# Cassandra for Internet of Things: An Experimental Evaluation

André Duarte
*Polytechnic Institute of Coimbra, ISEC, Rua Pedro Nunes, 3030-199, Coimbra, Portugal*

Jorge Bernardino
*Informatics, Polytechnic Institute of Coimbra, ISEC, Rua Pedro Nunes, 3030-199, Coimbra, Portugal*
*CISUC – Centre of Informatics and Systems of the University of Coimbra, University of Coimbra, 3030-290, Coimbra, Portugal*

Keywords: Internet of Things (IoT), Cassandra, NoSQL.

Abstract: The proliferation of the Internet of Things (IoT) increases the amount of data that is being produced. Therefore it is extremely important to find the best possible storage engine to process these huge amounts of data. With the intent of discovering which database engine better supports the characteristics of an IoT system it is essential to analyse the existing databases and test them in a practical context. With this objective we decided to use one of the most popular databases, Cassandra, to evaluate it on an IoT environment. We evaluate the querying processing time of Cassandra using queries of an IoT real time environment and comparing it with different types of data architectures. The main focus of this work is to investigate if Cassandra can provide good performance in an IoT system.

## 1 INTRODUCTION

Nowadays the world is evolving and producing large amounts of data due to the growth of Internet of Things (IoT). This constant and fast evolution leads developers to pursuit the best possible solutions to handle large amounts of data. Even though the need for intelligent data mining tools is extremely important, we also need to pay attention to the way this data is stored and which type of engine better fits the needs of an IoT system.

To the best of our knowledge, a perfect solution for the Internet of Things data layer does not exists. With this in mind we aim to find out the best possible solution for this type of environment. Therefore an investigation was conducted to understand which database would be the most suitable to provide a production ready environment. It is important to keep in mind that, generally speaking, these kind of systems need to handle large amounts of data, real time insertion of records and huge data diversity.

The database will be used in an Internet of Things environment that intends to gather data from a city and process it in order to find events that are considered dangerous. The system collects data from sensors and provides alerts to each subscribed application. It is important to understand that this system will act as a demonstrator for the data layer.

In (van der Veen et al., 2012) it is discussed that scaling systems that deal with sensors is becoming gradually difficult due to the amount of sensors and clients that extract data from them. Therefore it is significant to not only pay attention to the frequency of the data, but also to the huge volume that it will obtain.

According to (Abramova et al., 2014a; 2014b; 2014c; Barata et al., 2015) Cassandra seems to have a clear advantage in terms of the characteristics necessary to implement this system because it provides good writes speeds without sacrificing performance. Furthermore, Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency (Lakshman and Malik, 2010). Additionally the decision of choosing Cassandra is a result of its popularity and market share (DB-Engines Ranking, 2015). With all of this in mind, Cassandra seems to be a solid choice for this use case.

In addition to storing data, every system needs to provide it in order to query and filter later. It is



Duarte, A. and Bernardino, J.
Cassandra for Internet of Things: An Experimental Evaluation.
DOI: 10.5220/0005846400490056
In *Proceedings of the International Conference on Internet of Things and Big Data (IoTBD 2016)*, pages 49-56
ISBN: 978-989-758-183-0

important to keep in mind that systems included in the IoT context tend to be stream oriented, rather than batch. For this reason, the database to be chosen needs to accept data in streams, or at least support a high rate of data insertion, and have the necessary mechanisms to withstand this.

We aim to test which architecture for the data layer would best suit the needs of an IoT platform in terms of querying performance, without sacrificing the write performance. There were two relevant ways in terms of implementation. First, a single table with all the data, which would then be filtered and dealt with when needed. A second approach is multiple tables for each specific application that sends events.

From a theoretical standpoint it seems that the best way of organizing our data is through the creation of a table per application. This will result in smaller tables which, in comparison to a centralized table that stores everything are a lot faster because they have much less records. In a nutshell, we aim to understand which data architecture will have the best performance while querying the data.

