NoSQL Database Performance Tuning for IoT Data
Cassandra Case Study

Lucas B. Dias[1,2], Maristela Holanda[2], Ruben C. Huacarpuma[3] and Rafael T. de Sousa Jr[3]

[1]*Applied Economics Research Institute (Ipea), Brasília, Brazil*

[2]*Department of Computer Science, University of Brasília, Brasília, Brazil*

[3]*Cybersecurity INCT Unit 6, LATITUDE, Dept. of Eletrical Engineering, University of Brasília, Brasília, Brazil*

Keywords: Database, NoSQL, Internet of Things Data Management, Performance Tuning, Compaction Strategy.

Abstract: Data provided by Internet of Things (IoT) are time series and have some specific characteristics that must be considered with regard to storage and management. IoT data is very likely to be stored in NoSQL system databases where there are some particular engine and compaction strategies to manage time series data. In this article, two of these strategies found in the open source Cassandra database system are described, analyzed and compared. The configuration of these strategies is not trivial and may be very time consuming. To provide indicators, the strategy with the best time performance had its main parameter tested along 14 different values and results are shown, related to both response time and storage space needed. The results may help users to configure their IoT NoSQL databases in an efficient setup, may help designers to improve database compaction strategies or encourage the community to set new default values for the compaction strategies.

1 INTRODUCTION

IoT data may be classified as Big Data, as they present big Volume, Velocity and Variety, if the number of devices present in an IoT environment is big and increasing (Ramaswamy et al., 2013; Zhang et al., 2013; Zaslavsky et al., 2013). NoSQL Databases are more suitable than relational databases for storing the IoT Data (Vongsingthong, Suwimon and Smanchat, Sucha, 2015; Ma et al., 2013; Zhu, 2015), as they can handle Big Data and some of them have specific engines for time series data.

IoT Data is one specific case of time series data, as they are measurements taken along the time, in a sequential fashion (Ramesh et al., 2016). There are some characteristics of IoT data that must be treated by the database system to improve the efficiency of storage in data writing, data reading and space needed. The following represent these specificities (Waddington and Lin, 2016; Abu-Elkheir et al., 2013):

- *Massive Data* – The number of devices connected to the Internet has been increasing rapidly in the recent years and is expected to achieve 24 billion by 2020 (Gubbi et al., 2013). The amount of data is proportional to the number of devices, hence it may achieve huge volumes.
- *Data is ordered* – Sensor generated data very often comes with a timestamp, which can be used by the system (usually a Middleware) to insert data in the correct order (Cruz Huacarpuma et al., 2017). The device data is usually inserted in the database in the order it was observed in the environment.
- *Time based data retrieval* – Users typically want to query data by a time-related key.
- *Data rarely changes* – For the most part, IoT data does not change. Nonetheless, data can be overwritten or deleted if some severe accuracy error is perceived in some device. Consistency is not a critical issue, because it is very unlikely a user will get old data instead of the updated one.
- *IoT data expires* – The life cycle of the IoT Data depends on the application, but generally, if it intends to provide monitoring, old data is not useful and can be deleted or aggregated because it will seldom be needed. Even in more persistent applications, new data is more frequently queried than old data.

NoSQL databases fit very well some of those characteristics. There are even some specific time series databases. However, some general purpose NoSQL databases can also be very appropriate to store IoT Data. This paper presents some performance tuning

Dias, L., Holanda, M., Huacarpuma, R. and de Sousa Jr, R.
NoSQL Database Performance Tuning for IoT Data.
DOI: 10.5220/0006782702770284
In *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security (IoTBDS 2018)*, pages 277-284
ISBN: 978-989-758-296-7

configurations to make NoSQL databases more efficient in terms of response time and space used. More specifically, the database compaction parameters will be tuned. A study case is stated using the Cassandra NoSQL database.

The remainder of this paper is organized as follows: Section 2 explains the process of database compaction, so that the reader can understand the problem and the impact of the parameters in the performance; Section 3 presents related work; Section 4 describes the test environment and Section 5 presents the results, while Section 6 presents our conclusion, as well as ongoing and future work.

2 NoSQL DATA COMPACTION

Google BigTable (Chang et al., 2008) presented a new distributed, high performance database solution for a massive data income rate, with certain fault tolerance. Two years later, Lakshman and Malik from Facebook presented another database system, inspired both in BigTable and Dynamo (DeCandia et al., 2007), called Cassandra. It does not support a relational data model with relational integrity, but provides a flexible column family data model. “Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency“ (Lakshman and Malik, 2010).

