IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 20, Issue 6, Ver. III (Nov - Dec 2018), PP 46-52
www.iosrjournals.org

Managing IoT data using relational schema and JSON fields, a comparative study

Christodoulos Asiminidis[1], George Kokkonis[2], Sotirios Kontogiannis[1]
[1]*Laboratory team of Distributded Microcomoputer systems, Department of Mathematics, University of Ioannina, Ioannina, Greece*
[2]*Department of Business Administration, T.E.I of Western Macedonia, Grevena, Greece*
Corresponding Author: Christodoulos Asiminidis

Abstract: *Data transmitted from sensors and actuators as part of the Internet of Things (IoT) infrastructure are stored either in database tables following relational schema and normalization forms or in schema less collections using JSON string or binary formulation. As data content in such repositories radically increases, the selection and use of the appropriate storage types are essential in terms of performance and robustness. Furthermore, taking into account the amount of database capacity and processing needed, as well as the exponential increase and use of IoT devices, storage and retrieval of sensory data are the main bottlenecks and set the boundary requirements for IoT services functionalities.*
This paper tries to identify the performance characteristics that derive from data operations over IoT big datasets that are stored as records in relational schema tables or documents in tables containing JSON fields. Trying to pose an answer to the question which one performs better than the other the PostgreSQL open source relational database has been selected and examined for insert, select/find and aggregation function queries. The comparative study results are presented and thoroughly discussed.

Date of Submission: 20-12-2018 Date of acceptance: 06-01-2019

I. Introduction

Databases are used in order to satisfy data storage requirements. After their inception in the 1960's several types have been developed, each one using its own supported data representation. Initially performing as linked list navigational databases followed by the relational databases schemas and supported fields with joins, triggers, views, functions and stored procedures and afterwards object-oriented capabilities and other specific type fields for the process of storing images, videos or coordinates and vector objects (GIS capabilities).

In the late 2000s NoSQL schema less data stores appeared for the purpose of data storage of entities that their attributes timely change. The main representatives of these databases are MongoDB, Cassandra, Hyper table, HBase/Hadoop and CouchDB [15] emerged and became a popular trend [4]. Most commonly used database systems today use the relational model [7], which includes SQL as its query language. Nevertheless, over the last years NoSQL database solutions are becoming more prominent as massive amounts of rapidly growing data of non specific formulation or entities of non atomically fields and fields with partial and transitive dependencies are being collected today, in the form of super-entities called collections [11, 20]. This poses the question if the relational model reached their service limits.

Relational databases use normality forms (1NF, 2NF, 3NF, BCNF, 4NF and 5NF) on the notion of entities containing fields and entity records filling up table datasets. Normalization processes include the analysis of functional dependencies between entity attributes [11]. Normalization tries to eliminate the redundancy but not at the cost of integrity so as to improve the performance of database queries. De-normalization is the inverse process of normalization, where the normalized schema is converted into a schema which has redundant information. The performance is improved by using redundancy; however, in many cases keeping the data integrity intact may lead to redundant data inconsistencies [21].

De-normalization processes can also be defined as the methods of storing superior normal form joined relations as a base relation, which is kept normalizes only in a lower normal form. Such processes try to reduce the number of database tables, and table joins since joins can slow down the query process. There are various de-normalization techniques such as: Storing derivable values, pre-joining tables, hard-coded values and keeping details with master, etc. De-normalized schemas can greatly improve performance under extreme read-loads but the updates and inserts become programmable complex, since they require data duplication and hence has to be updated/inserted in more than one place [16].

NoSQL databases started gaining popularity in the last decade, when companies began investing into distributed databases [20]. For this purpose the category of NoSQL databases grew and included many subtypes

each better suited to specific datasets than others. Using subtypes containing attributes and super types linked to subtypes, NoSQL databases provide a schema flexibility that can be useful for data records of arbitrary fields offering also easy programming discrepancies away from the precariousness of relational databases query preparations and strict type checks. The notion of "documents" is the central concept here with documents being the equivalent of records in relational databases and collections being similar to tables.