The remainder of this paper is structured as follows. Section 2 gathers background on important concepts such as the IoT concept and the description of Cassandra. Section 3 describes Cassandra and its general characteristics. Section 4 defines the setup on which the tests were made. Section 5 presents the performance tests that were made. Finally, section 6 presents our main conclusions and suggests future work.

# 2 BACKGROUND

This section aims to present the necessary background on Internet of Things and Databases.

## 2.1 Internet of Things

According to (Friess and Vermesan, 2013) the Internet of Things (IoT) "is a concept and a paradigm that considers pervasive presence in the environment of a variety of things/objects that through wireless and wired connections and unique addressing schemes are able to interact with each other and cooperate with other things/objects to create new applications/services and reach common goals." IoT is a concept reflecting a connected set of anyone, anything, anytime, anyplace, any service, and any network (Islam et al., 2015). The IoT is a megatrend in next-generation technologies that can impact the whole business spectrum and can be thought of as the interconnection of uniquely identifiable smart objects and devices within today's Internet infrastructure with extended benefits. Benefits typically include the advanced connectivity of these devices, systems, and services that goes beyond machine-to-machine (M2M) scenarios (Höller et al., 2014). The IoT provides appropriate solutions for a wide range of applications such as smart cities, waste management, emergency services, logistics, retails, industrial control, and health care. The Internet of Things extends even beyond communications and new services, it will allow for a future where, with everything connected, people can feel more integrated with the world and let IoT do the day-to-day recurring tasks for them.

The IoT provides a new paradigm that will shape the world and create a new conception of the Internet and how people interact with it, due to the constant interconnectivity between people and the world (Jara et al., 2014). It will also provide the necessary resources for the creation of new applications and data driven platforms that will, hopefully, improve the citizen's quality of life. This new way of reinventing the Internet will not only provide endless possibilities to improve the overall interaction between humans and machines but also create new challenges, which need to be tackled, to cities themselves.

In short, Internet of Things is successfully thriving in the current world, therefore intelligent systems will continue to emerge alongside it.

## 2.2 Databases

A database can be treated as a related set of information, which allows the developer to access the data via queries that intend to express his/her needs regarding that set of data in that specific timeframe, either by using simple statements or complex filtering to enhance the granularity of the search. For this data to be queried it needs to be inserted during the time that the database is in place. Nowadays there are many types of databases, however in this paper we will focus on Cassandra. Usually databases are divided in two main classes - SQL and NoSQL databases (Abramova et al., 2015):

- SQL databases – these are the traditional relational databases that intend to store data in a structured way. Famous SQL engines are: MySQL, MS SQL Server and Oracle (DB-Engines Ranking, 2015);
- NoSQL databases – NoSQL "is used to refer to the databases that attempt to solve the problems of scalability and availability against that of

atomicity or consistency" (Vaish, 2013). NoSQL databases are divided in four main groups according to each use case and architecture, these groups are:

- Key-Value databases – These are the simplest NoSQL databases, which are based in a key-value organization that allows the developers to make CRUD (Create, Read, Update, and Delete) operations only with a key. The type of storage is BLOB (Binary Large Object) and the data structure is not organized in any fashion. According to (Redmond, Wilson and Carter, 2012) these databases have a very good performance, although aren't good for complex querying and aggregation. Examples of these databases are: Memcached (Memcached, 2015), Couchbase (Couchbase, 2015) and DynamoDB (DynamoDB, 2015);
- Document databases – As the name implies this type of database stores and retrieves document like files, which can be XML, JSON amongst others. According to (Redmond, Wilson and Carter, 2012) a document is a hash with a unique ID which has more values related to it. Examples of these databases are: MongoDB (MongoDB, 2015) and CouchDB (CouchDB, 2015);
- Column Family databases – These databases store data um column families which are tables with columns that are frequently accessed together. According to (Redmond, Wilson and Carter, 2012) a columnar structure "is about midway between relational and key-value". Databases of this type are: HBase (HBase, 2015) and Cassandra;
- Graph databases – According to (Robinson, Webber and Eifrem, 2013), graph databases "are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind". Famous databases of this type include (DB-Engines Ranking, 2015): Neo4j (Neo4j, 2015), OrientDB (OrientDB, 2015) and Titan (Titan, 2015).

