Using HBase to Implement Speed Layer in Time Series Data Storage Systems

Milko Marinov
Department of Computer Systems and Technologies
University of Ruse, Ruse, 7017, Bulgaria

Abstract—**In recent years, modern systems have become increasingly integrated, and the challenges are focused on delivering real-time analytics based on big data. Thus, using standard software tools to extract information from such datasets is not always possible. The Lambda Architecture proposed by Marz is an architectural solution that can manage the processing of large data volumes by combining real-time and data batch processing techniques. Choosing a suitable database management system for storing large volumes of time series data is not a trivial issue as various aspects such as low latency, high performance and the possibility of horizontal scalability must be taken into account. The new NoSQL approaches use for this purpose non-relational databases with significant advantages in terms of flexibility and performance in comparison with the traditional relational databases. With reference to this, the purpose of this paper is to analyse the general characteristics of time series data and the main activities performed by the Speed layer in a system based on the Lambda Architecture. Based on this, the use of a column-oriented NoSQL DBMS as a system for storing time series data is justified. The paper also addresses the challenges of using HBase as a system for storing and analysing time series data. These questions are related to the design of an appropriate database schema, the need to achieve balance between ease of access to the data and performance as well as considering the factors that affect the overload of individual nodes in the system.**

Keywords—Lambda architecture; speed layer; time series data; data storage system

I. Introduction

The accelerated development of technologies applied to big data has caused significant changes in the subject areas of storage, retrieval, and processing of data. Nowadays, the problems related to the big data are connected not only to the volume of data. Much of the data are acquired in real time and is most valuable if its interpretation takes place as it arrives [1,2,3]. Synthesizing, processing, and transforming this big data to valuable information is one of the great challenges of the technological world today.

In big data systems, an important property of data related to its processing is immutability. In such systems, to prevent data loss and data corruption, data are processed in a way that records can never be modified or deleted. By its nature, immutable data are simpler than mutable data [4]. This organization allows the system only to create and read records (CreateRead) as opposed to the additional capability of updating and deleting records (CRUpdateDelete) as implemented in relational databases. Thus, the write operations only add new data units [5]. This approach makes data processing highly scalable. The data system itself becomes a kind of a logging system, which adds a timestamp and a unique identifier to the data record, which is then kept in the data store.

Different architecture models are used when building the big data ecosystem. The Lambda Architecture, proposed by Nathan Marz [6], is a solution which combines real-time data processing techniques with batch processing techniques. The Lambda Architecture is a big data management software paradigm that supports data processing by balancing the performance, latency, and fault-tolerance of the system that is based on this architecture [7,8]. There is no single integrated tool that provides a comprehensive solution with reference to better accuracy, low latency, and high performance. Therefore, it is necessary to apply the idea of using a set of tools and techniques to build a comprehensive big data management system. With reference to this, Lambda Architecture defines several layers that correspond to a set of tools and techniques for building a big data processing system, i.e., a speed layer, a serving layer, and a batch layer [9]. The increasing need for new and improved storage and retrieval mechanisms resulted in the development and use of NoSQL database management systems such as HBase, MongoDB, Cassandra, CouchDB, Hypertable and big data platforms such as Hadoop and Spark [10,11,12]. Lambda Architecture defines a logical, well-motivated approach in linking these technologies together to build a system that meets user requirements. Each software tool offers its own trade-offs, but when these tools are used together, scalable systems with low latency, high fault-tolerance, and minimal complexity can be realized [13].

The main objective of the current research is to justify the use of a column-oriented NoSQL DBMS as a system for storing time series data. This suggestion is based on the general characteristics of time series data and the main activities performed by the Speed layer in a Lambda Architecture based system. This paper focuses on the challenges of using HBase as a system for storing and analysing time series data, related to designing an appropriate database schema, the need to strike a balance between ease of data access and performance, and consideration of factors that affect the overload of individual nodes in the system.

The remainder of this article includes the following: Section 2 surveys some related studies. Section 3 presents an overview of the Lambda Architecture with an emphasis on the Speed layer. Section 4 discusses the main characteristics of time series data. Section 5 outlines the key problems that arise when using HBase as a system for storing time series data and techniques for solving these issues. Section 6 contains the conclusion.