The most commonly used document store database is MongoDB [7, 14], used by many IoT services, since IoT services use sensors that acquire data objects of variable and time vary schema records. However, the programming conveniences offered, there is a processing effort tradeoff regarding the transactions performed in a non relational dataset [13]. This continuing battle between relational databases and NoSQL datasets lead to the incorporation of the NoSQL JSON formulation and corresponding documents query functionality into particular relational database fields called JSON, JSONP (JSON with padding) and JSONB [9] accordingly. More particularly open source PostgreSQL database as a pioneer int the area of object oriented databases has already implemented the JSON and JSONB fields in its RDBMS engine, without the complexity of having two separate databases for SQL and NoSQL datasets [18]**.**

In this paper the schema full and schema less representations of a big dataset containing IoT sensory measurements are put to test and examined in terms of performance over bulk data inserts, query data select aggregations and stored procedures internally implemented in the PostgreSQL database as aggregation functions. In section II an outline of the performance characteristics of relational PostgreSQL over NoSQL MongoDB are presented. In section III the authors present their performance evaluation scenarios and results followed by section IV, the evaluation summary conclusions.

II. Related Work

The main aspect of relational databases which guarantees the reliability of transactions is their adherence to the ACID properties: Atomicity, Consistency, Isolation, and Durability [2, 3]. That is, preserving data integrity, stability and availability. An important difference between relational databases and NoSQL databases is that NoSQL databases do not fully guarantee ACID properties. Their lack of ACID guarantees is due to their deployment architecture which typically involves having multiple nodes in order to achieve horizontal scalability and recovery in case of failover. This deployment, which is also referred to as replication, creates consistency issues to synchronization which can result in a secondary node becoming primary but not have an up-to-date content. NoSQL databases, apart from using an Application Programming Interface (API) or query language other than SQL to access and modify data may also use the Map reduce method which is important for performing a specific function on clustered dataset and retrieving only the queried result[8]**.**

Relational databases mainly include only schema full tables and fields, with the exception of PostgreSQL. PostgreSQL supports two additional fields called JSON and JSONB [18]. These two data types JSON and JSONB, as defined by the PostgreSQL documentation, difference is that JSON field stores an exact copy of the JSON input text, whereas JSONB stores data in a decomposed byte-form. This form is slower for data input and storage size, since it requires more space than the JSON field for a specific record, but it is faster to process, it supports indexing and joins which in turn can lead to simpler designs by replacing the JSON fields' necessity of an entity attribute value.

In order to measure the databases performance and scalability, uniform metrics are required. The most important metric for the application layer protocol that performs database transactions, is the time required for completing a set of prepared queries, which translates to the time required for the database service to complete a transaction (series of prepared SQL queries). Then the average query execution time is derived from the average number of queries per transaction and the average transactions execution time [2, 3 19]. For the process of transaction consistency estimation authors propose the query jitter metric (Tj) which is calculated using Equation 1 and expresses database queries variation over time:

$$Tj = TDB_{init} + \frac{(dT_1 - dT_2)}{\Sigma R_1^{insert\ |update\ |..} - \Sigma R_2^{insert\ |update\ |..}} \vee \text{ (ms)} \qquad \textbf{(1)}$$

where the sums ΣR1, ΣR2 are the number of records returned from queries 1 and 2 accordingly and dT1, dT2 is the time required completing the queries. TDB_init is the average initialization and setup time for each query which is assumed as a constant coefficient parameter for each query type accordingly and is calculated experimentally using a zero result query time estimate [6, 10]. The average metric values of all the benchmarks during a single run of the benchmarking harness are calculated in order to have a better overall idea of how the database behaves, as a single unity benchmark may deviate due to external factors such as an operating system utilization of the server CPU or performing burst I/O [1].

Before comparing the performance of embedded PostgreSQL fields, authors bibliographically examined the performance of PostgreSQL in comparison to MySQL and then collate the results with the mostly

used NoSQL MongoDB database performance evaluation experiments [1, 5, 6, 8, 11, 12, 17]. Focusing on the cross comparison results of PostgreSQL and MongoDB only, MongoDB is faster than PostgreSQL for insert queries presents and presents similar performance for Select-find queries, which deteriorates when the number of records increases in favor of MongoDB. The records/documents update performance between PostgreSQL and MongoDB showed that MongoDB is also more efficient as the number of records increases. Table 1 below summarizes the overall performance of PostgreSQL and MongoDB in insert, select-find and update experiments.

Table 1: Performance summary between PostgreSQL and MongoDB

Queries	Small number of records (Less than 10.000)	Big number of records (Less than 100.000)
Insert	MongoDB performs similarly to PostgreSQL	MongoDB is 5-10% faster than PostgreSQL
Select-find	MongoDB is 5% slower than PostgreSQL.	MongoDB is 15% slower than PostgreSQL.
Update	MongoDB performs similarly to PostgreSQL	MongoDB is 9% slower than PostgreSQL

In the following section the authors' performance evaluation scenarios performed in the PostgreSQL, using a big IoT data set are presented in detail.