In our case we opted for NoSQL databases because they seem the more appropriate fit for systems in the IoT paradigm. On the NoSQL databases we have opted for Cassandra, the next section will serve to explain our choice while introducing important topics related to Cassandra.

# 3 CASSANDRA DATABASE

Cassandra is a distributed storage system that manages large amounts of data across servers (Lakshman and Malik, 2010). Still according to this author Cassandra uses a combination of well-known procedures that grant scalability and availability.

In this section we will start by introducing Cassandra's data model and, later on, we will explain Cassandra's architecture.

The data model of Cassandra provides a high processing speed when writing the data, this is due to the indexing. Cassandra indexes data by key, which is a unique representation of the row that contains the data. Each row contains columns, which are attributes and finally these columns make up a column family. Figure 1 illustrates the data model, which is composed by rows, column families and keyspaces.

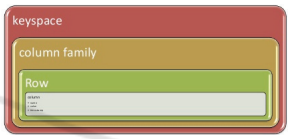


Figure 1: Cassandra's Data Model (Charsyam, 2011).

Furthermore we shall address the two important concepts that make up the data representation in Cassandra, which are the column families and the keyspaces.

- Column Family – A column family is a container for a group of rows (Hewitt, 2011). Column families are not defined, which means that the structure can be changed at any desired time, this improves the system's readiness to change and adapt during time;
- Keyspace – In Cassandra the keyspace is the equivalent to a database in the relational paradigm. The keyspace contains the column families which make up the full database. The keyspaces contain attributes that can be tuned to enhance the overall performance of the database, these attributes are: (1) replication factor, which refers to the number of physical copies of the data. For example if the replication factor is set to two data will be replicated twice; (2) replica placement strategy, this attribute is used for defining the strategy of how data is placed in the cluster. The possibilities to define the replicas are, Simple Strategy which is most used when we have a single group of nodes in the cluster and Network Topology Strategy which is more used when the cluster is working across multiple

machines providing a way of managing the replicas in all the machines.

Cassandra uses a peer-to-peer architecture, which means that all nodes within a cluster can receive a request and respond to it (Strickland, 2014). This provides better availability when the database is online. Also, this provides redundancy, which help to keep the data safe and horizontal scalability. In Figure 2 we can observe the Cassandra peer-to-peer architecture.

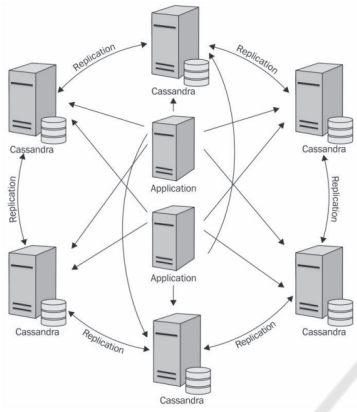


Figure 2: Cassandra Architecture (Strickland, 2014).

Furthermore, this architecture provides high availability to the database, which means that the system does not have a large downtime period, providing constant access to the data.

# 4 EXPERIMENTAL SETUP

The experiments that will be made will allow to learn which approach is better when storing data in the IoT. As mentioned in section 1 we have decided that there were two ways to organize the database that would be relevant in terms of implementation. A single table with all the data, which would then be filtered and dealt with when needed, or multiple tables for each specific application that sends events.

From a theoretical standpoint it seems that the best way of organizing our data is through the creation of a table per application. This will result in smaller tables which, in comparison to a centralized table that stores everything is considerable faster because they have significantly less records. Figure 3 illustrates the two different approaches.

The experimental setup was created with the following characteristics: (1) The operating system was Ubuntu 14.04 LTS 64bit; (2) The machine had a dual core, Core i5 480m with 6GB of RAM and an HDD; (3) The database ran in a single node to understand the minimum possible requirements when running the system; (4) The dataset used contained environmental data.