II. Related Work

One of the big challenges to extracting data from processes is handling real-time event data and providing operational support for ongoing processes. The research presented in [7] focuses on a real-time process discovery algorithm implemented by the authors in an integrated platform that is built using the Lambda Architecture principles. The proposed architecture solution makes it possible to scale up to big data processing tasks.

Trajectory prediction problems are classified in the category of big data processing tasks. In [1], a platform for predicting the next position of moving objects based on the Lambda Architecture is discussed. In the presented platform, data analysis is performed both in a batch mode and a real time mode. In the proposed system, the Lambda Architecture is applied to combine predictions made by heavy-weight models trained by using all available data, implemented by the Batch layer, on one hand, and light-weight models trained by using real-time data obtained from small samples, implemented by the Speed layer, on the other hand.

Maeda & Gaur propose in [14] a Lambda Architecture of a failure mode identification system for industrial assets that achieves low initial implementation costs by providing reasonable accuracy in object classification. The architecture consists of a data acquisition node, such as a Raspberry Pi, in which lightweight computations are performed. This node processes high-speed vibration data in real-time to extract important characteristics about objects and uses a deep learning engine that is trained in a cloud platform.

The nature of heterogeneous IoT devices introduces the challenge of collecting and processing the large data sets for their analysis in detecting cyber-attacks in near real-time. However, the traditional Intrusion Detection System cannot cope with such a problem due to scalability limitations and insufficient storage and processing capabilities. To address these challenges, Alghamdi & Bellaiche present in [15] a model of Intrusion Detection System based on the Lambda Architecture. The proposed solution enables the detection of suspicious activities in real time and allows them to be classified by analyzing historical data in the Batch layer. Suthakar et al. describe in detail in [9] a study of an Optimized Lambda Architecture using the Apache Spark ecosystem, which involves the modelling of an efficient way to transparently connect batch processing and real-time processing.

Data storages, whose architecture is based on the relational data model, are not able to meet the current needs in terms of data storage requirements as well as data read and write speed requirements. Therefore, research related to real-time data management systems is based on distributed storage organized in a cluster, for example built on the Hadoop ecosystem. The study in [4] proposes the use of HBase DBMS, whose storage structure is based on the column-oriented data model. The described system provides real-time monitoring of sensor data and satisfies data storage and processing requirements. In [16], the researchers discuss the use of the Hadoop ecosystem in finance.

Bao & Cao analyse in [17] the challenges to storing and retrieving social network data. In this study, they present a query optimization scheme based on HBase DBMS. The table structure is designed according to the characteristics of HBase, such as the high efficiency of row keys and storage which is based on the column-oriented data model. A coprocessor is used to design a secondary index, which transforms some queries to the attributes into row key queries of the index tables so that this can support flexible queries to social network data with high scalability and low latency. In [18,19], a similar solution is proposed regarding the indexing mechanism in HBase for sensor data processing. A secondary memory index mechanism is used. The retrieval speed of the indexed data is significantly improved because the indexing is stored and maintained in the memory.

In order to increase the performance of the storage process of data retrieved in the real time, the process must be distributed. It can be realized by taking advantage of the MapReduce technology. HBase offers a data loading tool that can process data stored in TSV or CSV formats which is called ImportTSV [20]. This tool is based on the MapReduce model. Azqueta-Alzúaz et al. in [10] identify and quantify the problems associated with loading massive big data. They propose a tool for parallel massive data loading using HBase. This solution overcomes the defined problems of data loading in HBase.

III. Lambda Architecture Overview

The CAP theorem applies to trade-offs in distributed systems [6,11]. It states that in a distributed data storage system only two of the characteristics availability, consistency, and partition tolerance can be guaranteed. The meaning of partition tolerance is that the system characteristics are maintained even in case of network failures. This requirement is supported in modern systems. Therefore, when designing a data management system, a compromise has to be made between consistency and availability. The Lambda Architecture is designed to address the trade-offs that must be made in a distributed data storage and processing system. The main idea is to create two input data streams to be processed separately and to combine later the obtained results. The components of the Lambda Architecture are the Batch layer, the Serving layer, and the Speed layer (Fig. 1).

