the knowledge of the authors, there is no related work comparing distributed databases on such devices in order to analyze the trade-off between performance and replication.

This paper evaluates three distributed databases on a low-power low-cost cluster of commodity Single-Board Computers (SBC): relational Citus/PostgreSQL and NoSQL databases Cassandra and HBase. The cluster has 15 Raspberry Pi 3 nodes with Docker Swarm orchestration tool for service deployment and ingress load balancing over SBCs. We believe that a low-cost SBC cluster can support cloud serving goals such as scale-out, elasticity, and high availability. Experimental results demonstrated that there is a trade-off between performance and replication that provides availability and partition tolerance. The two properties are essential in the context of distributed systems with low-power boards. Cassandra attained better results with its consistency levels specified by the client. Both Citus and HBase enable consistency but it penalizes performance as the number of replicas increases.

The remainder of the paper is organized as follows. Section 2 presents the related works on SBC devices and distributed databases. Section 3 describes the background on CAP theorem, Citus/PostgreSQL, Cassandra, HBase, the YCSB

benchmark, and Docker Swarm. Section 4 presents the SBC cluster platform and methodology used. Our experimental results are presented in Sect. 5. Finally, Sect. 6 and Sect. 7, respectively, present the discussion and conclude the paper.

2 Related work

Performance of NoSQL databases has been subjected of study from several research papers. In [4], the authors provide techniques and guidelines for migration from relational databases to NoSQL, and compare MySQL with HBase using Apache Phoenix as SQL layer. A detailed survey on NoSQL stores based on four design aspects is described in [5] including a CAP theorem discussion. In [6], the study compared horizontal and vertical scale of Cassandra, MongoDB, and HBase for IoT datasets where Cassandra outperformed the others on distributed scenarios. Yao and Wang [7] proposed a dynamic quorum decision for Cassandra that takes into account unavailable replicas to rescale up or down the quorum size. Antas et al. [8] used SQL and NoSQL database systems to evaluate data mining algorithms to create a COVID-19 predictive model. Other papers evaluated NoSQL databases with benchmarks such as YCSB [9–12] or NDBench [13] benchmark tools.

A number of research papers report experiments on SBC devices for distributed computing in low-power devices. [14] used the Himeno benchmark on a Raspberry Pi 1 cluster, while [15] reported HPL results over a Raspberry Pi 2 cluster. These studies concluded that a low-power cluster may achieve performance and energy efficiency despite hardware limitations. In addition, a number of works reported performance of container-based solutions for HPC applications such as Singularity [16], LXC [17], and Docker with orchestration [18–20]. In [21], the authors compared two orchestration tools and showed that Docker Swarm outperformed Kubernetes in resource usage and performance efficiency on a Raspberry Pi 3 cluster. In [22], they compared container-based Cassandra instances over virtual machines. The authors in [23] used YCSB to evaluate the Cassandra database on a Raspberry Pi 2 cluster.

The previous works show NoSQL database evaluations either on high-end servers or single database experiments. But none of the previous works focus on the impact of replication factor over low-power cluster devices. In our previous work, we presented preliminary performance results of Cassandra with a fixed replication factor over a low-power cluster of Raspberry Pi 3 boards [24]. Thus, to the knowledge of the authors, there is no related work comparing distributed databases on such devices in order to analyze the trade-off between performance and replication.

3 Background

3.1 CAP theorem

The CAP theorem, or Brewer theorem, was first proposed by Eric Brewer in 2000 [25] and formalized in 2002 [26]. The theorem introduced a fundamental

trade-off that at most two of the three desirable properties can be achieved simultaneously by any distributed data store:

- *Consistency (C)* A read receives the same response on any cluster node and the most recent write version of data or an error.
- *Availability (A)* Every request gets a response within a finite time even if the system has failures.
- *Partition tolerance (P)* Partition refers to a communication failure across distributed nodes by interruption or response time. Therefore, partition tolerance means that the distributed system works even after communication failures on the cluster.

Distributed databases can be classified in three categories considering the CAP properties: CP, AP, and CA.

CP databases provide consistency and partition tolerance at the expense of availability. When a partition occurs on any node, the system has to shut down the unavailable node until the partition is solved. The system may deny requests during the partition and may compromise the correct execution of its consensus protocol. Hence, the system cannot guarantee availability. HBase [27] and Redis [28] are examples of CP systems.

AP databases trade-off consistency for availability and partition tolerance. All nodes remain available when a partition occurs but not all of them have an up-to-date version of the data. A synchronization strategy is necessary to update nodes unavailable due to the partition. Many AP systems have eventual consistency that allows updates during partition events and delays data synchronization. Eventually all updates reach all replicas when the partition is solved [29]. The system cannot guarantee consistency on the period between new updates and data synchronization. Cassandra [30] and Amazon's Dynamo [31] are examples of AP databases.

CA databases provide consistency and availability without partition tolerance guarantee. In theory, such systems do not provide fault tolerance strategies and become unavailable until the partition is solved. Consequently, CA combination seems impossible since partitions are inevitable in distributed systems [5]. Still, a distributed CA database is possible with replication strategies across multiple nodes. Relational databases such as PostgreSQL [32] and MySQL [33] are examples of CA databases.

3.2 Citus/PostgreSQL

Citus [34] is a PostgreSQL extension that integrates data sharding and distributed query planner for horizontal scaling oblivious to the application. It maintains compatibility with the latest PostgreSQL features with full compatibility with the single-server version.