The NoSQL systems derived from BigTable (Cassandra, HBase, RocksDB etc.) use a storage structure where recently inserted data stays in memory, in structures called *Memtables*, which are, essentially, string arrays (Chang et al., 2008). Specifically in Cassandra, the data is flushed to disk when (i) Memtables reaches its maximum size, (ii) Memtables reach their maximum age or (iii) when the user commands. The flush operation involves saving the data in the disk, in a structure called Sorted-String Table (*SSTable*). The SSTables are immutable and this provides good write-intensive performance (Singh, 2015).

If eventually a cell from a SSTable column family gets updated, the new value is stored in another Memtable, live in memory. If this Memtable undergoes a flush, then there will be two SSTables with data from the same cell. When there is a read operation, the system must read data in those two SSTables, seek for this cell in the Memtables, merge every cell value implicitly and exhibit the most recent value to the user. If one cell has different values along several SSTables, this process will be very costly (DataStax, 2017).

Deleting cells in Cassandra does not free up space, initially. When a delete operation occurs, the cell is marked as a *tombstone*, a process similar to a soft deletion in relational databases. Each tombstone receives an expiration time, called a ‘grace period’, which is set in the column-family specification. Tombstones are not retrieved to the user but they stay on disk until their grace period expires and the SSTables are compacted.

Compaction, in this context, is not related to compression or data compaction algorithms. It is the term used to define the operation that merges two or more SSTables, unites the column family cells that have different values and evicts the tombstone expired data, freeing up storage space (Chang et al., 2008; Singh, 2015). Since SSTables are already ordered, this operation is not CPU bound, but it is I/O bound (Ghosh et al., 2015). Typically, storage space will have a peak, needed to allocate space to the new SSTable, followed by a reduction. This operation is illustrated in Figure 1.

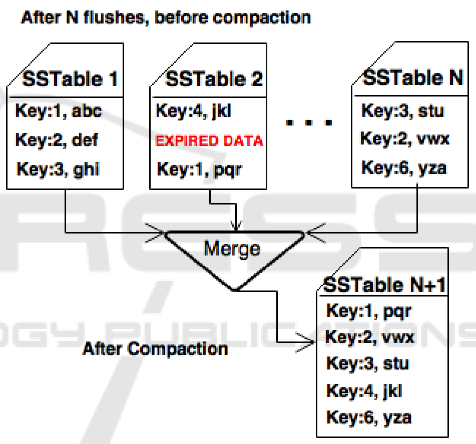


Figure 1: Cassandra database compaction.

If Compaction is not made, read operations will have to seek many SSTables to look for the data, on other hand, the compaction operation is I/O intensive and therefore impacts on the system performance, especially if performed frequently. Hence, an optimal amount of compaction operations is necessary where these compaction operations yield better read and write response time. In this paper we investigate how to achieve a near-optimal point, in terms of response time and disk usage in an IoT environment.

The main objectives of compaction operations are: (i) regain disk space, purging the expired and deleted data; (ii) to store data in a contiguous fashion, keeping data from the same column family and partition in the same SSTable, or in as few as possible; (iii) to optimize server reads, assuring they will have to access as few SSTables as possible, to faster response (DataStax, 2017).

There are two compaction strategies in Cassandra to handle time series: Date Tiered Compaction Strategy (DTCS) and Time Window Compaction Strategy (TWCS). Both rely on grouping data by its age (i.e. difference between insertion and now) and define time windows. One capital difference is that in case of TWCS, as long as the time window has not been reached, it uses another method, the Size Tiered Compaction Strategy (STCS) to compact SSTables, if needed. In other hand, DTCS does not compact if the data threshold has not been reached. Less compaction results in Cassandra using less effort in organizing data, however, if a queried cell lays in many SSTables, read operations will have to scan more SSTables in disk while seeking for data.

3 RELATED WORK

Studies have been conducted about NoSQL Performance tuning, however, very few have broadened the configuration of database compaction strategies.

Lu and Xiaohui studied the DTCS, present in the Cassandra system and, based on the simulated results, concluded that Cassandra is a remarkable database and is very suitable for storing time series, such as the IoT Data (Lu and Xiaohui, 2016).