III. Performance Evaluation scenarios and results

Author's experimental scenarios include performance measurements of PostgreSQL relational database between relational schema and JSON fields using IoT data. For the purpose of this study, authors used the 10.5 version of PostgreSQL, a server of Intel Core 2 CPU which runs at 2 GHz with 4GB RAM where 1.5GB of RAM are reserved by the PostgreSQL service and a SATA hard disk of 320GB that can sustain 78MB/s of buffered reads, 1Gb/s cached reads. The system which has been used is Ubuntu 18.04.To minimize network delays and most importantly to increase jitter time metric accuracy (as calculated from Equation 1), PostgreSQL queries have been performed locally (minimizing network jitter) in the experimental database server using Python scripts.

The dataset used included IoT data in JSON format. Every JSON record is 168 bytes on average. The average insertion, selection, jitter and aggregation function time has been measured and averaged over the number of records. For the PostgreSQL database authors used a big IoT dataset of derived sensory data from a meteorological station that contains 1.5 year of measurements (up to 1.100.000 records). The database contains fields of sensory measurements of date time, temperature, humidity, pressure, dew point, rainfall, and wind speed and wind direction. Since PostgreSQL supports the JSON data type, authors migrated those data from a MongoDB database to a PostgreSQL table by storing the JSON sensory data to a PostgreSQL JSON field without any transformation.

Scenario 1.a. Insert queries and insert queries jitter time on JSON field table

For insert queries experimentation, PostgreSQL JSON field execution time varies according to the number of documents already inserted in the table. For small number of documents in the table, the average insertion time is 10.8ms. For medium number of documents in the table there is a slight time decrement and then increment of the query execution time, which remains almost constant for big number of existing table documents, close to 9.4ms. Jitter time, is constant as shown at Table 2, which means that there is consistency in respect to queries execution time.

Table 2: Average insert query execution and jitter time for JSON table

No. of existing documents in the Table	Avg. Insertion Time (ms)	Avg. Jitter Time(ms)
50K	10.80475725	0.000147996
100K	10.42759795	6.79E-06
200K	9.805020907	4.0793E-06
300K	9.561896896	1.50E-07
400K	9.858754606	6.45661E-07
500K	10.14608878	-0.000159567E-05
600K	9.981811373	-8.5117E-05
700K	9.232856781	-7.10482E-07
800K	9.428665195	-1.7619E-06
900K	9.667404556	8.32383E-05
1M	9.597382772	-5.40857E-05

Scenario 1.b. Insert queries and insert queries jitter time on relational table

Using the same IoT dataset of the scenario 1.a, authors migrated the data to a relational database table by transferring each JSON attribute to appropriate size fields. Then the average insertion time and jitter have been measured. The same PostgreSQL database has been used. For insert queries experimentation, PostgreSQL relational table maintains a smooth average execution time of 40ms, as presented at Table 3. Jitter time presents a non-significant variation, which is smaller than 0.002ms.

The comparison between relational table and JSON table insertion time shows that the relational data insertion time is 4 times less than the JSON field table insertion time. This signifies that the JSON field table data insertion is much more efficient than the relational table. Regarding jitter time, both relational and JSON tables jitter footprints are small, which corresponds to no significant database in-consistencies over the number of existing database records or documents.

Table 3: Average insert query execution and jitter time for the relational table

No. of existing records in the table	Avg. Insertion Time for relational schema (ms)	Avg. jitter time for relational schema (ms)
50K	40.00160742	0.002390102
100K	40.22283722	0.001426959
200K	41.01418487	-0.00013122
300K	41.32908446	-5.66954E-06
400K	40.68967547	7.51011E-07
500K	41.16252482	-9.57956E-06
600K	40.54067992	-0.000827602
700K	39.84358471	-0.000470476
800K	40.47535517	0.000650847
900K	40.09842679	1.00135E-06
1M	41.29881756	-5.40857E-05

Scenario 2.a. Select queries and select queries jitter time on JSON table

For select queries experimentation, PostgreSQL presents poor performance as the number of selected documents increases. For small number of documents JSON table did not perform also well. As the number of records increases, the average select query execution time increases dramatically from 6s for 10K returned documents up to 10s for 1M documents, as presented at Table 4.

