Benchmarking Apache Cassandra in Polyglot Persistence

Literature Review

Bradley Culligan
cllbra005@myuct.ac.za
University of Cape Town
Cape Town, South Africa

ABSTRACT

Apache Cassandra is the market-leading column based NoSQL database model useful for high write throughput operations and analytical research on the stored data. Cassandra is one such database model that can be applied in designing polyglot persistence — the term meaning that multiple data stores should be utilised to take advantage of the way the data intends to be used. In short, use the right tool for the right job. We review the previous research on this paradigm and find that most of the research is focused on polyglot persistence design in general, or on categorising the various NoSQL models for specific use cases. We conclude that additional research is required to focus on investigating specific data models (such as Cassandra) and how they will perform in polyglot persistence.

KEYWORDS

Database, SQL, NoSQL, Cassandra, Column, Polyglot Persistence, Big Data, Performance

1 INTRODUCTION

The big data era has brought challenges to the traditional relational database model. Big data was originally characterised by three traits [22]:

Volume The quantity of data.
Velocity The rate at which data is created.
Variety The structured, semi-structured and unstructured notion of data.

This definition has been extended to include more characteristics such as *veracity* and *variability* which relates to the truth of the data (possible errors) and the flow of the data respectively. For the purposes of this review, the problems in the relational model were made clear by the original three traits (volume, velocity and variety) so we shall limit our discussion to these characteristics.

Web 2.0, the Internet of Things (IoT) and digitisation has led to vast quantities of data — too vast for the relational model to handle [20]. Relational databases have proved to not scale well to this volume of data. The velocity of data also requires that the relational model is able to perform fast transcations. However, the poor scalability and complex concurrency techniques have caused a bottleneck on performance. Further, the relational model requires strict schema which is not well-suited towards the variety of data in big data. The NoSQL model has been developed to better suit big data and directly appeals these issues.

The NoSQL model covers a broad range of database models, each with their own strengths and weaknesses. Since their inception, time has allowed them to mature and become solutions used in the modern data focuesd world. However, while NoSQL may offer solutions to certain problems, they are not able solve all the problems alone. Polyglot persistence has been deemed the solution, allowing various data models to be used such that the right tool can be used for the right job. This creates a database system that is able to handle a larger variety of data problems, and better optimise those solutions by using a data model which will have better performance [21]. Polyglot persistence is already implemented at some of the top technical companies, such as Amazon and Google, due to the types of data problems that they are required to deal with. In general, these companies led the pursuit of NoSQL as database solutions and we can now look to them in understanding that polyglot persistence needs to be researched in order to better prepare ourselves for future concerns in our database models.

Orchestrating the various components of polyglot persistence can be a difficult task and can poise further problems in terms of speed, reliability, traceability and data management within the polyglot system [15]. Thus, polyglot persistence architectures need to be designed on a per project basis. In this review, we focus on outlining the Apache Cassandra database to better understand how it would fit into polyglot persistence. Cassandra is the top ranked columnar NoSQL database[1] and is designed to handle large volumes of data spread across many commodity servers while still providing a highly available service which has no single point of failure [23]. The column based storage also juxtaposes with the row storage in the relational model. This highlights the different approaches to data modelling and data queries, allowing for a better appreciation for the need to research polyglot persistence. Further, Cassandra is well suited towards analytical data and IoT devices (as discussed in this review) which are becoming more prevalent and require focuesd research on database models which best suit this data.

This review will proceed to provide context of the development that happened in database systems, highlighting the need for new database models and the growing problems that data presents as our world becomes further data-centric. We then discuss various architectural components of Cassandra to better understand what can affect the performance and what data models are well suited for Cassandra. An analysis of research on Cassandra's use is performed, furthering our knowledge of the performance of Cassandra with various kinds of queries. Then benchmarking techniques are discussed, and finally we conclude by summarising our overall findings and declaring how the research will proceed.

2 DEVELOPMENT OF DATASTORES

In this section, we provide context of the various types of datastores, their evolution in the world of data, and an understanding of the key principles underlying the various database systems.