(Kona, 2016) and (Ravu, 2016) simulated the Cassandra behavior in a high intensive workload. The former concluded that the DTCS is not the best strategy in a write-intensive workload (90% write operations and 10% read operations) and the latter concluded that in a balanced workload (50% read and 50% write) the DTCS was the most efficient strategy to date. However, this paper differs from those two studies in that the data in the present work is confined to IoT Data.

The Cassandra performance was also analyzed by (Sathvik, 2016), by changing parameters and the configuration of Memtables and Key-Cache, that is a special table that saves pointers to the rows in each SSTable. However, it differs from our study because compaction strategy parameters were not changed. Indeed, they conclude recommending that future works consider altering the parameters of compaction strategies to further optimize performance.

Unlike these studies, our paper presents a performance tuning approach to compare the novel TWCS to the preexistent DTCS. Furthermore we change the TWCS main parameters in order to evaluate response time, throughput and disk usage.

4 TEST ENVIRONMENT AND METRICS

This section presents in Subsection 4.1 the computational environment where experiments were executed; Subsection 4.2 presents the IoT environment simulation; Subsection 4.3 show the execution of two test cases and finally Subsection 4.4 explains what metrics were chosen and why.

4.1 Test Bed

All tests were performed in a cluster with 10 nodes of equal capacity. Each node has one Intel Xeon® core; 3.2GB of RAM memory; 7200 rpm spinning disks with 50GB of space; Linux installed, with Ubuntu distribution. Cassandra version 3.11.1 is installed in all nodes. They are all virtual machines and the host server is not exclusive for these tests. We tried to minimize the impact of other services on tests by repeating the stress operations on different weekdays and different schedules.

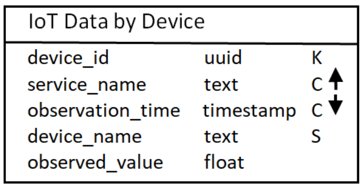
The main tool used to generate and simulate the operations is the Cassandra Stress Tool (Apache, 2016), maintained by the Cassandra Community. Every stress operation was run on a machine outside the cluster. The stress process had 24 threads that generated, inserted and queried data in parallel.

4.2 IoT Environment

The IoT scenario is typically a write-intensive application. The test operations were distributed as 90% of write operations and 10% of read queries, distributed as: 4% for a query that selects all measures from one device, which we call *devdata*, 3% for a query that retrieves one value, given the device and observation time (named *onerow*) and a query that retrieves the average value of one device, given its Id (named *avgdata*).

In all tests, a one-hour Time to Live (TTL) parameter was applied to the column family. This parameter sets the data expiration time, which added to the *grace period* parameter (that was fixed in half an hour) states that data may get purged from the table after 90 minutes.

The column family schema was modeled with the partition key (K) being the device identifier and each device may provide different services (in tests, a fixed number of five services). The service name, along the observation value compose the clustering key (C). The model is presented in the Chebotko notation, a special model for column family databases (Chebotko et al., 2015), in Figure 2.



IoT Data by Device		
device_id	uuid	K
service_name	text	C ↑
observation_time	timestamp	C ↓
device_name	text	S
observed_value	float	

Figure 2: Column family schema.

The observation time is stored in a descending order, because more recent data is more often queried and retrieved. The device name does not repeat at every observation, so it is defined as a static column (S), what in Cassandra makes it be saved beside the partition key and not repeatedly in every row.

4.3 Test Cases

The first test case modeled was the comparison between DTCS and TWCS. The former was marked as deprecated although it is still available in Cassandra, whilst the latter is the natural substitute for DTCS. Although the community has made tests to perform this substitution (Jirsa, 2016), we could not find any published paper that compared TWCS with DTCS.

The second test case modeled was the adjustment of the main parameter in TWCS to achieve a near-optimal performance result. The parameter is *compaction_window_size*, which, in short, tells the database system the age at which the SSTables become available to compact and when not to compact them anymore.