Table 4: Average Select query execution and jitter time for JSON table.

No. of selected records	Avg. Selection time for JSON table (ms)	Avg. Selection jitter time for JSON table (ms)
10K	6157.742023	-3837.219
100K	9899.653912	2939.173937
250K	6756.292105	-722.5949759
400K	7375.653028	1083.530903
500K	8019.70005	-701.3947967
700K	8670.635939	-479.6288009
800K	9075.858831	1003.731966
900K	9473.566055	-529.6288009
1M	10168.4258	-145.3735844

Jitter varies significantly with respect to execution time. The average jitter time for all cases is 1270ms while the average select query execution time is 8300ms. This means that there is a high inconsistency, since average jitter time is close to the 15% of the average execution time. For low number of records average jitter is close to 50% of the select query execution time. For high number of records average jitter is close to 5% of the select query execution time. This signifies query possible query consistencies that are of high probability for low number of selected documents that gradually reduces as the number of returned documents increasses

Scenario 2.b. Select queries and select queries jitter time on relational table

For select queries on a relational table, for very small number of records, the average selection time is 600ms. For big number of selected records, the average select query time is 2s. Jitter time has been measured for this scenario as well. For small number of records, jitter time is close to 1.5% of the total execution time. For big number of records, jitter time is close to 0.5% of the total execution time. This means that for the relational table there are no important inconsistencies in respect to the queries execution time.

The comparison results between relational and JSON tables show that for small number of records/documents, relational table is 8 times faster than the JSON table. For big number of records relational table is 4 times faster than the JSON table. Authors notice that while the number of records increases, the

execution time difference between relational and JSON query execution time radically decreases, especially between small and big number of records as mentioned above (see Figures 4, 5). This indicates that the relational table outperforms JSON table. The selection of records in scenario 2.b has been performed up to 400K returned records, due to PostgreSQL *'out of memory'* reached limit.

Table 5: Average select query execution and jitter time for relational table

No. of selected records	Average Selection time for relational table(ms)	Average Selection jitter time for relational table(ms)
10K	968.3041573	2.483129505
100K	397.4092007	-19.48785782
200K	1638.284922	-11.76905575
400K	2348.677874	-76.88879967

Scenario 3.a Aggregation function query time and jitter time on JSON table

For JSON table aggregation queries, authors used the average (AVG) aggregation function and measured its execution time over the number of selected documents. For small number of documents there is a constant execution time of 3350ms. For big number of records the average execution time increases up to 6000ms. Results are presented at Table 6.

Jitter time is close to 150 milliseconds for small number of aggregated documents. For big number of documents, jitter time is close to182ms. This means that jitter time is almost constant over the number of aggregated documents. This means that there are minimum inconsistencies close to 3.5% of the average aggregation execution time.

Table 6: Average Aggregation function time and jitter time for JSON table

No. of aggregate function records	Average Aggregation function time for JSON table(ms)	Average Jitter Aggregation function time for JSON table(ms)
10K	3306.502104	145.2691555
100K	3350.203037	128.3073425
200K	3386.135101	158.2260132
300K	3659.034967	112.8232479
400K	4022.873878	161.7748737
500K	4420.567989	153.7988186
700K	4824.729919	192.7640438
800K	5104.82502	147.9272842
900K	5309.696913	261.1157894
1M	5788.780928	156.0409069

Scenario 3.b Aggregation function query time and jitter on relational table

For the relational table aggregation queries, authors used the average(AVG) aggregation function and measured its execution time over the number of selected records. It should be mentioned that whilst implementing a select query over big number of records, the PostgreSQL required memory and crashed out. Whereas, while calling an aggregated function over a big number of records, the PostgreSQL successfully returned the results.

As presented at Table 7, for small number of records the aggregation function execution time is on average 175ms, while for big number of records execution time is 225ms on average. Jitter time presents variations around 2.75ms on average, which corresponds to the 1.3% of the average aggregation function execution time. This means that there are no important inconsistencies regarding the aggregation functions execution time for the relational table. The comparison between relational and JSON tables in aggregation function query time has shown that relational table is 20 times faster than JSON table. This signifies that relational table in aggregation function query time is much more efficient than JSON table.