[1] db-engines.com/en/ranking

2.1 Before NoSQL

In the 1960s, computers became increasingly popular and cost effective for use by private companies to increase their data storage capacity [4]. The new direct-access storage (such as drum memory or disks which allow random access to data) contrasted with the old tape-based systems (requiring sequential access). This gave rise to the development of two main data models — the *network model* CODASYL (Conference on Data System Language) and the *hierarchical model* IMS (Information Management System). These database systems typically accessed data by following pointers between records. This meant that simple alterations to the database often required rewriting the underlying access scheme. Further, users required knowledge of how the data was physically organised in the machine in order to query information. This led to E.F. Codd conceiving of the relational database model in 1970 [9] where he describes

> Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Thus, he describes that the data models available at the time were not satisfactory solutions and proceeds to provide a formal treatment of the relational model based on the mathematical brances of set theory and predicate logic. This model disconnects the logical organisation of data from the physical information storage [4]. During the 1970s, the SQL language was also created, along with the Entity-Relationship (ER) model. These tools gave rise the the Relational Database Management System (RDBMS) and allowed for the relational model to become the de facto standard still implemented and taught as one of the main database models today.

The 1980s and 1990s saw a boom in the database market allowing the relational model to mature and for variations to develop. The object-oriented database system (later replaced by object-relational mapping (ORM) software) which aimed to take advantage of the new object-oriented programming languages, and Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP) are some of the variations developed over this period [4]. Data volume increased exponentially as computing power continued to grow and become more commonplace. This began to highlight problems in the relational model. For example, execution time became slower as the information volume grew larger [2]. Thus, the Web 2.0 era began to highlight the issues of the relational model and, in 2009, NoSQL became popular as the solution at an event about distributed databases [11].

2.2 NoSQL

NoSQL (Not Only SQL) encompasses a large class of databases which typically do not conform to the relational model [17]. In general, we can divide the NoSQL class into four core data models:

Key-Value Store Data is stored as a set of unique keys and an associated value [2]. The value can be anything, including another key. Structure is simpler allowing the query speed to be faster than the relational model, but the model is not powerful enough for range queries [16]. Examples include Amazon's DynamoDB and Riak.

Document Store Similar to a key-value store, this model also uses a key for lookup but the value is semantic (being semi-structured) and normally stored in JSON or XML format [18]. The added structure in comparison to the key-value store brings greater flexibility in accessing the data (e.g. retrieving parts of the document) [16]. Examples include MongoDB and CouchDB.

Column Store This is most similar to the relational model, but data is stored by column rather than by row. Examples include Apache Cassandra and HBase.

Graph Store Data that can be represented as a graph is stored in this model, thereby modelling the relationships [2]. Examples include Neo4J and OrientDB.

Each of these models have different strengths and weaknesses which allows them to better perform in certain situations. Thus, combining the various NoSQL data models to make use of their strengths individually will allow us to create a stronger data model overall.

NoSQL models also differ to the relational model in their schemas. Data models determine how data is organised and manipulated. The schemas define this structure. However, this limits relational databases to be a poor fit for semi-structured and unstructured data which is prevalent in big data [13]. Most NoSQL models are said to be *schemaless*. This does not indicate a lack of schema, but rather schemaless is "schema-on-read" implying that data can be stored without requiring the prior declaration of a schema [6]. That is, there is still a structure to the data and the schema information is implicit in the data and code. Data structure changes can be made without needing to shut down the database for alterations [20]. Normalisation is also often ignored in favour of de-normalisation in order to better match the structure of the data for how it is used.

A transcation is a unit of work in a database that contains a set of operations [11]. For these transactions to be well behaved, the ACID (Atomicity, Consistency, Isolation, Durability) model was proposed for relational databases. ACID increases complexity of database system design but eases the developer whom uses an ACID compliant database. Transactions are appropriately handled to consistently store the data and manage concurrency issues [19]. However, the ACID model creates a bottleneck in systems which grow larger in data size and have to deal with big data. The BASE paradigm contrasts this by better suiting NoSQL's emphasis on availability and performance [14]. BASE is characterised by [21]:

Basically Available Data is always available though it may not be consistent.