In both test cases, the volume of data was the same. The number of rows inserted *RI* is given by Equation 1.

$$RI = OP * IR * TS * OV \qquad (1)$$

The stress tool created 2,000,000 operations *OP* (90% writes and 10% reads), which means that the insert ratio (*IR*) is 0.9. In each operation, a device received 5 time series (*TS*), one for each service name. Each time series gets 60 observation values (*OV*).

$$RI = 2{,}000{,}000 * 0.9 * 5 * 60 = 540{,}000{,}000 \qquad (2)$$

These values resulted in 540 million rows inserted throughout the cluster, as shown in Equation 2. This configuration was set so that there would be enough time for the data to expire. With TTL (60 minutes) plus the grace period (30 minutes) set at 90 minutes, each stress operation lasted more than 120 minutes, so the 540 million rows never persisted at the same time. After 90 minutes, as new data was coming, old data was being purged.

4.4 Metrics

To evaluate the performance, the main metrics chosen were:

- **throughput** – the amount of operations per second, in the case of the read queries, and the amount of rows per second, in the case of insert operations. Typically, as the scenarios involved the same amount of data, the throughput impacted directly on the total simulation time.
- **latency** – the time needed for the system database to respond to a request. It starts counting when the request is received and stops when the message is delivered to the client. Both write and read latency were evaluated.
- **disk space** – the total amount of disk space used to store the data.

The chosen metrics represent the user point-of-view of the system, and although the CPU, Memory and I/O used have been analyzed, they are not presented here because they are reflected in the chosen metrics. For instance, an I/O bottleneck will impact the throughput and latency. With some exceptions, the same happens to memory-bound and CPU-bound operations.

The metrics were provided by Cassandra either by the Stress Tool logs or by the Core Metrics library, which runs inside Cassandra, in a configurable way. They both provide column family specific measures, which aids test precision because it isolates the measures from other column families or possible OS requests.

5 RESULTS

The tests were made according to the two test cases presented in Subsection 4.3. The results collected and analyzed are presented in the following subsections.

5.1 DTCS Versus TWCS

In order to test the two compaction strategies, the same data was inserted and queried in two column families, one at a time. The schemas of the two column families are the same as Figure 2, except for the compaction strategy. The column family IoT_DTCS was created with DTCS with its default parameters except the most relevant: *base_time_seconds*, which was set to 10 minutes. In other hand, the column family IoT_TWCS was created with TWCS default parameters, except the *compaction_window_size*, which was set also to 10 minutes.

The first performance indicator shown is the elapsed time needed to manage the data. Overall, six stress operations were made in each compaction strategy and the standard deviation was low, which is why we report the mean. The time results are presented in Table 1, with standard deviation (STD) in percent values of the mean and Speedup being the DTCS time divided by TWCS time.

Table 1: Mean time elapsed for the 600M operations using DTCS and TWCS.

Strategy	Mean Time	STD (%)	Speedup
DTCS	02h41m48s	0.71%	N/A
TWCS	02h14m57s	0.18%	1.20

The TWCS has stored the same amount of data 20% faster than the DTCS. This may also be seen in the throughput of the insertion operation (see Figure 3) expressed in inserted rows per second. The mean throughput standard deviation among the six simulations was 6.5% overall – not in every 30 second interval, but throughout the whole stress process.

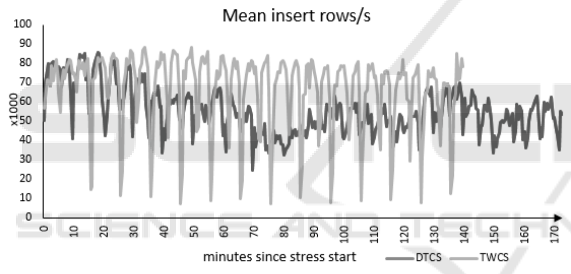


Figure 3: Both scenarios were executed in different times but displayed together along the same timeline. The y axis represents thousands of inserted rows per second while the x axis stands for the execution time, in minutes.

Moreover, the read operations, that are more susceptible to the compaction operations are, overall, better using the TWCS than DTCS. To illustrate, we show in Table 2 the results of one stress operation of TWCS and the comparison over one stress operation of DTCS. The ones chosen among the six stress operations were those with an execution time closer to the mean time. The rightmost column is the DTCS latency divided by TWCS latency.

Similarly, as in elapsed time, TWCS was shown to have a faster mean latency at 22%. The fact that the median time was slightly faster in DTCS does not represent a relevant advantage, because the TWCS mean and the other two percentiles are lower. The percentile 0.99 shows that Cassandra can respond to 99% of the requests in under 677.4 milliseconds using TWCS, while the same percentile in DTCS was responded up to 1618.9 milliseconds. The upper percentiles show that this database system respond to the majority of the users in a timely fashion. The maximum latency shows high numbers, but consideration must be made to the fact that 600 million operations were performed, and that it would be very costly to guarantee low-latency responses to all requests, using regular commodity hardware (DeCandia et al., 2007).