The components through which the Batch layer and the Serving layer are implemented involve the execution of computational functions on each piece of data, i.e., on the data as a whole [6]. These layers satisfy all characteristics that are required for a data processing system except one: low latency updates. The only task of the Speed layer is to satisfy the latter requirement. Executing functions on an entire data set, which could be measured in petabytes, is an operation that requires considerable computational resources. The Speed layer should use a completely different approach from the one used by the Batch and Serving layers to reduce latency of updates as much as possible. Whenever data changes, the Speed layer recalculates only those results that depend on the changed data, i.e., incremental computation is performed. The main functionalities that must be implemented by the Speed layer are storing real-time views and processing the incoming data stream to update these views. The Speed layer is more complex than the Batch layer and more errors may occur when storing and processing the data. The Speed layer is only responsible for the data that will be included in the Batch views, which are part of the Serving layer. The amount of this data is significantly smaller than the amount of the main dataset. This allows more flexibility in the design of the Speed layer. Real time views that are obtained as a result of the Speed layer are temporary, i.e., they are not stored permanently. Once the data are accepted and used in the batch views, it can be discarded from the Speed layer.

The choice of Lambda Architecture when building a distributed system may have the following disadvantages:

- The different layers in this architecture make it too complex in terms of synchronization between layers. It is possible that the cost of synchronization between the batch and speed layer will increase. More computational resources, time and efforts are required to run both the Batch and Speed layers.
- The implementation of the Lambda Architecture requires the use of a large number of technologies, which makes it difficult to find specialists who know the whole set of tools.
- Under certain conditions, this architecture could contain a large surplus of tools that need to be configured for each scenario.

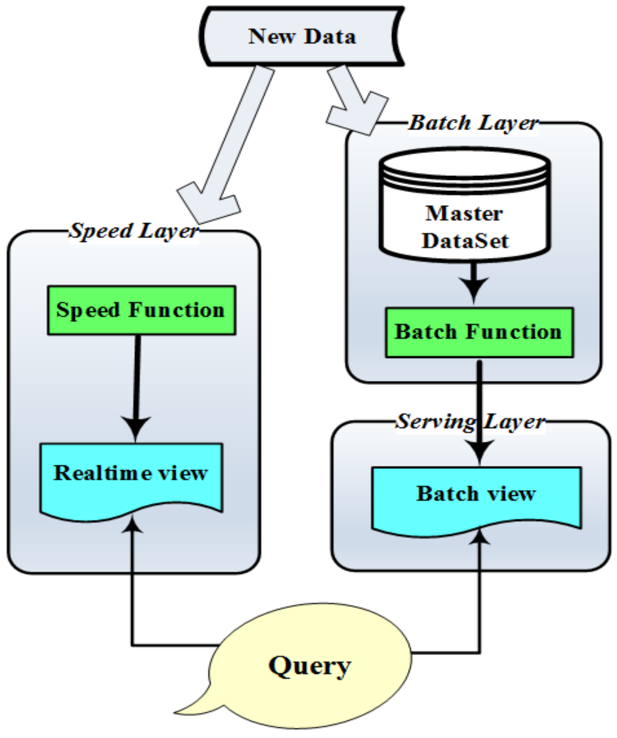

Fig. 1. Lambda Architecture Main Components.

IV. Characteristics of Time Series Data

Time series data are a set of data points, and each value is connected to a timestamp. Formally, a time series can be defined as a set of pairs, each consisting of a timestamp and a value [2,20]. Generally speaking, time series datasets are sequences of data records ordered according to the time of their occurrence. Time series data is characterized by chronological consistency, large volume and high degree of competitiveness. The time series datasets are used in situations in which, once measurements have been taken, they are not revised or updated. However, they are rather used when the set of measurements is accumulated by adding new data for each parameter that is measured at each new time point. Time series data entries are rarely changed, and time series data are extracted by reading a continuous sequence of samples, obtained after summarizing or aggregating the samples, in the order in which they are received. These time series data characteristics limit the requirements for the technology used to store this type of data [23].