Soft State The system state might be changing even if there is no input. Thus, the system is always in a soft state because data may not be up-to-date.

Eventual Consistency Data is not guaranteed to be consistent after each operation, but, over time, the received changes will eventually propogate to all nodes in the system.

Note that ACID and BASE contrast each other in their design focus, but modern systems use a mix of the approaches [5]. This is highlighted by the CAP theorem which is where the principles of ACID and BASE are derived from. CAP represents [11]:

Consistency Once data is written, it is available and up to date for all users of the system.

Availability Each operation will always have a response.
Partition Tolerance Operations can still be completed when there is a network fault/failure.

The CAP theorem states that in a distributed system, all three characteristics of CAP cannot be guaranteed simultaneously but rather two of the three are chosen [18]. Data models can therefore be considered one of CA, CP, or AP.

As NoSQL has matured over the years, it has become apparent that no single type of database system will fit the needs of all applications [6]. *Polyglot persistence* is the term for a heterogeneous database system where a variety of data models are used to fit the needs of the problem rather than seeking an overall solution. Although relational databases are still the most used model today, polyglot persistence has been gaining popularity as the future solution and many relational models are evolving to support NoSQL features.

3 CASSANDRA ARCHITECTURE

This research focueses on Cassandra's place in polyglot persistence by benchmarking its performance and size. Whilst polyglot persistence should be designed on a per project basis, this research can assist with defining areas where Cassandra is likely to be a good fit, a poor fit and supplying performance levels that could be expected. Here we discuss the architecture of Cassandra to better understand what is happening to the data and how this can affect performance. Architectural analysis also highlights factors which will affect Cassandra's applicability in polyglot persistence. We begin this section with an overview of the terminology used in Cassandra [25]:

Node An individual machine storing data.
Data Centre A collection of connected nodes.
Cluster A cluster contians one or more data centres.
Column The atomic unit of information — a tuple of name, value and timestamp.
Super Column A tuple of name and value, where the value is another column.
Column Family A group of columns which "belong" together, similar to a table in a RDBMS.
Keyspace A container that maintains multiple column families.

The main use in understanding a keyspace is for setting the replication factor. Cassandra is an open source, peer-to-peer, distributed database with linear scalability and fault-tolerance on commodity hardware. It is categorised under the AP model from the CAP theorem but has a configurable consistency by adjusting the number of replicas to consider before confirming an operation [12]. The consistency levels include options such as ALL, QUORUM, ONE, TWO, THREE, and ANY. This highlights that the number of replicas chosen by the replication factor can thus affect this property, and with it, the speed performance of the database. If we let N be the number of nodes that store replicas of the data; W be the number of replicas needed to acknowledge the completion of a write; and R be the number of replicas contacted for consensus during a read operation, then when W + R > N, we can say that the write set and the read set always overlap and can guarantee strong consistency [29]. Strong consistency implies that after an update completes, a subsequent access will return the updated value. However, when W + R <= N, then we only have weak/eventual consistency because it is possible that the read set and the write set do not overlap. Weak/eventual consistency implies that there is no guarantee of returning the updated value on subsequent access after an update operation. Thus, we can use these frameworks to adjust levels of consistency and availability to suit our specific needs. This architectural choice creates awareness that frequently updated data, which is subsequently accessed, will struggle to make efficient use of the Cassandra design model and implies that Cassandra is likely to be a poor model fit if this is a requirement from our database.

In Cassandra's model, all nodes are operationally equivalent and a coordinator node is chosen when an operation is requested. The node responsible for the operation is found via the consistent hashing algorithm which then allows read and write operations to occur as the following discussion displays [25]. When a write operation is initialised, the data is first captured by a commit log which allows for data recovery. The data is then written to an in-memory structure called a memTable. When the memTable is full, data is pushed to the SSTable on disk and an SSTable index is kept in memory. These structures are append-only which allows for fast reads, especially because they are on consecutive disk blocks. Periodically, this data will go through a process called compaction which will clean the data (keeping only the most recent values of an item) and perform compression. This data is then permanently stored.