Table 2: Latency (ms) statistics with percentiles.

TWCS	Read	Insert	Total	DT/TW
mean	181.8	88.6	97.9	1.22
median	62.0	40.9	41.5	0.93
0.95	595.9	231.6	260.0	1.40
0.99	1683.7	426.2	677.4	2.39
0.999	8018.1	4945.1	5326.8	1.09
max	27263	46271	46271	0.52

However, DTCS has showed an advantage over TWCS in disk space used. In Figure 4, all nodes had their space summed and an average has been applied to all six stress operations. It is important to remember that data becomes expired after 90 minutes. Although new rows are continuously being inserted, the volume becomes stable after 110 minutes because the compaction starts to purge expired data.

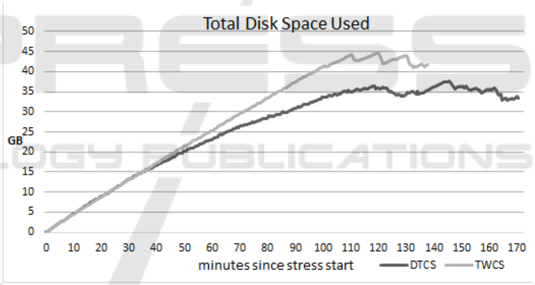


Figure 4: Both scenarios were executed in different times but displayed together along the same timeline.

Figure 4 shows that DTCS uses less disk space than TWCS. Throughout the whole cluster, the maximum disk space used was 18.4% higher in TWCS than in DTCS. Considering all 30 seconds intervals, the mean disk space used was 5.5% higher in TWCS. Tests were run with the compression disabled, to enhance performance. Further tests must be carried out with the default compression algorithm enabled to observe if the space advantage persists in the same level or is decreased.

5.2 Compaction Window Size

TWCS was shown to perform better than DTCS in terms of response time. Moreover, the latter has been marked as deprecated by the Cassandra community, which justifies our choice of furthering studies with

TWCS.

The main parameter of this compaction strategy is *compaction window size*, which sets the time interval for contemporary data in the same SSTable. In these tests we tried to find a near-optimal parameter that would give the best performance results to our IoT test environment. Fourteen test cases were executed, changing only the *compaction window size* parameter, receiving values from 1 to 240 minutes.

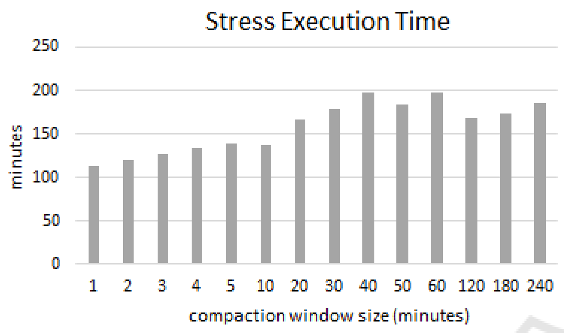


Figure 5: The y axis shows the execution time and x axis the time window size, both in minutes. The fastest execution was with 1 minute window.

Figure 5 shows that the most time-efficient configuration was the 1 minute window size. As the windows increase, the execution time increases and it does not decrease to achieve the lowest values of 1 minute, since the 240 minutes is greater than the execution time, which means that values greater than 240 would not have any difference in the execution of these tests operations.

The total disk space used throughout the 10 cluster nodes is presented in Figure 6. The compaction sizes are shown from 1 to 60 minutes. The space used does not decrease with compaction window size more than 60 minutes. As in the DTCS versus TWCS case, the fastest configuration is the one that uses more space. The 1 minute window size used more space, both in average and peak.

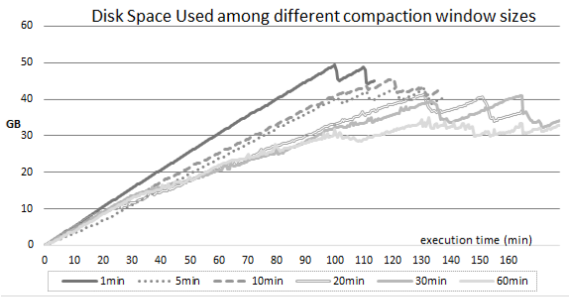


Figure 6: The 1 minute window uses more space.