Table 7: Average Aggregation function query time on relational table

No. of aggregate function records	Average aggregation function time for relational table(ms)	Average Jitter time for aggregation function for relational table(ms)
10K	173.3779907	0.271558762
100K	176.5909195	1.517009735
300K	185.8799458	1.37758255
400K	196.336031	0.472784042
600K	206.0739994	1.980781555
700K	216.812849	0.169754028
800K	225.0239849	1.870632172
900K	233.2780361	8.156061172
1M	245.1839447	9.302377701

Scenario 4.a Selection query and jitter time for JSONB table

In this scenario, authors examined the JSONB data fields using the same IoT dataset and performing select queries. The results are presented in Table 8. The average selection query time for the JSONB table is proportional to the number of selected documents. For small number of documents, the average selection query time is 1450ms. For big number of records is 6700ms on average. Results are presented in Table 8.

Jitter time for JSONB table for small number of documents is approximately to 0.4% of the average selection query execution time. For big number of records jitter is close to 50ms that corresponds to the 0.7% of the average selection query execution time. This means that the JSONB table select queries are consistent in respect to their execution time

The comparison amongst relational, JSON and JSONB tables has shown that JSONB table selection query time is 1.5 faster than the JSON table. Furthermore, the relational table selection queries time are 3 times faster than the JSONB table queries. Jitter time is almost constant for JSONB and relational tables which signifies data consistency over queries. However, there have been query inconsistencies spotted on the JSON table for data selection queries, mainly due to its spurious variations over the number of selected documents.

Table 8: Average select query time and jitter time for JSONB table

No. of selected records	Average JSONB Selection time (ms)	Average Jitter time for Selection time for JSONB table (ms)
10K	1364.244938	-4.009246929
100K	1467.782974	-9.11569571
200K	2437.329054	-6.512642028
400K	3509.732008	31.91089628
500K	4704.421997	-20.36929124
700K	5936.168909	-59.06295769
800K	6692.276001	30.12999151
900K	7408.374071	-77.4040221
1M	8818.401098	-197.371721

Scenario 4.b Aggregation function query time on JSONB table

Authors also tested the average (AVG) aggregation function on the JSONB table. As presented in Table 9, for small number of documents, average aggregation function query time on JSONB table remains constant around 405ms. For big number of documents, query time is at 550ms on average.

Jitter time for JSONB table on aggregation function for small number of documents is approximately to 0.1% of the average selection query execution time. For big number of records jitter is close to 225ms that corresponds to the 0.3% of the average aggregation function query execution time. This means that the JSONB table select queries are consistent in respect to their execution time.

Table 9: Average aggregation function query time on JSONB table

No. of aggregate function records	Average aggregation function query time for JSONB table(ms)	Average aggregation function jitter time JSONB table(ms)
10K	405.796051	-6.53076
100K	404.6578407	-9.71961
300K	405.4729939	3.271341
400K	447.701931	-5.25808
600K	513.0839348	-2.50816
700K	555.0577641	-0.22125
800K	580.655098	-1.71018
900K	606.5020561	19.15026
1M	646.1930275	38.83266

The comparison amongst relational, JSON and JSONB tables has shown that JSONB table aggregation function query time is 10 times faster than the JSON table. Furthermore, the relational table selection query time is 3 times faster than the JSONB table. Jitter time is consistent between JSONB and relational tables. Jitter time is close to 6.5 ms for small number of records and 20ms for big number of records. Jitter time corresponds to 1.28% of the average aggregation function query time on average for the JSONB table. This means that there are no important inconsistencies regarding the aggregation functions execution time for the JSONB table.

IV. Conclusion

In this paper, authors examined the performance of relational and non-relational PostgreSQL fields using IoT data, through a series of experimental scenarios. From the authors' experimentation it has been noticed that on insert queries JSON table is much more efficient than relational table. Specifically, relational table insert query execution time is 3 times faster than the JSON table. However, on the select query experiments, the relational table performed 6 times better than the JSON table and 3 times faster than the

JSONB table. On aggregation function experiments, the relational table performs 2 times faster than JSONB table, and in turn the JSONB table performs 7-10 times faster than the JSON table.

Authors noticed that there is guaranteed data consistency as spotted by jitter time metric amongst relational, JSON and JSONB tables, with the exception of high probability of query variations for the JSON table, close to 7% of the average selection query execution time and minimum inconsistencies for aggregation function queries for the JSON table, close to the 3% of the average aggregation function execution time.

Concluding, the main IoT data requirements are fast database inserts and schema less records. Fast database inserts contribute positively on the IoT devices energy conservation, while schema less is essential for the IoT industry due to the IoT devices and embedded to the devices sensors escalation. Based on the aforementioned information requirements, the authors forward the JSONB data type for IoT application use, since its improvements over the relational table bring it closer to the relational table performance while fully adopting the IoT primary requirements.