A Citus cluster has two node types: coordinator and worker. The cluster may have one or many coordinators, and zero or many workers. The coordinator stores the metadata of distributed tables and workers connect to the coordinator. Worker nodes

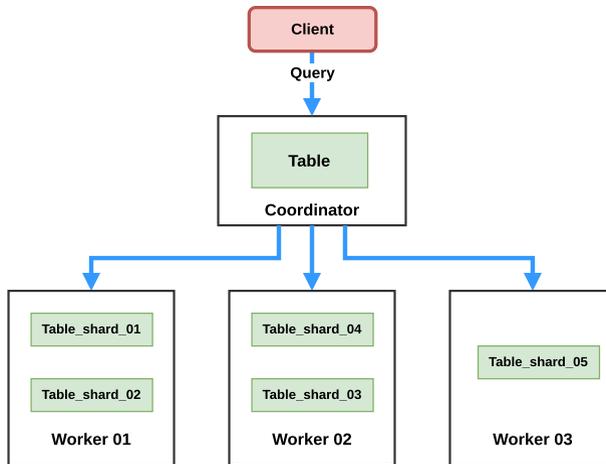


Fig. 1 Citus architecture with one coordinator and three workers. Worker may have more than one data shards

store data shards of distributed tables and execute queries. Figure 1 shows an example of a Citus cluster with one coordinator and three workers.

Citus adds to PostgreSQL the concept of distributed tables. They are hash-partitioned over a column into multiple logical shards with each shard containing a contiguous range of hash values. Each worker can have multiple logical shards placed in a round-robin fashion.

The distributed query planner handles queries to Citus tables and executes at the coordinator node. The planner produces a distributed query plan optimized to different workloads, and it consists of a set of tasks to run on workers. The distributed query executor calls the generated distributed plan. If the query targets a single worker node, it delegates the execution to the worker node for execution. Queries with shards on multiple workers generate tasks that Citus runs in parallel through multiple connections per node. If the query requires a merge step on the coordinator, the command is internally executed as a distributed `SELECT` and `COPY` to the destination table.

3.3 HBase

Apache HBase is a distributed, column-oriented, NoSQL storage for huge datasets based on Google Bigtable [35]. Its architecture is a master–slave type with the master node called HMaster and slave nodes HRegionServer. The HMaster server manages and monitors the cluster operations as well as load balancing across slaves. The HRegionServer can serve many regions that are shards of a distributed table. HBase has *autosharding* by splitting and moving regions on servers. Data storage is natively handled by HDFS [36] which allows data processing integration with Hadoop [27]. HBase uses Apache ZooKeeper as distributed coordination service to establish communication between clients and HRegionServers, to keep information

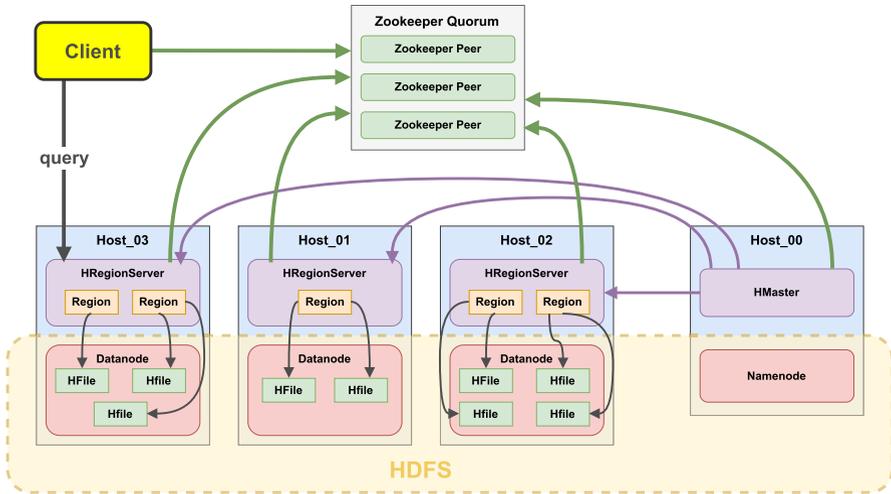


Fig. 2 HBase architecture with one HMaster and three HRegionServers

about cluster configuration and topology, and cluster monitoring. Figure 2 shows an example of a HBase cluster with one HMaster and three HRegionServers.

Each HRegionServer has the following components:

- *HFiles* persistent store files as ordered maps from keys to values.
- *WAL (Write-Ahead Log)* commit log to store data before flushed to HFile.
- *BlockCache* is a cache to keep in memory most frequently read from HFiles.
- *MemStore* memory buffer to keep written data in memory before flushed to the disk.

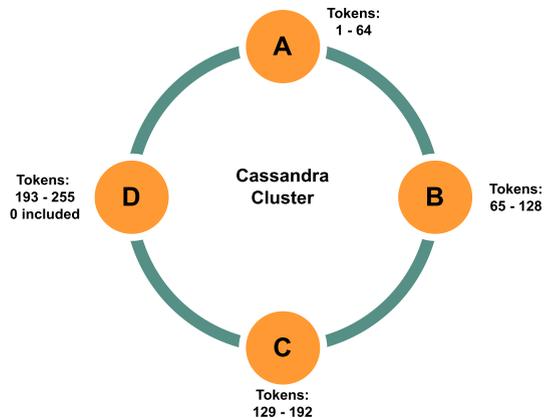
Updated data are first written to the WAL, which resides on HDFS, and then in the memory memstore. Data are flushed as a HFile to disk once data in memory exceeds a threshold value. Reading data depend on what is stored in the memstore, which has not been written to disk, and on-disk store files. WAL commit log is never used for data retrieval and only for recovery purposes.

3.4 Cassandra

Cassandra [30] is an open-source, distributed, and decentralized database that bases its distribution design on Amazon’s Dynamo [31] and its data model on Google’s Bigtable [35].

A Cassandra cluster has horizontal scalability which means that it can scale up and scale down seamlessly. Its eventual consistency is tunable by setting the consistency level and replication factor. The replication factor configures the number of nodes that updates will be propagated. The consistency level sets how many replicas must acknowledge a write or respond to a read operation in order to be considered successful. The clients must specify this level at each operation.

Fig. 3 Token distribution in a Cassandra cluster



Communication between the nodes of the Cassandra cluster is carried out through the Gossip protocol. It uses peer-to-peer communication that allows the nodes to exchange information periodically about their state and the state of the other nodes within the cluster, thus ensuring that possible failures are identified [37].

Data partitioning between the nodes of the cluster is performed dynamically through a consistent hash algorithm, which makes each node responsible for a portion of the data, according to the interval established by the token generated for the node. If there is a new node joining the cluster, the token values will be automatically adjusted so that each node has a portion of data proportional to the other nodes. Figure 3 shows an example of the distribution of a range from 0 to 255 tokens in a 4-node Cassandra cluster.