Although the idea of collecting and analyzing time series data is not new, the combination of some factors such as the volume of current datasets, the speed of data accumulation, and the variety of new data sources turn the task of building scalable time series databases into an enormous challenge. Time series data require different approaches and different tools [24].

It is difficult to store data whose nature is unpredictable. This makes it necessary to store and process time series data in a database. Especially when it comes to large volumes of time series data, the requirements for its efficient storage become increasingly important. A time series database is a way of storing multiple time series so that queries to retrieve data from one or more time series for a specific period can be executed efficiently. Time series databases allow users to predict the behavior of an object by analyzing its past states. The queries made to the time series data can be implemented as large, sequential scans, which are very efficient if the data are stored appropriately in a time series database. And if the data volume is very large, a non-relational time series database based on a suitable NoSQL data model is usually needed to provide sufficient scalability. In addition to considering the characteristics, the nature of time-series data, as well as the requirements for high storage reliability and horizontal scalability, require the use of a NoSQL distributed time-series database as the most suitable for storing and processing all these large volumes of data. The new NoSQL-based approaches use non-relational databases for this purpose with significant advantages in terms of flexibility and performance over traditional relational databases.

V. Storing and Processing Time Series Data in HBase

HBase is a distributed DBMS built on HDFS. One of the significant advantages of HBase is the ability to combine real-time queries with batch MapReduce jobs in the Hadoop ecosystem, using HDFS as a shared storage platform [21,22]. All rows in HBase are sorted lexicographically by row key. In the column-oriented model, the data are organized at the logical level into tables, rows, and columns. A table in HBase is multi-dimensional and can be queried using the primary key. The key structure is presented in Fig. 2. HBase columns can have multiple versions of the same row key. A typical HBase cluster has one active master node, one or more backup masters, and regional servers (Fig. 3). The HBase Master node assigns the corresponding regions to Region Servers. The first one is the ROOT region, which contains all the META regions which must be assigned. It also monitors the state of Region Servers and if it detects a failure in any Region Server, it restores it using the replicated data. In addition, the HBase Master is responsible for table maintenance. The tasks which HBase Master performs are related to adding or deleting tables and making changes to the table structure. The Region Server handles read and write client requests. It interacts with the HBase Master to obtain a list of regions to serve and informs the master node of its availability.

When HBase is used as a system for storing time series data, problems arise, and they are related to overloading one of the Range Servers and scattering of data. Since there is a wide variety of time series data, it is necessary to take into account the specific features of every type of data when it is stored in HBase.

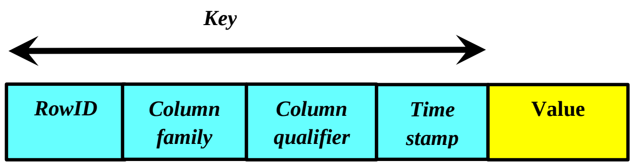

Fig. 2. Key Structure in Column-oriented Data.

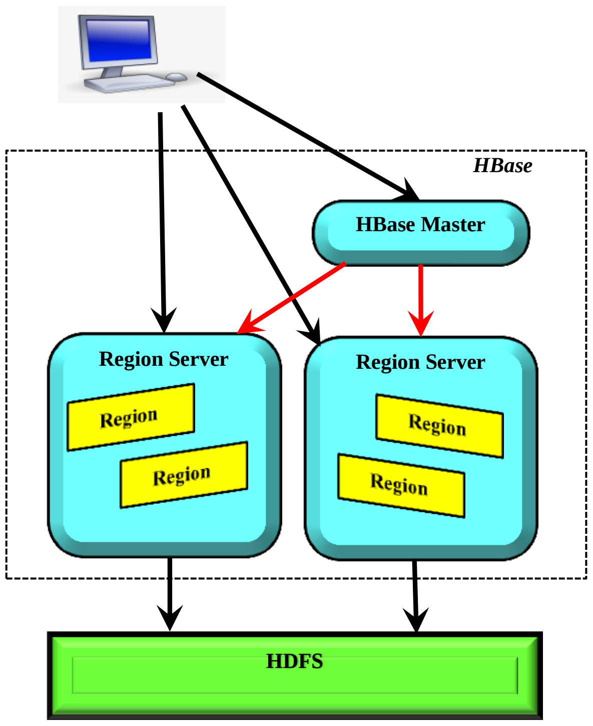

Fig. 3. HBase General Architecture.