References

[1] Aboutorabi S., Rezapour M., Moradi, M., and Ghadiri, N., "Performance evaluation of SQL and MongoDB databases for big e-commerce data ", In proc. of CSICSSE conf., DOI: 10.1109/CSICSSE.2015.7369245, 2015

[2] Afolabi A. O. and Ajayi A. O., Performance Evaluation of a database Management System, Journal of Engineering and Applied Sciences Vol. 3 (2): 155-160, 2008, ISSN: 1816-949X, 2008

[3] Al-Qerem, A. Performance Evaluation of Transaction Processing in Mobile Data Base Systems, International Journal of Database Management Systems (IJDMS) vol. 6 (2), ISSN: 0975-5985, 2014

[4] Berg K., Seymour T., and Coel R., History of Databases. International Journal of Management and Information Services, Vol. 17,No. 1, DOI: 10.19030/ijmis.v17i1.7587, 2013

[5] Damodaran D. B., Salim S. and Vargese M. V., Performance evaluation of MySQL and MongoDB databases, International Journal of Cybernetics & Informatics IJCI, Vol. 5, No. 2, ISSN:2320-8430, 2016

[6] DB-Engines, System Properties Comparison MySQL vs. PostgreSQL, Technical report. https://db-engines.com/en/system/MySQL%3BPostgreSQL%3BSTSdb, 2016

[7] DB-engines, The DB-Engines Ranking ranks database management systems according to their popularity, Internet: https://db-engines.com/en/ranking, 2018

[8] Gyorodi C., Olah, A. I., Gyorodi R. and Bandici L., A Comparative Study between the Capabilities of MySQL vs. MongoDB as a Back-End for an online Platform, International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 7(1), ISSN:2158-107X, 2016

[9] JSONB, JSON Binding Binary data v. 1.0 for converting java objects to JSON messages, http://json-b.net/users-guide.html, 2016

[10] Kontogiannis S. and Karakos A., ALBL: An Adaptive Load BaLancing algorithm for distributed web systems, International Journal of Communication Networks and Distributed Systems, Vol. 13(2), July 2014, pp. 144-168, 2014.

[11] Lourenco J.R., Abramova V., Vieira M., Cabral B., Bernardino J., NoSQL Databases: A Software Engineering Perspective. In: Rocha A., Correia A., Costanzo S., Reis L. (eds) New Contributions in Information Systems and Technologies. Advances in Intelligent Systems and Computing, vol 353. Springer, Cham, 2015

[12] Maksimov D., Performance Comparison of MongoDB and PostgreSQL with JSON types, Master Thesis, Tallin University of Technology, faculty of Information Technology, https://digi.lit.ttu.ee, 2015

[13] Mei S., Why you should never use MongoDB, http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/, 2013

[14] MongoDB, MongoDB document database and documentation, Internet: https://docs.mongodb.com, 2012

[15] NoSQL Archive, List of NoSQL Databases, http://nosql-database.org, 2018

[16] Ovais T., Databases: Normalization or Denormalization. Which is the better technique?, Technical report, http://www.ovaistariq.net/199/databases-normalization-or-denormalization-which-is-the-better-technique/#.W0S30nsVTIU, 2010

[17] Parker Z., Scott P., Vrbsky V. Susan, Comparing NoSQL MongoDB to an SQL DB. Proceedings of the 51st ACM Southeast Conference. DOI: 10.1145/2498328.2500, 2013

[18] PostgreSQL, PostgreSQL JSON and JSONB datatypes, https://www.PostgreSQL.org/docs/9.4/datatype-json.html, 2016

[19] Stancu-Mara, S., and Baumann, P., A Comparative Benchmark of large Objects in Relational Databases. In Proc. of the 2008 international symposium on Database engineering & applications, pp. 277-284, ACM, 2008

[20] Strozzi C., NoSQL: Non-SQL RDBMS, http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, 2016

[21] Tech Differences Foundation, Difference between Normalizations and Denormalization, Technical report, https://techdifferences.com/difference-between-normalization-and-denormalization.html, 2017

Christodoulos Asiminidis. " Managing IoT data using relational schema and JSON fields, a comparative study. "IOSR Journal of Computer Engineering (IOSR-JCE) 20.6 (2018): 46-52.