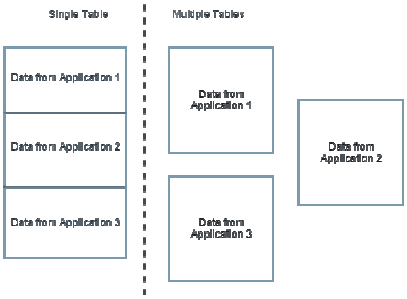


Figure 3: Data layer possible architectures.

We have decided not to use a benchmark tool because we wished to approximate the tests made with a real world scenario. Also, with this approach we guarantee that the performances we see are more accurate and can be replicated in a production environment.

The chosen queries intend to illustrate regular situations during the usage of the system, which reflect the better approaches to the problem, keeping in mind that attention to the write speed is also needed. To analyse them, different queries will be created, matching the needs while the system is in place. These queries may vary from time to time, although some of them will be a recurrent task that needs to be performed. Additionally, it is important to keep in mind that these queries are to be performed in an IoT system, which generates alerts with the data that comes from the sensors scattered around a city. These alerts are filterable and searchable throughout the lifecycle of the system.
In the experiments we have three queries types:

Q1: Alert selection from a specific type – This query is performed to provide the number of alerts of each type (e.g. Number of ‘warning’ alerts);

Q2: Alert selection for a submitted rule – This query will be used to see how many alerts were raised by a submitted rule (e.g. how many alerts were generated by rule X);

Q3: Alert selection in a range of time – This query serves to select a type of alerts (e.g. ‘warning’, ‘critical’) in a period of time.

These queries give a broad perspective of the system in terms of querying performance.

To query the database we use the Cassandra CQL shell, to record the times we have enabled tracing which allow us to have a detailed view of the

query and created indexes to allow filtering to happen.


Figure 4: Row prototype.

Figure 4 shows the row prototype which is composed by the following columns:

- alert_uuid – This field is of the type UUID, it represents the universal id of the alert to keep each alert unique;
- config_id – This field is of the type UUID, it represents the application id which created this alert;
- event_query – This field is of type TEXT and it represent the rule needed to fire the alert;
- alert_type – This field is of the type VARCHAR and represents the type of alert which was generated (i.e. Critical, Warning);
- event_type – This field is of the type VARCHAR and represents the type of event to be processed (i.e. Environment, Traffic);
- event_window – This field is of the type TEXT and represents the event window which triggered the alert;
- event_body – This field is of the type TEXT and represent the full event which triggered the alert;
- created_on – This field is of the type TIMESTAMP and it represents the timestamp on which the alert was triggered.

On the next section we will present the results of the experiments.

# 5 EVALUATION

In this section we show the experimental evaluation for query processing time. Each chart contains, in the Y axis, the "Query Time (ms)" which represents the time the queries took to be processed. In the X axis, we have "Table Name" which represents the table where the query was made. The tables are divided by configuration and each represents an application. The "Table Name" axis uses the following notation:

- App1-App5: correspond to applications with data that comes from environmental sensors. Each of these applications have 100.000 records;
- All: corresponds to the single table containing all the information. This table will have 500.000 records.

The values presented in the experiments were obtained by executing the same query five times and then calculating the average value. Also, the first three queries of each run were discarded due to the possibility of cold boots. In the figures the dots represent the average value of the query speed and the error bars represent the standard deviation to that value. For a better approximation of a real system, the queries were made in no specific order. This has to do with the Cassandra reading architecture which is faster if the table is in memory.

In the next sections we will show the values obtained during the experiments and present a summary of the values obtained.

## 5.1 Querying an Alert of a Specific Type (Q1)

In the experiment we use this query to select all the alerts of type 'warning' from the applications. Using the CQL language the query looks like this:

```
SELECT * FROM query_performance.alerts_
<config_id>    WHERE    alert_type    =
<alert_type>;
```

For the table with all of the data the query used was:

```
SELECT * FROM query_performance.alerts_
full WHERE config_id = <config_id> AND
alert_type    =    <alert_type>    ALLOW
FILTERING;
```

This a very simple query, since it only lists the alerts of type "warning" that were generated by the application. However it is expected to see an enormous change in terms of performance, due to the amount of data in the "All" table. Figure 5 shows the performance for Q1.