The more space needed is due to the fact that, in the 1 minute compaction window size, SSTables are grouped in a small window and therefore are not compacted anymore until it expires. With larger window sizes, the SSTables continue to be compacted until the time window size has not been reached. Thus, with a big window size, there will be more compaction operations using STCS, which is the default compaction strategy as long as the time window is not achieved.

Another interesting aspect that must be analyzed is the relation between the read latency and write latency. As showed in Section 2, the read operation depends on the number of SSTables. Without expired data, the lower the compaction window size, the more SSTables in a long term. However, the effect of augmenting read latency is not direct, because, for each SSTable, there is a bloom filter that tells if some partition is not present in that file. Therefore, it avoids scanning most of the SSTables, which reduces the impact of a bigger number of SSTables in the read operation.

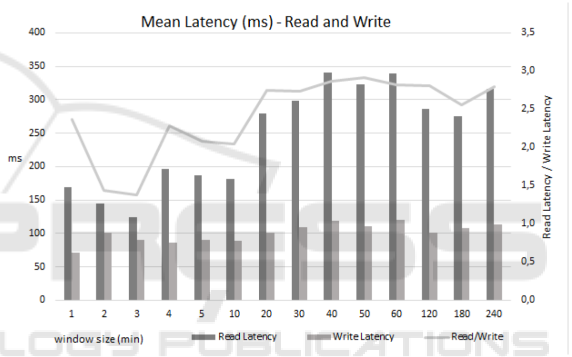


Figure 7: The read and write latency values and ratios for different compaction window sizes.

Notably, the lowest read/write ratio is in the 3 minute compaction window size. In this test scenario, the proportion between the amount of insert and read operations are 9 to 1. Thus the insert operation is far more relevant for the total execution time. Nevertheless, if more read operations occur, most likely the optimal compaction window size will be closer to 3 minutes than to 1 minute.

All experimental results can be found at https://github.com/lucasbenevides/iotbds.

6 CONCLUSION AND FUTURE WORK

Data derived from IoT environments may achieve high throughput levels and may get bulky as time goes by. In order to optimize the computational resources in the IoT data storing task, this work has tuned a NoSQL database system, specifically the compaction strategies, and analyzed its results.

In many computational problems, there is a trade-off between storage space and time. In both test cases performed in this work this trade-off remains. In the first test case, Cassandra's TWCS performed 20% faster than DTCS, but the former used, in peak, 18.4% more disk space. In addition, the configuration of TWCS is simpler than the one to DTCS, what corroborates with the community decision to deprecate DTCS and continue developing TWCS. Users should choose DTCS only if the disk space is limited.

The second test case evaluated different configurations of the TWCS parameter *compaction_window_size* in relation to a scenario where data becomes expired after 90 minutes. The 1 minute value was near-optimal in terms of elapsed time, latency and throughput. Nevertheless, the space trade-off continued to be an issue. The fastest test case was shown to be the one that used more space, in peak 40.8%, in comparison with the 60 minute window size, which lasted 72.6% more.

The 1 minute was the lowest value accepted in the TWCS configuration. As it has shown to be the near optimal, we recommend that other values lower than 1 minute be accepted by Cassandra.

The ratio between the expiration time (TTL plus grace period) and compaction window size shall be tested with different values. If a near-optimal relation can be confirmed, users may have a "golden rule" to configure their column-families.

Further work is needed mainly in the second test case. Tests must be performed in a larger test bed to evaluate if the results are consistent. Likewise, other read/write ratio operations must be tested. Although 600 million rows is already a considerable size, tests should be made with larger data sets – preferably with real data sets instead of generated data as in the present paper.

This study will be useful within a broader research goal the authors aim to achieve: the creation of an auto tuning component for compaction parameters, initially within TWCS. These results will help to create rules that lead to an autonomous performance tuning agent, which intends to eliminate the time and effort users spend tuning the database compaction strategy parameters.