3.5 YCSB benchmark suite

The Yahoo Cloud Serving Benchmark (YCSB) [9] is a cloud database benchmark to analyze several workloads. YCSB is an open-source project, and it has support to many distributed NoSQL and relational databases such as Cassandra and PostgreSQL, respectively.

It includes a set of pre-defined workloads in order to evaluate different aspects of systems' performance. In this paper, we consider four of six basic workloads available:

- Workload A—heavy update with 50% read and 50% update;
- Workload B—heavy read with 95% read and 5% update;
- Workload C—100% reads;
- Workload D—read latest update with 95% read and 5% insert.

3.6 Docker swarm

Docker is an open-source platform that allows the creation of customized images used as a definition base for containers. According to [38], a container is a standard

unit of software that packages the code and all of its dependencies allowing the application to run quickly and reliably regardless of the runtime environment. Unlike virtual machines, which are based on the hardware abstraction of the host machine through a hypervisor, container virtualization is performed at the operating system level. Therefore, it allows the resources of the physical machine to be partitioned, creating isolated user spaces running on the same operating system kernel as the host machine.

Docker Swarm is a container orchestration tool incorporated natively on the Docker platform through the SwarmKit library. Docker Swarm enables the creation and management of a cluster composed of machines running the Docker engine, which interact with each other through the Docker API [39]. Each of these machines, or Docker nodes, can play the role of manager, worker, or perform both functions. The managing node is responsible for performing the cluster orchestration and management functions, being responsible for the distribution of the work units (called tasks) among the other nodes in the cluster, and can also execute tasks like a worker node. On the other hand, worker nodes do not have responsibilities linked to the maintenance of the cluster, and their function is related to the execution of tasks; however, worker nodes can also be promoted to manager nodes [40].

The definition of the tasks to be performed on the Swarm cluster nodes is called Service. When structuring a service for a Swarm cluster, details of each of the tasks to be performed are specified, being possible to define the Docker image to be used, which commands will be executed, network configurations and volumes definition, among other configurations.

4 Methodology

Maintenance and configuration of distributed systems presents some challenges in managing and deploying an application. Given the heterogeneous hardware characteristics of devices, there may be different libraries and software dependencies for each architecture, in addition to the individual monitoring and maintenance of each cluster node.

In order to minimize these problems, the Docker Swarm was used to manage and orchestrate the resources of the low-power devices, as it allows the abstraction of some management tasks of the cluster nodes, besides facilitating the installation, configuration, and dimensioning of resources. Both YCSB and Databases nodes are running inside Docker containers distributed among the 15 SBCs in the cluster plus an additional node used as monitoring server.

4.1 Hardware and software specifications

Our low-power SBC cluster consists of 15 Raspberry Pi 3 B devices interconnected via Ethernet on a Gigabit speed switch. Each board has a USB 2.0 stick of 32 GB attached exclusively for database storage. An additional Raspberry Pi 3 B board was

Table 1 Raspberry Pi 3 B specifications

SoC	Broadcom BCM2837
CPU	Cortex-A53 64-bit 1.2 GHz quad-core
Architecture	ARMv7
RAM size	1 GB
Ethernet speed	300 Mbps

used exclusively as a monitoring server. Table 1 summarizes the Raspberry Pi hardware configurations.

The operating system was HyprIoT OS¹ (version 1.11), a system based on the Debian Linux distribution, but with optimizations for running the Docker platform on devices with ARM processors. The running Docker Engine version was the 19.03. The database versions were 3.11.4 for Cassandra, 2.2.6 for HBase, and 13.2 for PostgreSQL with Citus 10.1.2.

4.2 Cassandra configuration

The testbed topology shown in Fig. 4 consists of 15 SBCs running a single Cassandra container each, with 1 seed node and the other 14 as common Cassandra nodes. We had to adapt some Cassandra configurations due to the hardware limitations of the Raspberry Pi SBCs. It was necessary to set JVM heap limitation values according to the amount of memory available on the devices in order to avoid out of memory exceptions. The maximum heap size was set to 430 MB and the size of the heap new generation to 142 MB. Each container has a volume that maps the Cassandra storage directory (`/var/lib/cassandra`) to the USB stick on the host SBC.

Because they run as a stack on the Docker Swarm, all Cassandra containers are interconnected through an overlay network called `cassandra-net`. It allows the internal network of the Swarm stack to be isolated from the external host devices network. We chose to run YCSB from a container connected to the `cassandra-net` overlay network to avoid the need to add another proxy service to the stack.

Another problem with multiple nodes and hardware limitations is the write and read timeout exceptions. They are usually triggered when one or more nodes are unable to respond to the request within an established time limit. Therefore, to avoid this problem, the solution adopted was to define a timeout large enough that all nodes in the cluster can handle it. In this case, the value of 60,000 milliseconds was established for the writing and reading timeouts.

The replication strategy of our experiments was *SimpleStrategy* since there is one single cluster as a Cassandra datacenter. The replication factor is configured at table creation. In addition, we configured YCSB consistency levels to `ONE` that means that a write must be written to the commit log and memtable of at least one replica node.

¹ HyprIoTOS, <https://blog.hypriot.com/>.