A read operation first utilises the memTable, then consults the Bloom filter (an algorithm for testing membership of an element in a set), then the SSTable (if necessary) before finally using the permanent data store. Fully understanding the architecture and discussing further principles such as gossiping and vector clocks is not important for discussions of performance, but there are two key takeaways from this discussion which are outlined here. The first is that data progressively moves from faster data stores to slower ones and there are tools in Cassandra which utilise this knowledge to further optimise operations. Better performance utilisation of Cassandra is possible by designing our models and queries to focus on data that is stored in these faster mediums, as long as the data is not updated soon after being written. Secondly, knowledge of this architecture enlightens that poor utilisation of this framework can lead to further degraded performance. For example, in Discord's blog post of 2017[2], they discuss their growing need for a big data solution and how Cassandra was their implemented solution. Discord discusses the anti-pattern of read-before-write where a user deletes a message at the same time as another user edits that message. This can lead to nullified data and the solution Discord used was to recognise such a message as corrupt and delete it. However, due to the architecture of Cassandra, delete operations are not immediate. Delete operations must be replicated to other nodes even when other nodes are unavailable. Cassandra achieves this by treating delete operations as "tombstone" write operations. Tombestones are skipped on read operations and are only removed when the compaction process occurs. The tombstone lifespan is configurable but defaults to ten days. Thus operations which create many tombstones can create clutter and bring the performance of the database to a halt. In particular, Cassandra is implemented in Java and thus

[2] discord.com/blog/how-discord-stores-billions-of-messages

a large number of tombstones generates a large amount of garbage for the Java Garbage Collector (GC) to collect. This ultimately halts performance while the Java GC performs its job. This brings to light the need to design the models and queries to be conscious of this. For example, rapidly updated data figures are likely to be a poor fit for the Cassandra model.

We close this section with a discussion of how data is physically stored in the column model [1]. The columns of data are stored together in memory. Compression works well on this model, allowing large reductions in data store size. When data is retrieved, a *block* of data is retrieved which may hold a certain amount of data. Thus, when the data items are physically stored together, we perform fewer block retrievals and therefore fewer I/O requests because the data required is stored on fewer blocks. I/O is a very expensive operation and is part of why there are various methods in Cassandra (such as the bloom filter and memtables) to minimize the need to retrieve data from the disk. This physical storage technique assists to further reduce the time to access data when performing column based operations. Hence, Cassandra is well suited towards analytical operations which operate on columns of data. However, this also implies that Cassandra is slower to perform transactional queries which will have to access various blocks in order to access the required data and complete the transaction. This is augmented by the clusters Cassandra is able to utilise techniques which only require the database to query specific nodes, thus minimizing interaction with the entire database.

4 CASSANDRA CASE STUDIES

We provide an analysis of prior areas where Cassandra has been implemented and studied. This will highlight areas where it has been well utilised, as well as issues related to implementation details and where Cassandra has been a poor fit. We also look at the queries run in Cassandra and how the column model limits or alters our logic in design of the model and our queries.

In the relational model, the data model is created with an ER diagram and the queries designed around the outputted model. The relational model has been designed that the data is retrievable if the tables have been modeled properly. In contrast, designing the model in Cassandra begins with focus on the query model [29]. This is key to modelling in Cassandra — the queries must be determined first and then the data is organised based on the query patterns [26]. This structure increases the speed of query response in Cassandra because data is stored the same way it is usually queried. Further, the lack of *joins* and *group by's* in Cassandra require this structure for certain queries to be applied. The partition keys in a column family of Cassandra are used to partition the data for load balance and can only be queried on equality, while the clustering keys support both range and equality filters [26]. Thus, applying different columns in these positions will affect the query performance. The example utilised in J. Qiao's et al. study [26] best displays this:

> If the columns (altitude, time) are clustering keys, then queries such as "Find the humidity at some locations where altitude=100Pa and between time [2018.05.01, 2018.05.02]" have low latencies, because data is sorted by the time column on disk. If the columns (time, altitude) are clustering keys, then the latencies of the former queries increase but queries such as "Find the humidity at some locations where altitude in [100Pa, 1000Pa] and time=2018.05.01" will have low latencies.