When analysing the results of Q1, shown on Figure 5, we can conclude that the separate tables were, in general, the best choice. Although in the second application we saw a little deviation from the average value, this is related with the reading architecture of Cassandra which is faster if the table is in memory.

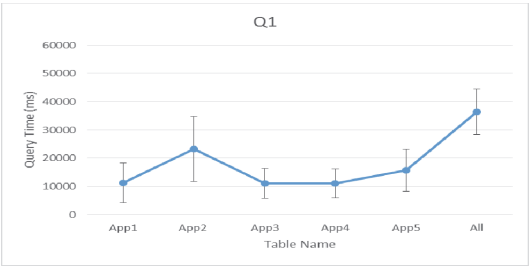


Figure 5: Execution of Query 1.

As explained before, we have tried to make queries to different tables in order to provide results which are useful for people who want to know if this database is a liable option for a production system.

## 5.2 Querying an Alert for a Rule (Q2)

This query intends to list every alert for a specific rule created by the user. The query, using the CQL language, will look like this:

```
SELECT * FROM query_performance.alerts_
<config_id> WHERE event_query=<rule>;
```

For the full table the query looks like this:

```
SELECT * FROM query_performance.alerts_
full WHERE config_id = <config_id> AND
event_query=<rule>    ALLOW FILTERING;
```

The query on the full table could not be completed because the operation timed out. The operation quitted when filtering the data with the where clause, this is due to the amount of data it needed to filter. We have tried to change the environment settings for Cassandra to try to overcome this situation, but the error persisted. This led to the removal of this query from the charts. Due to this problem, the comparison was made only between the applications. Furthermore, we can conclude that this query cannot be made in a production environment because the system cannot be stuck waiting for the query to end. On a real world system, and because IoT systems require near real time responses, it is impossible to implement this query because of the error it kept raising. Figure 6 shows the performance for Q2.

With the results of the execution of Q2, seen, we conclude that every application has similar performances when dealing with this query. The main conclusion to draw from this experiment is that the table with all the data could not be queried because it kept raising an out of time error. This is due to the amount of data which is stored in that table which Cassandra cannot filter.

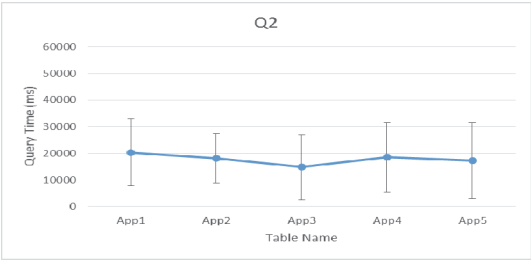


Figure 6: Execution of Query 2.

## 5.3 Querying an Alert on a Time Range (Q3)

This query selects all the alerts of each application in a time range. In the real system this query is important because it delivers a time based approach to the data. Using the CQL language the query looks like this:

```
SELECT * FROM query_performance.alerts_
<config_id>    WHERE    created_on    <=
<timestamp> AND config_id = <config_id>
ALLOW FILTERING;
```

The query made on the table with all of the information will look like this:

```
SELECT * FROM query_performance.alerts_
full  WHERE  created_on  <=  <timestamp>
AND    config_id    =    <config_id>    ALLOW
FILTERING;
```

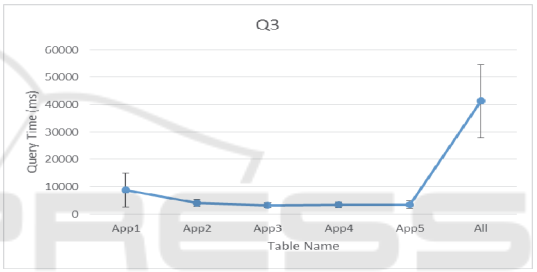


Figure 7: Execution of Query 3.

Figure 7 shows the processing execution time for Q3.

The query Q3, had comparable performance across all of the separate tables, the standard deviation on the first application is higher, due to discrepancy between the performances of when the table is in memory and needs to be loaded to memory. We can also see that the average time for the table with all the data is much higher than the others, once again proving that an architecture where the data is separated is better.