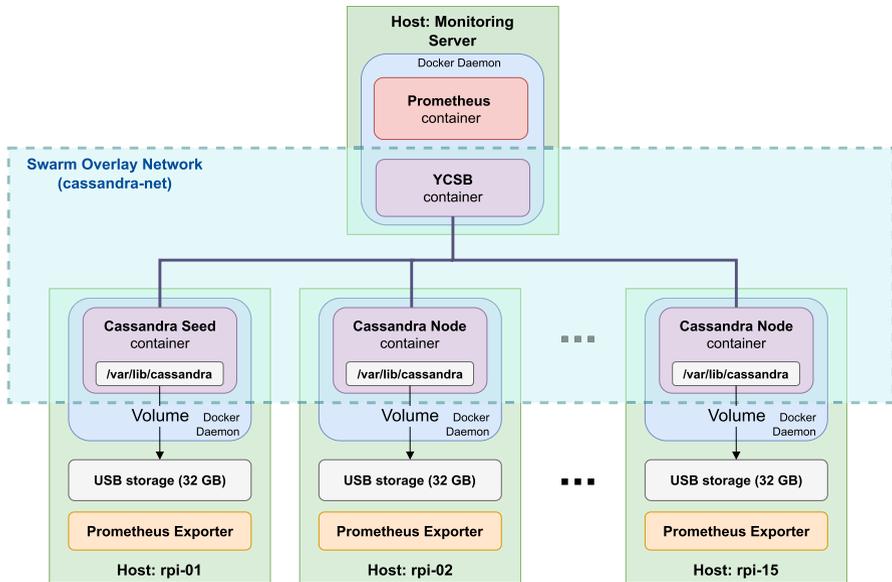


Fig. 4 Cassandra tested topology of our low-power low-cost SBC cluster

4.3 Citus configuration

The testbed topology shown in Fig. 5 consists of 15 SBCs running a single Citus container each, with 1 coordinator node and the other 14 as worker nodes. In a Citus cluster, all YCSB queries run on the coordinator node that is responsible to generate the query plan. Each Citus container has a volume that maps the PostgreSQL storage directory (`/var/postgresql/data`) to the USB stick on the host SBC.

The Citus tested topology has an additional Docker container on the coordinator node to execute the *membership-manager* service. This service monitors the network ingress and egress of cluster nodes since PostgreSQL does not have a native mechanism for horizontal scaling. The *membership-manager* of our experiments was modified in order to support the active monitoring of a Docker Swarm cluster. It actively checks the status of Swarm containers over an Unix Socket file mapped to `/var/run/docker.sock` on the host node. The replication factor in Citus is configured at table creation.

4.4 HBase configuration

The testbed topology shown in Fig. 6 consists of 15 SBCs running a single HMaster container, which does not storage data, and the other 14 as HRegionServer nodes. Besides, the cluster has 3 nodes running Zookeeper for coordination

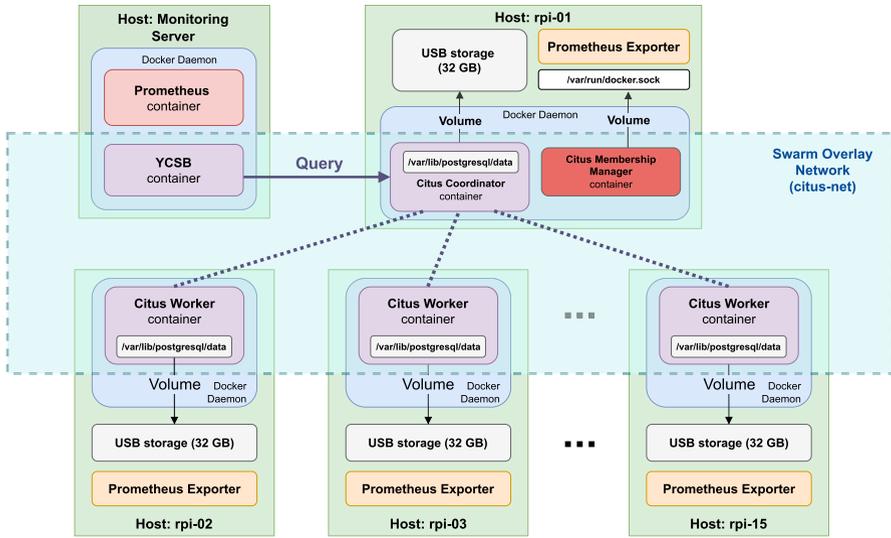


Fig. 5 Citus tested topology of our low-power low-cost SBC cluster

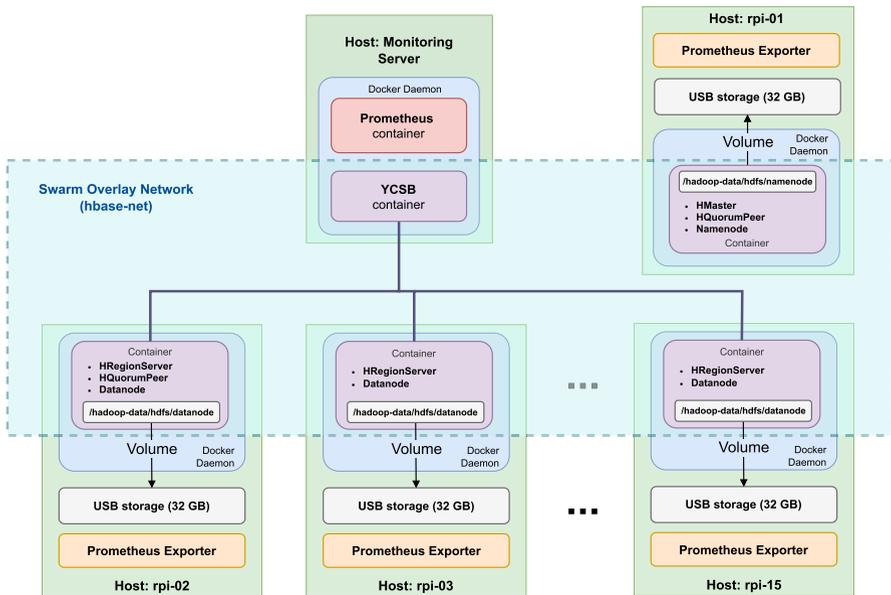


Fig. 6 HBase tested topology of our low-power low-cost SBC cluster

service. Each HBase container has a volume that maps the HRegionServer storage directory (/hadoop-data/hdfs/datanode) to the USB stick on the host SBC. The replication factor in HBase is configured at table creation.