One of Cassandra's primary features is to handle application workloads that require high performance when there is significant write volumes [7]. Therefore, Cassandra is well utilised in storing user activity updates and application statistics. There are many studies which have studied Cassandra focusing on its performance in analysing text. Often, Tweets are chosen as the text to be analysed. In a study comparing various NoSQL models' performance on analysing Tweets, Cassandra was found to be the most efficient [3]. A study on real-time text analytics also used Tweets as the medium of studying performance but the notion was extended that Cassandra is well suited towards monitoring resources, fraud detection, and user stream analysis [25]. These models work well with Cassandra because analytical information is performant when using a column-based data store which allows for efficient aggregation queries. In contrast, queries with short scans perform terribly on Cassandra [10].

These case studies highlighted that high write throughput data is suitable for Cassandra, as well as analytical data work. Data with fast changing values is not well suited to Cassandra because the architecture is not designed to handle these operations well. Thus performance is hindered in utilising Cassandra with poorly suited data operations. The design of the Cassandra model should also be query-focused and the designer should be aware of possible performance restrictions that they could be implementing based on how the model is designed.

5 BENCHMARKING

We now look at benchmarking techniques and tools for database systems. In particular, we focus on NoSQL benchmarking solutions and what insights we can draw from analysing the results.

Performance benchmarks focus on the latency of requests. This is an important focus point because there is a tradeoff between latency and throughput: as the load increases, the latency of individual requests also increases [10]. Application designers must therefore provide enough servers to maintain an acceptable latency. By increasing the workload applied, we can discover the performance of the system because it has its resources depleted. A better performing system will acheieve the required latency and throughput with fewer servers.

The Yahoo! Cloud Serving Benchmark (YCSB) framework was created to facilitate performance comparisons of modern database solutions due to the poor fit of benchmarking tools before the NoSQL boom [10]. YCSB has since been used in a variety of published benchmark comparisons and provides a standardised way to perform comparisons between different systems [14]. This is achieved through *workloads*. Workloads in YCSB allow the system to be benchmarked under various sectors of the performance space. Examples of workloads are included in the *core* workloads of YCSB [24]. These focus on operations such as:

Write Intensive 50% Read + 50% Update
Read Intensive 95% Read + 5% Update
Read Latest 95% Read + 5% Insert
Read Modify Write 50% Read + 50% Update

Note that these workloads also differ in their random choices — which operation to perform, which record to access. These decisions are governed by random distributions. For example, the distribution could be a *Uniform* distribution which allows an item to be chosen uniformly at random, or a *Zipfian* distribution could be chosen such that certain records are extremely popular. This allows us to model the distribution of data access to mimic various use cases. Thus, for example, a *Read Latest* workload could simulate the use case of user status updates.

A key part of YCSB is its extensibility, allowing new workload types to be defined for personal benchmark tests. The core package workloads are different from traditional benchmarking tools because they define the workload characteristics for which various database models have been created [10]. This is different from the traditional benchmarks (such as TPC-C) which focused on modelling particular applications. While benchmarks such as TPC-C allow for more realistic performance results, the results are limited to a narrow set of use cases. This would be a poor benchmark for our research as we intend to gather an array of statistics on Cassandra under a wide range of workload characteristics.

YCSB can offer benchmarking on the performance of Cassandra, but polyglot persistence requires consideration of other factors such as the thread and memory usage, the data store size, and various other system metrics. In addition to the benchmarks supplied by YCSB, it is beneficial to also have internal metrics on the state of the database nodes because this allows us to udnerstand resource utilisation. The metrics can be obtained through JMX (Java Management Extensions) because Cassandra exposes various metrics through MBeans (Managed Beans) [27]. These metrics can be gathered and exported in research to gain further insight into how the performance analytics were achieved.

These frameworks are suited towards being implemented in containers on an AWS EC2 instance. After simulating the data interactions, various time series data files containing metric information will be outputted which can then be further analysed. Use of statistical tools and visualisations on the outputted data can help realise the analytical insight that this research intends to acheieve.