The most logical key that can be used to store time series data is the timestamp. This guarantees the uniqueness of the key for every measurement at a specific point in time. With this approach to data storage, the data for each timestamp value can be accessed by performing a single read operation. It is also possible to easily perform a scan of a range of key values. A fast scan is guaranteed because all rows stored in HBase are sorted by key. However, the following two problems arise when this approach is applied and either of them can reduce the system performance:

- First, such an organization of the data may cause overloading of some of the Region servers. This is because at the time of writing, all data are concentrated in the regions that serve the corresponding key values, while in the other regions no data are being written. Similarly, when performing a read query targeting the most recent data, a small number of regions will be accessed. This will reduce the effect of being able to access a larger number of regions in parallel.

- Second, relatively few columns are stored in each row, and this can be very inefficient because little data are read at once and there are too many Bloom filter values that will be used in the search process.

When all new rows are written sequentially in HBase, they are all placed on the same server because they are sorted, and this requires them to be close together. HBase has a built-in automatic sharding mechanism. The new regions (areas of the hard disk where the data are written) which result from the sharding operation will be used later. In this way, they will balance the overload. In practice, the overload will not be noticeable at low write speeds as the RegionServer will be doing perfectly fine. This, however, will not lead to the efficient use of the entire HBase cluster because only one server will be used.

To avoid key concentration in sequential writes, it is necessary to do one of the following when developing the logical organization of the data:

- Indexing must be avoided if possible. In the case of time series data, the timestamp should not be used as the only key, some other data element should be added as well. In other words, a composite key should be used to uniquely identify the row and the moment when the corresponding value was received, or the event occurred.

- Storing write operations randomly. The main problem with time series data is that the sorted timestamps are the essential information, and they must be present in the data in one or another way. Therefore, using random writes in time series data will make the writes faster but the reads slower because the data will have to be collected from many locations. In some cases, even pseudo randomization can help to reduce the load of some servers.

- Adding a fragment identifier at the beginning of the key. In this way, the load can be distributed among the set of Region servers. When the data are being read, it must be read by each server. Then the fragment identifier must be added to the timestamp, and finally the query results must be combined in the memory.

If the write of a time series data set has a rather small row (with a small number of columns), this will lead to a problem associated with cache-hit and large Bloom filters when data are searched for. Cache-hit is a condition in which the data that is requested for processing is in the cache memory. In terms of HBase, this can be explained as follows. Hadoop reads blocks from HDFS, which typically range in size from 64 MB to 256 MB. If the data to be read and written is much smaller than this size, this will lead to inefficient cluster operation and hence cache memory problems. Bloom filters are used to answer the question whether, based on the key, it is possible to locate the data in the corresponding region. The answer is not definite, but it should be understood as maybe yes, which requires reading the region and searching. If the keys identify rows containing little information (called thin rows), this will cause the use of too many Bloom filter keys, which will take up disk space and reduce the efficiency of their use.

One technique for increasing the speed at which data can be retrieved from a time series database is by storing a large number of values in each row. In DBMSs that support a column-oriented data model, and HBase is exactly such a system, the number of columns is almost unlimited. This feature can be used to store numerous values within a single row. This allows data points to be accessed at a higher speed. The speed at which data can be scanned depends on the number of rows scanned, the total number of values retrieved, and the volume of data retrieved. If the number of rows is reduced, the fraction of data loss in retrieval will be significantly reduced, resulting in an increase in retrieval speed. For example, if the row key contains <time_series_ID> and <Start_time_of_Tme_Window>, and the column names correspond to the offset from the start of the time window when the value of the corresponding data element will be written, then the result will be a table with many columns. This means that the data retrieval from a particular time series for a particular time period would involve mainly sequential read operations and would therefore be much faster in comparison to a situation in which the rows were scattered.

Such an organization leads to a reduction in the number of rows in the table. In addition, rows that contain data from the same time series are close to each other when the data are stored. To take advantage of the benefits of this structure with reference to its performance, the number of samples in each time window must be sufficiently large. This will cause a significant reduction in the number of rows that must be retrieved.