4.5 Monitoring

We used the Prometheus² monitoring and alerting toolkit to monitor the use of the hardware and system resources of each SBC in the cluster. The data scraping is done through the Exporters who capture the metrics and expose them so that the Prometheus server can collect and store them in a time-series database. The Prometheus server runs in a Docker container on the Monitoring Server host, and the Exporters are executed in each of the SBCs as a background process directly on the host system. Unlike other containers, the Prometheus server does not run as a Docker Swarm service and is not part of the overlay network, allowing access to Exporters by the Prometheus server for data collection.

A Node Exporter³ was installed in each SBC, which collects data from the metrics every 5s and exposes them to be consumed by Prometheus Server via HTTP queries to the device. Although the Node Exporter obtains information on several system metrics, we only focus on monitoring RAM consumption in this work.

4.6 Methodology

All YCSB executions were composed of two steps: The first generates records to load into the database; the second executes a number of operations. The initialization phase loads data records of 10 columns where each column has 100 bytes or 1 KB per record. Data are randomly generated, and each record has a unique primary key. The data volume load was fixed on 1 million records or 1 GB in disk space. We ran a load phase for each workload type and data size followed by 30 repetitions of its workload operations. Each workload execution performs 10,000 operations on the database. The replication factor was tested from one to three nodes at each workload execution.

The analysis of variance was computed on statistically significant differences based on the appropriate F - tests. Each result is a mean of 30 executions, and the 95% confidence interval is represented by a vertical line around the mean values. The mean differences were compared using the Tukey's honest significantly difference (HSD) test ($P < 0.05$) [41] in order to indicate significant difference groups in each workload represented by lowercase letters in each workload. The results were subjected to analysis of variance (ANOVA) $P < 0.05$. Results with the same lowercase letter mean that there is no evidence of a difference between them, and results with more than one letter show an overlap of samples within the confidence interval.

² Prometheus, <https://prometheus.io/>.

³ Node Exporter, https://github.com/prometheus/node_exporter.

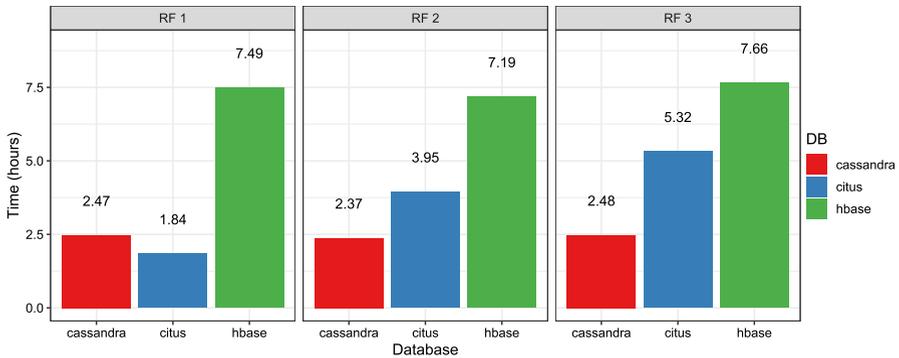


Fig. 7 YCSB load time in hours grouped by replication factor (RF)

5 Experimental results

The goal of our experiments is to evaluate three distributed databases over a cluster of low-power Raspberry Pi boards. Our objectives are:

- Evaluate the three databases over our cluster in terms of YCSB throughput operations and read/write latency;
- Analyze the impact of different workloads;
- Identify strengths and weaknesses of SQL and NoSQL distributed databases in low-power cluster board.

Figure 7 shows the load time of records in hours before workload executions. Citus had the lowest load time with 1.8 h, followed by Cassandra and HBase. In addition, replication factor increased Citus load time in an almost directly proportional ratio. Cassandra and HBase did not show significant impact related to the replication factor.

Figure 8 shows the throughput results in our experiments. Cassandra outperformed HBase for workloads A and D on all replication factor values. HBase showed better throughput for workload C and replication factors 2 and 3. A replication factor greater than one reduced throughput of Cassandra, but it did not have significant impact for HBase with one letter on B and D, and two letters on A and C. Citus had the lowest throughput results on all four workload, and replication factor had no significant difference except for workload A with replication factor 1.

Figure 9 shows the read latency results in our experiments. Cassandra and HBase did not show significant differences on most workloads and replication factors with the exception of workload D. Replication factor did not have a significant difference on all workloads for both databases. Citus had the highest read latency results on all four workloads, and replication factor did increase latency significantly.

Figure 10 shows the write/update latency results in our experiments. Citus showed high latency results on workloads A and B which correspond to update operations. Nonetheless, Citus showed an execution error on the three replication

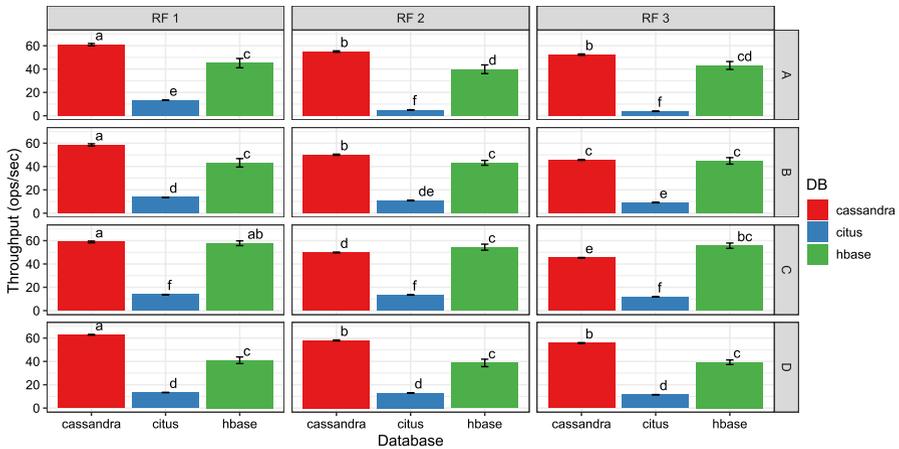


Fig. 8 Throughput results of YCSB measured with Cassandra, Citus, and HBase on our 15 Raspberry Pi 3 cluster. We measured throughput over three replication factors (RF) on four workloads (uppercase letters). Higher is better for throughput. Data represent the mean and the 95% confidence interval. Different lowercase letters indicate significant differences among databases and replication factor numbers in the same workload by the Tukey HSD test ($p < 0.05$)