6 CONCLUSION

This review began by noticing that the relational model could not handle the big data era, even when trying to extend the model by providing scalable solutions and the two-phase commit [8]. The relational model is still applicable to situations and has benefits such as ACID and concurrency control, as well as developer familiarity and full-fledged tools in the RDBMS. But, there are biga data problems for which the relational model simply does not work and NoSQL is required. While NoSQL models all allowed big data to be managed, the individual solutions categorised as NoSQL are specialised models which each have their own strengths and weaknesses. Polyglot persistence was recognised as the solution to this problem such that an appropriate data model could be selected to best handle each individual problem. However, the system implemented in polyglot persistence should be defined per project.

This review focused instead on how Apache Cassandra might fit into such a system. The architecture of Cassandra was discussed to better understand what could affect the performance and storage size of a Cassandra database. These details show the types of data and queries which may best be suited for Cassandra. This discussion is further extended when looking at the case studies. The case studies gave us practical exposure to where Cassandra performed well, and where it performed poorly. We found that Cassandra is well suited to analytical work and for logging data such as time series where there is a large number of write operations. Finally, we discussed benchmarking tools and techniques for Cassandra. Specific NoSQL benchmarking tools have already been implemented to gather statistics on the database performance under various workloads. These workloads focus on various areas of stress, such as read or write heavy operations. Finally, internal metrics can assist with understanding the reasons behind the performance of the model and provide deeper insight into Cassandra's applicability when designing polyglot persistence.

Polyglot persistence poises its own set of problems which are not tackled here. For example, there are structural concerns with designing such a system which may affect the reliability and speed performance of the model as well as usability and compatibility concerns [28]. Inter-model communication may grow in complexity and traceability of data becomes a concern for troubleshooting. Thus, this review has been limited to a discussion only on the performance and storage size of Apache Cassandra in the light of implementing it as one of the databases in polyglot persistence.

Further, this research proceeds to perform an experimental analysis on Cassandra. Workloads focusing on a variety of read, write, and read-write operations will be implemented and performance compared between Cassandra and MySQL. This review has highlighted the large variety in performance readings that are possible due to the nature of the work that the database model intends to do, and because of the configurability of Cassandra. A Latin Square Design will be well suited to gather performance statistics on Cassandra under a variety of workloads and configurations. The analysis of this data can thus provide a firm understanding of whether Cassandra is an appropriate model fit and a strong prediction of how Cassandra will perform should it be chosen as one of the models in a project using polyglot persistence.

REFERENCES

[1] 2020. https://youtu.be/XNrsRVMfj1c

[2] Veronika Abramova and Jorge Bernardino. 2013. NoSQL databases. *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13* (2013). https://doi.org/10.1145/2494444.2494447

[3] Souad Amghar, Safae Cherdal, and Salma Mouline. 2020. Storing, preprocessing and analyzing Tweets: Finding the suitable NoSQL system. (2020). arXiv:arXiv:2005.01393

[4] Kristi L. Berg, Tom Seymour, and Richa Goel. 2012. History of databases. *International Journal of Management & Information Systems (IJMIS)* 17, 1 (2012), 29–36. https://doi.org/10.19030/ijmis.v17i1.7587

[5] Eric Brewer. 2012. Cap twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/mc.2012.37

[6] Carlos J. Fernández Candel, Diego Sevilla Ruiz, and Jesús J. García-Molina. 2021. A Unified Metamodel for NoSQL and Relational Databases. (2021). https://doi.org/10.1016/j.is.2021.101898 arXiv:arXiv:2105.06494

[7] Jeff Carpenter and Eben Hewitt. 2022. *Cassandra: The definitive guide: Distributed data at web scale* (3 ed.). O'Reilly.

[8] Rick Cattell. 2011. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record* 39, 4 (2011), 12–27. https://doi.org/10.1145/1978915.1978919

[9] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (2010). https://doi.org/10.