ACKNOWLEDGEMENTS

This research work has the support of the Brazilian research and innovation Agencies CAPES (Grant 23038.007604/2014-69 FORTE–Tempestive Forensics Project), CNPq (Grant 465741/2014-2 – Cybersecurity INCT) and FAPDF (Grants 0193.001366/2016 UIoT–Universal Internet of Things and 0193.001365/2016–Secure Software Defined Data Center (SSDDC)), as well as the Ministry of Planning, Development and Management (Grants 005/2016 DIPLA–Planning and Management Directorate and 11/2016 SEST–Secretariat of State-owned Federal Companies) and the DPGU–Brazilian Union Public Defender (Grant 066/2016).

REFERENCES

Abu-Elkheir, M., Hayajneh, M., and Ali, N. (2013). Data Management for the Internet of Things: Design Primitives and Solution. *Sensors*, 13(11):15582–15612.

Apache, S. F. (2016). The Cassandra-stress tool. http://cassandra.apache.org/doc/latest/tools/cassandra_stress.html. Last accessed 22 January 2018.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

Chebotko, A., Kashlev, A., and Lu, S. (2015). A Big Data Modeling Methodology for Apache Cassandra. pages 238–245. IEEE.

Cruz Huacarpuma, R., de Sousa Junior, R. T., de Holanda, M. T., de Oliveira Albuquerque, R., García Villalba, L. J., and Kim, T.-H. (2017). Distributed Data Service for Data Management in Internet of Things Middleware. *Sensors*, 17(5):977.

DataStax (2017). Datastax docs : The write path to compaction. https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml_write_path_c.html. Last accessed 22 January 2018.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and 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, pages 205–220, New York, NY, USA. ACM.

Ghosh, M., Gupta, I., Gupta, S., and Kumar, N. (2015). Fast Compaction Algorithms for NoSQL Databases. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 452–461.

Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7).

Jirsa, J. (2016). TWCS experiments and improvement proposals - ASF JIRA [CASSANDRA-10195]. https://issues.apache.org/jira/browse/CASSANDRA-10195.

Kona, S. (2016). *Compactions in Apache Cassandra : Performance Analysis of Compaction Strategies in Apache Cassandra*. Master's thesis, Blekinge Institute of Technology, Karlskrona, Sweden.

Lakshman, A. and Malik, P. (2010). Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.

Lu, B. and Xiaohui, Y. (2016). Research on Cassandra data compaction strategies for time-series data. *Journal of Computers*, 11(6):504–513.

Ma, M., Wang, P., and Chu, C. H. (2013). Data Management for Internet of Things: Challenges, Approaches and Opportunities. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 1144–1151.

Ramaswamy, L., Lawson, V., and Gogineni, S. V. (2013). Towards a quality-centric big data architecture for federated sensor services. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 86–93. IEEE.

Ramesh, D., Sinha, A., and Singh, S. (2016). Data modelling for discrete time series data using Cassandra and MongoDB. In *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*, pages 598–601.

Ravu, V. S. S. J. S. (2016). *Compaction Strategies in Apache Cassandra : Analysis of Default Cassandra stress model*. Master's thesis, Blekinge Institute of Technology, Karlskrona, Sweden.

Sathvik, K. (2016). *Performance Tuning of Big Data Platform : Cassandra Case Study*. PhD thesis, Blekinge Institute of Technology, Faculty of Computing, Department of Communication Systems, Karlskrona, Sweden.

Singh, K. (2015). Survey of NoSQL Database Engines for Big Data. Master's thesis, Aalto University. School of Science.

Vongsingthong, Suwimon and Smanchat, Sucha (2015). A Review of Data Management in Internet of Things. *KKU Research Journal*, pages 215–240.

Waddington, D. G. and Lin, C. (2016). A Fast Lightweight Time-Series Store for IoT Data. *CoRR - Computing Research Repository*. arXiv: 1605.01435.

Zaslavsky, A., Perera, C., and Georgakopoulos, D. (2013). Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*.

Zhang, J., Iannucci, B., Hennessy, M., Gopal, K., Xiao, S., Kumar, S., Pfeffer, D., Aljedia, B., Ren, Y., Griss, M., Rosenberg, S., Cao, J., and Rowe, A. (2013). Sensor Data as a Service – A Federated Platform for Mobile Data-centric Service Development and Sharing. In *2013 IEEE International Conference on Services Computing*, pages 446–453.

Zhu, S. (2015). *Creating a NoSQL database for the Internet of Things : Creating a key-value store on the SensibleThings platform*. PhD thesis, Mid Sweden University, Faculty of Science, Technology and Media, Department of Information and Communication systems, Sundsvall, Sweden.