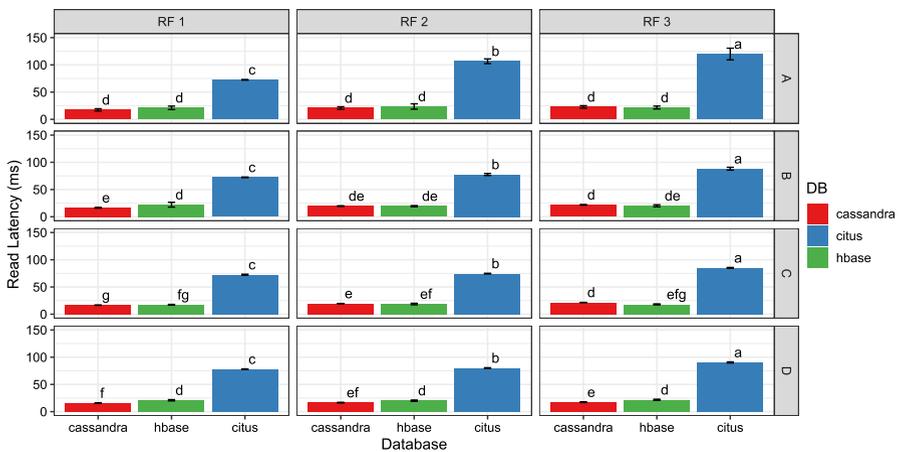


Fig. 9 Read latency results of YCSB measured with Cassandra, Citus, and HBase on our 15 Raspberry Pi 3 cluster. Lower is better for latency. Statistics as shown in Fig. 8

factor values of D workload. This error was due to a failure to read results after the batched insert operations. Replication factor did not have significant difference for Cassandra and HBase on A and B workloads. Cassandra had the lowest write latency results on workloads A and B.

Table 2 summarizes minimum and maximum median values on each workload. Replication factor increased read latency on all workloads of our experiments since Cassandra had lower latencies with one replica. However, write latency decreased

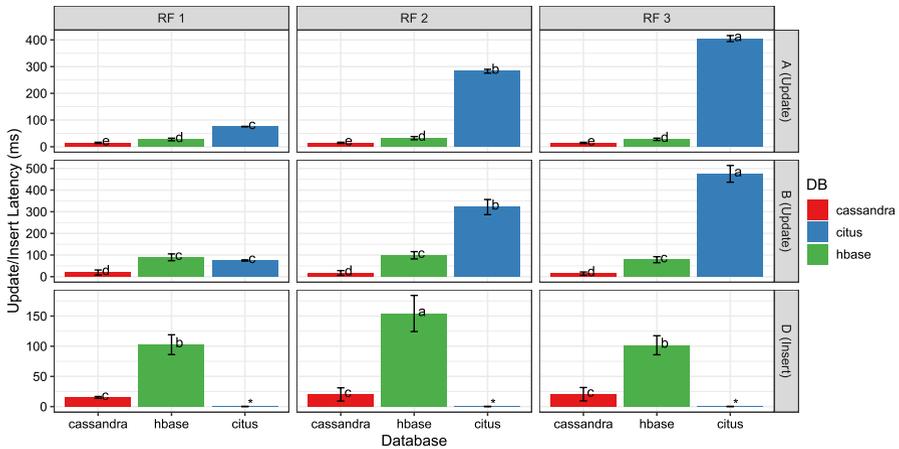


Fig. 10 Write/update latency results of YCSB measured with Cassandra, Citus, and HBase on our 15 Raspberry Pi 3 cluster. Lower is better for latency. Statistics as shown in Fig. 8

Table 2 Summary of read, update (A and B), and insert (D) latency operations per workload in milliseconds

Workload	Read latency (ms)		Update/write latency (ms)	
	Min	Max	Min	Max
A	17.2 (cassandra/1)	119.9 (citus/3)	14.4 (cassandra/3)	404.8 (citus/3)
B	16.4 (cassandra/1)	88.2 (citus/3)	13.9 (cassandra/3)	475.0 (citus/3)
C	16.5 (cassandra/1)	85.1 (citus/3)	–	–
D	15.4 (cassandra/1)	90.3 (citus/3)	0.22 (citus/2)	154.2 (hbase/2)

Values are the median followed by the database and replication factor measured

as the number of replicas was greater than one on all four workloads. Cassandra was better on all other workloads for read (one replica) and update (three replicas) latencies.

6 Discussion

Our experimental results showed that Cassandra attained better results on write and update workloads, while HBase performed better on heavy read workload C with more than 2 replicas. It seems that eventual consistency and integrated storage structure of Cassandra are the main reasons for better results. These findings are expected for a database classified as AP by the CAP theorem. Besides, HBase results suggest that its

write overhead for consistency affected performance as expected by a CP database. The first step on a HBase data update is a new entry on the WAL commit log that resides on HDFS and is replicated. Consequently, each data update implies disk operations on each data replica.

Experimental results with Citus reported high latency values for read and write workloads, but a lower load time for one replica on Fig. 7. Its load time results on one replica are due to batched operations with COPY instead of insertion operations for each data entry. However, the replication factor impacted Citus significantly on all results. One explanation could be that Citus replicates each operation on distributed tables by the number of shards causing additional costs. Despite the additional overhead costs, Citus enables a distributed database with CA properties through replication of logical shards and distributed queries.

Throughput results had a significant impact due to Raspberry Pi 3 hardware limitations. We believe that one of the main limitations is RAM memory since it was the only saturated resource in our Prometheus monitoring observations. Related works also reported Cassandra experiments under 2 GB for virtualized environments [10, 12, 23]. Other authors attained significant throughput results for on smaller datasets [6, 10–12, 42] or low-power devices [23]. On the other hand, we had latency results comparable to related papers running Cassandra on virtualized and native high-end platforms [43–45].