1145/1807128.1807152

[11] Alejandro Corbellini, Cristian Mateos, Alejandro Zunino, Daniela Godoy, and Silvia Schiaffino. 2017. Persisting big-data: The nosql landscape. *Information Systems* 63 (2017), 1–23. https://doi.org/10.1016/j.is.2016.07.009

[12] Elif Dede, Bedri Sendir, Pinar Kuzlu, Jessica Hartog, and Madhusudhan Govindaraju. 2013. An evaluation of Cassandra for Hadoop. *2013 IEEE Sixth International Conference on Cloud Computing* (2013). https://doi.org/10.1109/cloud.2013.31

[13] Adanma Cecilia Eberendu. 2016. Unstructured Data: An overview of the data of Big Data. *International Journal of Computer Trends and Technology* 38, 1 (2016), 46–50. https://doi.org/10.14445/22312803/ijctt-v38p109

[14] Deka Ganesh Chandra. 2015. Base analysis of nosql database. *Future Generation Computer Systems* 52 (2015), 13–21. https://doi.org/10.1016/j.future.2015.05.003

[15] Felix Gessert and Norbert Ritter. 2016. Scalable Data Management: NoSQL data stores in research and Practice. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)* (2016). https://doi.org/10.1109/icde.2016.7498360

[16] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. 2016. NoSQL database systems: A survey and decision guidance. *Computer Science - Research and Development* 32, 3-4 (2016), 353–365. https://doi.org/10.1007/s00450-016-0334-3

[17] Venkat N. Gudivada, Dhana Rao, and Vijay V. Raghavan. 2014. NoSQL systems for Big Data Management. *2014 IEEE World Congress on Services* (2014). https://doi.org/10.1109/services.2014.42

[18] Jing Han, Haihong E, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications*. 363–366. https://doi.org/10.1109/ICPCA.2011.6106531

[19] Ricardo Jimenez-Peris, Marta Patino-Martinez, Ivan Brondino, and Valerio Vianello. 2016. Transactional processing for polyglot persistence. *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)* (2016). https://doi.org/10.1109/waina.2016.144

[20] Samiya Khan, Xiufeng Liu, Syed Arshad Ali, and Mansaf Alam. 2019. Bivariate, Cluster and Suitability Analysis of NoSQL Solutions for Different Application Areas. (2019). arXiv:arXiv:1911.11181

[21] Pwint Phyu Khine and Zhaoshun Wang. 2019. A review of Polyglot persistence in the big data world. *Information* 10, 4 (2019), 141. https://doi.org/10.3390/info10040141

[22] Rob Kitchin and Gavin McArdle. 2016. What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets. *Big Data & Society* 3, 1 (2016). https://doi.org/10.1177/2053951716631130

[23] Avinash Lakshman and Prashant Malik. 2010. Cassandra. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

[24] Yaser Mansouri, Victor Prokhorenko, Faheem Ullah, and M. Ali Babar. 2021. Evaluation of Distributed Databases in Hybrid Clouds and Edge Computing: Energy, Bandwidth, and Storage Consumption. (2021). arXiv:arXiv:2109.07260

[25] Hassan Nazeer, Waheed Iqbal, Fawaz Bokhari, Faisal Bukhari, and Shuja Ur Rehman Baig. 2017. Real-time Text Analytics Pipeline Using Open-source Big Data Tools. (2017). arXiv:arXiv:1712.04344

[26] Jialin Qiao, Xiangdong Huang, Lei Rui, and Jianmin Wang. 2018. Heterogeneous Replica for Query on Cassandra. (2018). arXiv:arXiv:1810.01037

[27] Anup Shirolkar. 2020. Apache Cassandra Monitoring: A Best Practice Guide. instaclustr.com/blog/cassandra-monitoring-best-practice-guide/

[28] Luís H. Villaça, Leonardo G. Azevedo, and Fernanda Baião. 2018. Query strategies on polyglot persistence in microservices. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (2018). https://doi.org/10.1145/3167132.3167316

[29] Guoxi Wang and Jianfeng Tang. 2012. The nosql principles and basic application of Cassandra Model. *2012 International Conference on Computer Science and Service System* (2012). https://doi.org/10.1109/csss.2012.336