This technique is similar to the default table structure used by OpenTSDB [5,20]. OpenTSDB is an open-source distributed time series database designed to control clusters of commodity servers with a high level of granularity. The interaction between OpenTSDB and HBase is presented in Fig. 4. An OpenTSDB consists of interacting components for loading and accessing time series data. These include data collectors, Time Series Daemons (TSDs), and various user interface management related functions. Each TSD is independent. There is no master, no shared state, and as many TSDs as required can be run so that the system can handle the workload. Each TSD uses HBase to store and retrieve time series data. On the servers where measurements are being taken, there is a collection process that sends data to the TSD. The TSDs are responsible for finding time series to which data will be added and each data point will be inserted as it is received in the data storage layer. OpenTSDB uses HDFS as a file system for storing large data sets. A simplified web-based user interface is supported, and users query various metrics in real time through it.

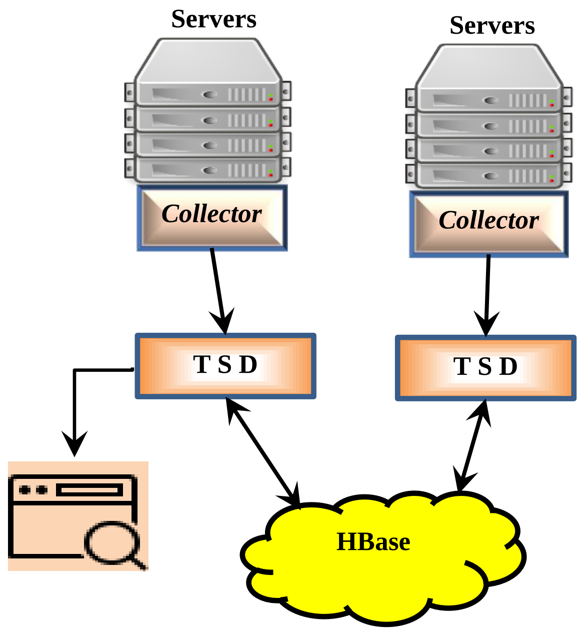

Fig. 4. OpenTSDB Components Interaction.

VI. Conclusion

The Lambda Architecture allows users to optimize the cost of processing large volumes of data by dividing the storage and processing of input data into two streams - data that needs to be processed in real time and data on which batch processing will be performed. The Lambda Architecture provides a consistent approach to building a big data system that can perform real-time data storage and processing in a low-latency, high-throughput, and fault-tolerant manner. The Speed layer implementation needs to consider the characteristics of time series data as well as the requirements of high storage reliability and horizontal scalability. This requires the use of a NoSQL distributed time series database based on a column-oriented data model.

When using HBase as a time series data storage system, it is necessary to properly design a row key that is based on the timestamp in order to overcome the problems associated with overloading one of the Range Servers and the data scatter problem. The effectiveness of the speed layer can be significantly increased when HBase is integrated with OpenTSDB. All OpenTSDB data points are stored in one “big” table, which is called tsdb by default. All values are stored in a single column family. This is done to take advantage of the key ordering in HBase and the distribution of regions over individual RegionServers.

The author's further efforts will be focused on expanding research on the application of other software architectures used to build time series data storage systems and incorporate these technologies in the Distributed Databases course of the Computer Systems and Technologies master degree at the University of Ruse.

References

[1] E. Psomakelis, K. Tserpes, D. Zissis, D. Anagnostopoulos and T. Varvarigou, “Context agnostic trajectory prediction based on λ-architecture,” Future Generation Computer Systems, vol. 110, pp. 531–539, 2020.

[2] A. Noury and M. Amini, “An access and inference control model for time series databases,” Future Generation Computer Systems, vol. 92, pp. 93–108, 2019.

[3] A. Pandya, O. Odunsi, C. Liu, A. Cuzzocrea and J. Wang, "Adaptive and efficient streaming time series forecasting with Lambda architecture and Spark," in 2020 IEEE International Conference on Big Data (Big Data), pp. 5182-5190, 2020.