Furthermore, it is clear that there is a trade-off between performance and replication. Database applications have to consider a twofold decision concerning application requirements and hardware specifications. CP databases suit applications that demand consistency and ACID operations, while AP databases provide availability for high data volume applications. AP is essential in the context of a low-power board cluster due to its low mean time between failures (MTBF) compared to high-end servers. Therefore, Cassandra can be an option for production systems with low-power boards using its tunable consistency levels by the client. Citus and HBase enable consistency but it penalizes performance with the increase in replicas.

Our experimental results provide evidence that a SBC cluster can support cloud serving goals [9] such as scale-out supporting 15 boards and balancing load across them. Moreover, it appears that our cluster architecture has elasticity and high availability goals due to the Docker Swarm orchestration. New nodes can be added or removed at runtime on the cluster for elasticity, while high availability comes from distributed databases and ingress load balancing of Docker Swarm.

We estimate that the total power usage on our cluster may be 108 kJ per hour at maximum load assuming that 400 mA and a Raspberry Pi 3 B consumption at 5 V. A typical high-end server may consume 720 kJ per hour at 200 W without considering air cooling. These metrics support our arguments for the usage of distributed databases on low-power boards due to low-energy consumption and low budget cost.

7 Conclusion

In this paper, we evaluated three distributed databases on a low-power Raspberry Pi 3 cluster of 15 nodes. The cluster had the Docker Swarm orchestration tool for service deployment and ingress load balancing between SBCs. We believe that a

low-cost SBC cluster can support cloud serving goals such as scale-out, elasticity, and high availability. Experimental results clearly demonstrated that there is a trade-off between performance and replication, which provides availability and partition tolerance. Besides, both properties are essential in the context of distributed systems with low-power boards.

Cassandra attained better results with its tunable consistency levels by the client. Besides, read and write latencies were comparable to results from other work on high-end or virtualized platforms. Both Citus and HBase enable strong consistency that penalizes performance as the number of replicas increases.

Future works include a comparison of consistency levels with Cassandra, an in-depth energy consumption analysis, and the impact of node failures on low-power devices.

Acknowledgements This work was fully supported by CNPq. This research received funding from CAPES (Brazil)—Finance Code 001, UFSM/FATEC through Project Number 041250-9.07.0025 (100548), and by the project "GREEN-CLOUD" (#16/2551-0000 488-9) from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

Authors' contributions L.F.S. implemented the configuration files, configured the experimental testbed, ran the experimental results, prepared all figures, and wrote a draft version in portuguese. J.V.F.L. contributed to the statistical analysis of experimental results and wrote the main manuscript text. All authors reviewed the manuscript.

Funding This research received funding from: Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq)—Brazil Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code 001. FAPERGS and CNPq Brazil, program PRONEX 12/2014, by the project "GREEN-CLOUD" (#16/2551-0000 488-9). UFSM/FATEC through Project Number 041250-9.07.0025 (100548).

Availability of data and materials The referenced data and materials are available at: Scripts to run our experiments at: https://bitbucket.org/jvlima/supe2022-ycsb_benchmark_docker Modified YCSB source code at: <https://bitbucket.org/jvlima/supe2022-ycsb> Docker files for Citus at: <https://bitbucket.org/jvlima/supe2022-postgres-citus-pi> Docker files for HBase at: <https://bitbucket.org/jvlima/supe2022-hbaseswarmrpi> Docker files for Cassandra at: <https://bitbucket.org/jvlima/supe2022-cassandra-armhf> All experimental results at: <https://bitbucket.org/jvlima/supe2022-ycsb-prom-metrics>.

Declarations

Conflict of interest The authors declare that they have no competing interests.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

References

1. David Reinsel JR, Gantz J (2018) The digitization of the world from edge to core. <https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf>
2. Johnston SJ, Basford PJ, Perkins CS, Herry H, Tso FP, Pezaros D, Mullins RD, Yoneki E, Cox SJ, Singer J (2018) Commodity single board computer clusters and their applications. *Future Gener Comput Syst* 89:201–212. <https://doi.org/10.1016/j.future.2018.06.048>