## 5.4 Results Summary

The results show that, as expected, the single table had the worst performance. This is due to the amount of data that Cassandra has to filter, which cannot be placed in memory all at once. Although the results of the "All" table were not five times worse we conclude that the best implementation is with separate tables which not only give a better performance, but also provide a better overall data separation.

The performance changes between the first two applications are a little bit different, this might be due to the size of the string that is being searched. The main differences are between the "All" table, which was finished on Q1 but not on Q2. This is due to the fact that, on these tables, data is sequentially organized which means that if the query results are not on the first records, Cassandra cannot load all the data to memory and initiate the filtering process.

The average query processing time in Q3 is smaller than on the others, this is related to the fact that the dataset is not heterogeneous enough in terms of dates because the values of the applications were recorded on a single day. Also, filtering is made by primary key because in Cassandra to make a time range query the column with the date needs to be on the primary key of the table.

In short, we think that these queries, although very straightforward, give a quick and simple performance overview to a data layer architecture in the IoT.

# 6 CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, a complete solution for the IoT data layer does not exist. With the intent to find a suitable and workable solution we have tested two different architectures for the data layer, which provide two different approaches when dealing with data. For this we have evaluated the NoSQL database Cassandra, which have been applied in an Internet of Things platform. The queries that were made gave us, not only an initial perspective of how Cassandra will handle the system workloads, but also will provide knowledge for whoever wants to have an idea of how Cassandra handles data in the IoT environment.

To run the tests we have tried to make constant changes to the query order to enhance the credibility of the results, this was done because the system will not have a constant pattern of querying, when deployed. This has a great impact in terms of query performance because, as we have seen before, if a table is queried twice in a row the second time it will be in memory. Additionally, it is important to refer that the tests were made on a personal computer, which makes the RAM management more difficult, due to other processes that might be running at the same time.

The results show that the single table had the worst performance. From this, we conclude that the best implementation is with separate tables, which not only give a better performance, but also provide a better data independence. In the IoT, data is produced continuously by each application, which means that distinct tables would also be a good choice, providing an independent way for each application to store its data and be able to scale without sacrificing performance.

In summary, from this work we can conclude that Cassandra can be used on an IoT platform as the main database system because it contains the necessary characteristics to handle the overall requirements of these platforms.

The dataset used could be larger and more heterogeneous, although the results have shown differences between the two approaches. Nevertheless, tests with larger datasets and with a bigger variety of data are needed in order to understand if scalability is an issue.

As future work we suggest that similar tests can be made with sharding, which is a horizontal division of data that improves the overall performance of the queries. The main goal is to divide the applications by shard, providing a similar approach to the separate tables we have seen. We also would like to distribute the system, testing it for better availability.

# ACKNOWLEDGEMENTS

This work was partially financed by national funding via the Foundation for Science and Technology and by the European Regional Development Fund (FEDER), through the COMPETE 2020 – Operational Program for Competitiveness and Internationalization (POCI).

This work was also made possible with the help of Ubiwhere, Lda, which provided the dataset and useful inputs in discussions.

# REFERENCES

Abramova, V. and Bernardino, J. (2013). NoSQL databases. *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*, pp. 14-22.

Abramova, V., Bernardino, J. and Furtado, P. (2014a). Evaluating Cassandra Scalability with YCSB. *International Conference on Database and Expert Systems Applications*, DEXA 2014, pp.199-207.

Abramova, V., Bernardino, J. and Furtado, P. (2014b). Testing Cloud Benchmark Scalability with Cassandra. *2014 IEEE World Congress on Services*.

Abramova, V., Bernardino, J. and Furtado, P. (2014c). Which NoSQL Database? A Performance Overview. *Open Journal of Databases (OJDB)*, Vol 1. Issue 2, pp.17-24.

Abramova, M., Bernardino, J. and Furtado, P. (2015). SQL or NoSQL? Performance and scalability evaluation. *Int. Journal of Business Process Integration and Management*, Vol. 7 (4), pp. 314-321.