[4] J. Yang, X. Chi, M. Zhu and W. Wang, “Research and Design of Sensor Data Management System Based on Distributed Storage,” in 2020 Int. Conf. on Computer Science and Management Technology (ICCSMT), pp. 128-132, 2020.

[5] A. Nielsen, Practical Time Series Analysis, O'Reilly Media, Inc., CA, 2019.

[6] N. Marz and J. Warren, Big Data: principles and best practices of scalable real-time systems, Manning Publications, 2015.

[7] A. Batyuk and V. Voityshyn, “Streaming Process Discovery for Lambda Architecture-Based Process Monitoring Platform,” IEEE 13th Int. Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT), pp. 298-301, 2018.

[8] M. Kiran, P. Murphy, I. Monga, J. Dugan and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," 2015 IEEE International Conference on Big Data (Big Data), pp. 2785-2792, 2015.

[9] U. Suthakar, L. Magnoni, D. R. Smith and A. Khan, “Optimised Lambda Architecture for Monitoring Scientific Infrastructure,” in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 6, pp. 1395-1408, 2021.

[10] A. Azqueta-Alzúaz, M. Patiño-Martinez, I. Brondino and R. Jimenez-Peris, “Massive Data Load on Distributed Database Systems over HBase,” 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 776-779, 2017.

[11] D. McCreary and A. Kelly, Making sense of NoSQL, Manning Publications, 2014.

[12] B. Jose and S. Abraham, “Performance analysis of NoSQL and relational databases with MongoDB and MySQL”, Materials Today: Proceedings, vol. 24(3), pp. 2036-2043, 2020.

[13] F. Cerezo, C. E. Cuesta, J. C. Moreno-Herranz and B. Vela, "Deconstructing the Lambda Architecture: An Experience Report," 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 196-201, 2019.

[14] D. Maeda and S. Gaur, “Lambda architecture for robust condition based maintenance with simulated failure modes,” IEEE/ACM Symposium on Edge Computing (SEC), pp. 152-154, 2020.

[15] R. Alghamdi and M. Bellaiche, “A Deep Intrusion Detection System in Lambda Architecture Based on Edge Cloud Computing for IoT,” 4th International Conference on Artificial Intelligence and Big Data (ICAIBD), pp. 561-566, 2021.

[16] F. de Moura Rezende dos Santos and M. Holanda, “Performance Analysis of Financial Institution Operations in a NoSQL Columnar Database,” 15th Iberian Conference on Information Systems and Technologies (CISTI), pp. 1-6, 2020.

[17] C. Bao and M. Cao, “Query Optimization of Massive Social Network Data Based on HBase,” IEEE 4th International Conference on Big Data Analytics (ICBDA), pp. 94-97, 2019.

[18] F. Ye, S. Zhu, Y. Lou, Z. Liu, Y. Chen and Q. Huang, "Research on Index Mechanism of HBase Based on Coprocessor for Sensor Data," IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), pp. 598-603, 2019.

[19] P. Zhengjun and Z. Lianfen, "Application and research of massive big data storage system based on HBase," IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), pp. 219-223, 2018.

[20] T. Dunning and E. Friedman, Time Series Databases: New Ways to Store and Access Data, O’Reilly Media, Inc., CA, 2015.

[21] P. Wang, F. Xu, M. Ma and L. Duan, "Efficient Spatial Big Data Storage and Query in HBase," 2019 IEEE International Conference on Smart Cloud (SmartCloud), pp. 149-155, 2019.

[22] H. Ochiai, H. Ikegami, Y. Teranishi and H. Esaki, "Facility Information Management on HBase: Large-Scale Storage for Time-Series Data," 2014 IEEE 38th International Computer Software and Applications Conference Workshops, pp. 306-311, 2014.

[23] G. Liu, W. Zhu, C. Saunders, F. Gao and Y. Yub, “Real-Time Complex Event Processing and Analytics for Smart Grid”, Procedia Computer Science, vol. 61, pp. 113-119, 2015.

[24] K. Mishra, S. Basu, U. Maulik, “Graft: A graph based time series data mining framework,” Engineering Applications of Artificial Intelligence, vol. 110, 2022.