3. Wolf W, Jerraya AA, Martin G (2008) Multiprocessor system-on-chip (MPSoC) technology. *Trans Comput Aided Des Integr Circuits Syst* 27(10):1701–1713. <https://doi.org/10.1109/TCAD.2008.923415>
4. Kim H-J, Ko E-J, Jeon Y-H, Lee K-H (2020) Techniques and guidelines for effective migration from RDBMS to NoSQL. *J Supercomput* 76(10):7936–7950. <https://doi.org/10.1007/s11227-018-2361-2>
5. Davoudian A, Chen L, Liu M (2018) A survey on NoSQL stores. *ACM Comput Surv*. <https://doi.org/10.1145/3158661>
6. Hendawi A, Gupta J, Liu J, Teredesai A, Ramakrishnan N, Shah M, El-Sappagh S, Kwak K-S, Ali M (2019) Benchmarking large-scale data management for Internet of Things. *J Supercomput* 75(12):8207–8230. <https://doi.org/10.1007/s11227-019-02984-6>
7. Yao X, Wang C-L (2020) Probabilistic consistency guarantee in partial quorum-based data store. *IEEE Trans Parallel Distrib Syst* 31(8):1815–1827. <https://doi.org/10.1109/TPDS.2020.2973619>
8. Antas J, Rocha Silva R, Bernardino J (2022) Assessment of SQL and NoSQL systems to store and mine Covid-19 data. *Computers*. <https://doi.org/10.3390/computers11020029>
9. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. ACM, New York, pp 143–154. <https://doi.org/10.1145/1807128.1807152>
10. Abramova V, Bernardino J, Furtado P (2014) Evaluating cassandra scalability with YCSB. In: Decker H, Lhotská L, Link S, Spies M, Wagner RR (eds) *Database and expert systems applications*. Springer, Cham, pp 199–207
11. Swaminathan SN, Elmasri R (2016) Quantitative analysis of scalable NoSQL databases. In: *2016 IEEE International Congress on Big Data (BigData Congress)*, pp 323–326. <https://doi.org/10.1109/BigDataCongress.2016.49>
12. Abramova V, Bernardino J (2013) NoSQL databases: MongoDB vs cassandra. In: *Proceedings of the International C* Conference on Computer Science and Software Engineering*. C3S2E '13. ACM, New York, pp 14–22. <https://doi.org/10.1145/2494444.2494447>
13. Papanagioutou I, Chella V (2018) NDBench: benchmarking microservices at scale
14. Ashari A, Riassetiawan M (2015) High performance computing on cluster and multicore architecture. *Telkomnika (Telecommun Comput Electron Control)* 13(4):1408–1413
15. Priyambodo TK, Lisan AW, Riassetiawan M (2018) Inexpensive green mini supercomputer based on single board computer cluster. *J Telecommun Electron Comput Eng* 10(1–6):141–145
16. Sande Veiga V, Simon M, Azab A, Fernandez C, Muscianisi G, Fiameni G, Marocchi S (2019) Evaluation and benchmarking of singularity MPI containers on EU research e-infrastructure. In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp 1–10. <https://doi.org/10.1109/CANOPIE-HPC49598.2019.00006>
17. Beserra D, Moreno ED, Endo PT, Barreto J, Sadok D, Fernandes S (2015) Performance analysis of LXC for HPC environments. In: *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pp 358–363. <https://doi.org/10.1109/CISIS.2015.53>
18. Zhang J, Lu X, Panda DK (2016) High performance MPI library for container-based HPC cloud on InfiniBand clusters. In: *2016 45th International Conference on Parallel Processing (ICPP)*, pp 268–277. <https://doi.org/10.1109/ICPP.2016.38>
19. Steffanel LA, Charão AS, da Silva Alves B (2019) A containerized tool to deploy scientific applications over SoC-based systems: the case of meteorological forecasting with WRF. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress, Heraklion, Crete, pp 561–568. <https://doi.org/10.5220/0007799705610568>. INSTICC
20. Steffanel LA, Charão AS, Alves B, de Araujo LR, da Silva LF (2020) MPI to go: container clusters for MPI applications. In: Ferguson D, Méndez Muñoz V, Pahl C, Helfert M (eds) *Cloud computing and service science*. Springer, Cham, pp 199–222
21. Fayos-Jordan R, Felici-Castell S, Segura-Garcia J, Pastor-Aparicio A, Lopez-Ballester J (2019) Elastic computing in the fog on Internet of Things to improve the performance of low cost nodes. *Electronics* 8(12):1489
22. Shirinbab S, Lundberg L, Casalicchio E (2020) Performance evaluation of containers and virtual machines when running cassandra workload concurrently. *Concurr Comput Pract Exp* 32(17):5693. <https://doi.org/10.1002/cpe.5693>
23. Richardson DP, Lin AC, Pecarina JM (2017) Hosting distributed databases on internet of things-scale devices. In: *2017 IEEE Conference on Dependable and Secure Computing*, pp 352–357

24. Da Silva LF, Lima JVF (2021) An evaluation of cassandra nosql database on a low-power cluster. In: 2021 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pp 9–14. <https://doi.org/10.1109/SBAC-PADW53941.2021.00012>
25. Brewer EA (2000) Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. PODC '00. ACM, New York, p 7. <https://doi.org/10.1145/343477.343502>
26. Gilbert S, Lynch N (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33(2):51–59. <https://doi.org/10.1145/564585.564601>
27. Hadoop A (2022) Apache Hadoop. <http://hadoop.apache.org/>
28. Redis Ltd (2022) Redis. <https://redis.io/>
29. Brewer E (2012) Cap twelve years later: how the rules have changed. *Computer* 45(2):23–29. <https://doi.org/10.1109/MC.2012.37>
30. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. *SIGOPS Oper Syst Rev* 44(2):35–40. <https://doi.org/10.1145/1773912.1773922>
31. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vossshall P, Vogels W (2007) Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles. SOSP '07. ACM, New York, pp 205–220. <https://doi.org/10.1145/1294261.1294281>
32. Stonebraker M, Rowe LA, Hirohama M (1990) The implementation of postgres. *IEEE Trans Knowl Data Eng* 2(1):125–142. <https://doi.org/10.1109/69.50912>
33. Oracle Corporation: MySQL (2022). <https://www.mysql.com/>
34. Cubukcu U, Erdogan O, Pathak S, Sannakkayala S, Slot M (2021) Citus: distributed postgresql for data-intensive applications. In: Proceedings of the 2021 International Conference on Management of Data. SIGMOD '21. ACM, New York, pp 2490–2502. <https://doi.org/10.1145/3448016.3457551>
35. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: a distributed storage system for structured data. *ACM Trans Comput Syst*. <https://doi.org/10.1145/1365815.1365816>
36. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: MSST '10. IEEE Computer Society, pp 1–10 (2010). <https://doi.org/10.1109/MSST.2010.5496972>
37. Cassandra A (2020) Dynamo. <https://cassandra.apache.org/doc/latest/architecture>
38. Docker: what is a Container? (2020). <https://www.docker.com/resources/what-container>
39. Docker: Swarm mode overview (2020). <https://docs.docker.com/engine/swarm/>
40. Docker: Swarm mode key concepts (2020). <https://docs.docker.com/engine/swarm/key-concepts/>
41. Witte RS, Witte JS (2017) *Statistics*, 11th edn. Wiley, Hoboken
42. Tang E, Fan Y (2016) Performance comparison between five NoSQL databases. In: 2016 7th International Conference on Cloud Computing and Big Data (CCBD), pp 105–109. <https://doi.org/10.1109/CCBD.2016.030>
43. Datastax (2015) Benchmarking top NoSQL databases. Technical report, End Point Corporation
44. Kuhlenskamp J, Klems M, Röss O (2014) Benchmarking scalability and elasticity of distributed database systems. *Proc VLDB Endow* 7(12):1219–1230. <https://doi.org/10.14778/2732977.2732995>
45. Huang X, Wang J, Zhong Y, Song S, Yu PS (2015) Optimizing data partition for scaling out NoSQL cluster. *Concurr Comput Pract Exp* 27(18):5793–5809. <https://doi.org/10.1002/cpe.3643>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.