Barata, M., Bernardino, J. and Furtado, P. (2015). Cassandra: what it does and what it does not and benchmarking. *Int. Journal of Business Process Integration and Management*, Vol. 7 (4), pp. 364-371.

Charsyam - Cassandra Data Model - https://charsyam.wordpress.com/tag/cassandra-data-model/ [online] Available at: [Accessed 08-01-2015]

Couchbase.com, (2015). *Couchbase*. Available at: http://www.couchbase.com/ [Accessed 25 Sep. 2015].

Couchdb.apache.org, (2015). *Apache CouchDB*. [online] Available at: http://couchdb.apache.org/ [Accessed 25 Sep. 2015]

DataStax, (2014). *ALLOW FILTERING explained*. [online] Available at: http://www.datastax.com/dev/blog/allow-filtering-explained-2 [Accessed 5 Jul. 2015].

DB-Engines Ranking [online] http://db-engines.com/en/ranking (Accessed 22 April of 2015)

Docs.aws.amazon.com, (2015). *What Is Amazon DynamoDB? - Amazon DynamoDB*. Available at: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html [Ac. 25 Sep. 2015].

Docs.datastax.com, (2015). Apache Cassandra™ 2.0. [online] Available at: http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html [Accessed 25 Oct. 2015].

DZone, (2015). *DZone Database*. [online] Available at: https://dzone.com/articles/introduction-apache-cassandras [Accessed 21 Jul. 2015].

Hbase.apache.org, (2015). *Apache HBase – Apache HBase™ Home*. [online] Available at: http://hbase.apache.org/ [Accessed 25 Sep. 2015].

Hewitt, E. (2011). *Cassandra The definitive guide*. Beijing. O'Reilly.

Höller, J., Tsiatsis, V., Mulligan, C., Karnouskos, S. Avesand, S. and Boyle D., From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence. Amsterdam, The Netherlands: Elsevier, 2014.

Islam, S. M.; Kwak D., Kabir H., Hossain, M., Kyung-Sup Kwak, "The Internet of Things for Health Care: A Comprehensive Survey," in Access, IEEE , vol.3, no., pp.678-708, 2015

Jara, A. J.; Genoud, D.; Bocchi, Y., "Big Data in Smart Cities: From Poisson to Human Dynamics," Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on ,, pp.785- 790, 13-16 May 2014

Lakshman, A. and Malik, P. (2010). Cassandra. *SIGOPS Oper. Syst. Rev.*, 44(2), p.35.

Memcached.org, (2015). *memcached - a distributed memory object caching system*. [online] Available at: http://memcached.org/ [Accessed 25 Sep. 2015].

MongoDB, (2015). *MongoDB*. [online] Available at: http://mongodb.com [Accessed 25 Sep. 2015].

Neo4j Graph Database, (2015). *Neo4j, the World's Leading Graph Database*. [online] Available at: http://neo4j.com [Accessed 25 Sep. 2015].

OrientDB Multi-Model NoSQL Database, (2015). *OrientDB - OrientDB Multi-Model NoSQL Database*. Available at: http://orientdb.com/orientdb/ [Accessed 25 Sep. 2015].

P. Friess and O. Vermesan, *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. Aalborg, Denmark: River Publishers, 2013.

Redmond, E., Wilson, J. and Carter, J. (2012). Seven databases in seven weeks. Dallas, Tex.: Pragmatic Bookshelf.

Robinson, I., Webber, J. and Eifrem, E. (2013). *Graph databases*. Sebastopol, Calif.: O'Reilly Media.

Strickland, R. (2014). *Cassandra high availability*. Birmingham. Packt Publishing.

Thinkaurelius.github.io, (2015). *Titan: Distributed Graph Database*. [online] Available at: http://thinkaurelius.github.io/titan/ [25 Sep. 2015].

Vaish, G. (2013). Getting started with NoSQL. Birmingham: Packt Publishing.

van der Veen, J. S.; van der Waaij, B.; Meijer, R.J., "Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual," Cloud Computing (CLOUD), 2012 IEEE 5th Int. Conference, pp.431- 438

Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. In: Proc. of ACM Symposium on Operating Systems Principles (SOSP 2001), pp. 230–243.