# EVALUATION OF GRAPH DATABASES PERFORMANCE THROUGH INDEXING TECHNIQUES

Steve Ataky Tsham Mpinda[1], Lucas Cesar Ferreira[1], Marcela Xavier Ribeiro[1], Marilde Terezinha Prado Santos[1]

[1]Department of Computer Science – Federal University of São Carlos (UFSCar)
São Carlos – SP – Brazil

***Abstract.***

*The aim of this paper is to evaluate, through indexing techniques, the performance of Neo4j and OrientDB, both graph databases technologies and to come up with strength and weaknesses os each technology as a candidate for a storage mechanism of a graph structure. An index is a data structure that makes the searching faster for a specific node in concern of graph databases. The referred data structure is habitually a B-tree, however, can be a hash table or some other logic structure as well. The pivotal point of having an index is to speed up search queries, primarily by reducing the number of nodes in a graph or table to be examined. Graphs and graph databases are more commonly associated with social networking or "graph search" style recommendations. Thus, these technologies remarkably are a core technology platform for some Internet giants like Hi5, Facebook, Google, Badoo, Twitter and LinkedIn. The key to understanding graph database systems, in the social networking context, is they give equal prominence to storing both the data (users, favorites) and the relationships between them (who liked what, who 'follows' whom, which post was liked the most, what is the shortest path to 'reach' who). By a suitable application case study, in case a Twitter social networking of almost 5,000 nodes imported in local servers (Neo4j and Orient-DB), one queried to retrieval the node with the searched data, first without index (full scan), and second with index, aiming at comparing the response time (statement query time) of the aforementioned graph databases and find out which of them has a better performance (the speed of data or information retrieval) and in which case. Thereof, the main results are presented in the section 6.*

**Keywords:**

*Evaluation, Comparison, Graph Database, Index system, Neo4j, Orient-DB.*

## 1. INTRODUCTION

Among the different data models, the relational model has dominated since the 80s, with implementations such as Oracle[1], MySQL[2] and MSSQL[3] - also known as the Relational Database Management Systems (RDBMS). Yet, lately in a growing number of use cases, the use of relational databases met pitfalls because of both problems and gaps in data modeling, as well as horizontal scalability constraints, distributed across multiple servers and large data volumes. There are two trends that have exposed these problems to the attention of the international developer community:

1. The exponential growth in term of data volume generated by users, systems and sensors, further accelerated due to the concentration of large portions of these volumes on large distributed systems like Amazon, Google and other cloud services.

2. The growing complexity and interdependence of data, accelerated by the Internet, social networks **Web 2.0** and opened access and standardized to data sources in a large number of different systems.

Relational databases are facing more difficulties to accommodate these trends. This led to the emergence of a number of technologies that address specific aspects of these issues, which can be used with existing. Alternative databases are nothing new, they have long existed in the form, for example, Object Oriented Databases, Hierarchical Databases (eg **LDAP**) and many others. However, in recent years some new projects have been launched and, in turn, came together under the name NoSQL Database, wherein data are denormalized and we rely on the application to meet generally with high latency and understanding [Steve et al. 2015 b]. One of the NoSQL databases, of increasingly importance, in which it is used the expressive power of the graph to build modeling complex structures, connected model as well as flexible, is the graph databases. [Han et al. 2010].

## 2. NOSQL ENVIRONMENT

NoSQL (Not Only SQL) is actually a very broad category grouping persistence solutions that do not follow the relational model, and not using SQL as a query language.

The term NoSQL was first used in earlier 1990, nonetheless it was only by the end of the 2000s that its options became much more focused and could be put into either of four different sectors or families.

In short, according to [Leonard 2014, Bogdan and Bucur 2011] NOSQL databases can be categorized according to their data models in the following 4 categories:

- Key-Values
- Column-family
- Documents Oriented
- Graphs Oriented Database

Below are examined two interesting aspects of NOSQL databases - scalability and complexity.

1. **Ramp-Load:** To ensure data integrity, most conventional database systems are based on transactions. This helps ensure data consistency. These transactional characteristics are also known by ACID acronym (Atomicity, Consistency, Isolation, Durability) [Brewer 2000]. Nevertheless, the horizontal increasing load on ACID systems has proven to be such a problematic exercise. There are conflicts between the different aspects of high availability in distributed systems that are not completely overridden - this is known as the CAP theorem.

– Strong Consistency: each user sees the same data version, even throughout the course updates of the data set.

– High Availability: each user can always acquire at least one copy of the data, in spite of the fact that some cluster machines may be unreachable.

– Partition Tolerance: the system as a whole keeps its characteristics even if deployed on different servers, transparently to the user.

In according with the CAP theorem, only two of the three scalability's aspects can be fully achieved simultaneously.

In order to work with large distributed systems, the various CAP constraints were examined more closely.

Many of NOSQL bases more than anything made concessions on Consistency constraints to obtain a better availability and better Partitioning. This led to the so-called systems BASE (Basically Available, Soft-state, Eventually).

Inasmuch there are no transactions from the classical sense and introduce constraints in the data model to allow better partition strategies.

2. **Complexity:** the growing interconnectivity of data systems has led to much denser data sets that cannot be automatically assigned as obvious, simple or domain independent, as noted by Todd Hoff. Reference may be made to Visual Complexity for details on viewing large and complex data sets.

## 3. GRAPH DATABASE

Prior to declaring overtaken the relational data model, one should call on mind that one of the reasons for the success of relational database systems (RDBMS) is its ability to model a supported data structure without redundancy or information loss, by means of the Normal Form. After the modeling stage, the data can be inserted, modified and interrogated under a complex and powerful way via SQL. As a matter of, there are some RDBMS that implement optimized schemas, e.g insertion speed or multidimensional queries for different use cases such as OLTP (online transaction processing), the OLAP (online analytical processing), web applications or reporting.

This is the theory. In practice, however, RDBMSs are reaching the limits of the CAP problem mentioned above, and have problems related to the implementation regarding SQL query execution performance "profound" that span many table joins . Amongst other problems such as scalability, schema evolution over time, modeling of tree structures, semi-structured data, hierarchies and networks, etc.

The graph, in turn, arose as an alternative to relational normalization [Steve et al. 2015a]; when we look at the projection of the business model on a data structure, there are two dominant schools - the relational way as used by RDBMS and graphs - networks and structures, used for example for the Semantic Web.

While structures are graph theory normalisable even in an RDBMS, this has serious implications in terms of performance for recursive structures such as trees or social graphs.

Each operation on a relationship in a network results in a join operation in the RDBMS, implemented as a set operation between all the primary keys of two tables - a slow operation and without ability to scale out while the number of tables' t-uples increases.

There is no general consensus on terminology regarding the graphs area. Nonetheless, an implicit definition is used and compared to other models which also involve graphs, like the object-oriented, semantic, and semi-structured models [Angles and Gutierrez 2008]. Thereby, there are many different graph models. Formally speaking, a graph is a collection of vertices and edges, in another word, a set of nodes and the relationships that connect them to each other [Robinson et al. 2013]. Graphs represent entities as nodes and the ways in which those entities relate to each other as relationships. Thence, some effort has been made to create the Attributed Graph Model (Property Graph Model), uniting the most different graph implementations. According to it, the information in a given graph is modeled using three basic blocks:

- node or vertex
- relationship or edge, with direction and type (oriented and marked)
- property or attribute, driven by an edge or a relationship

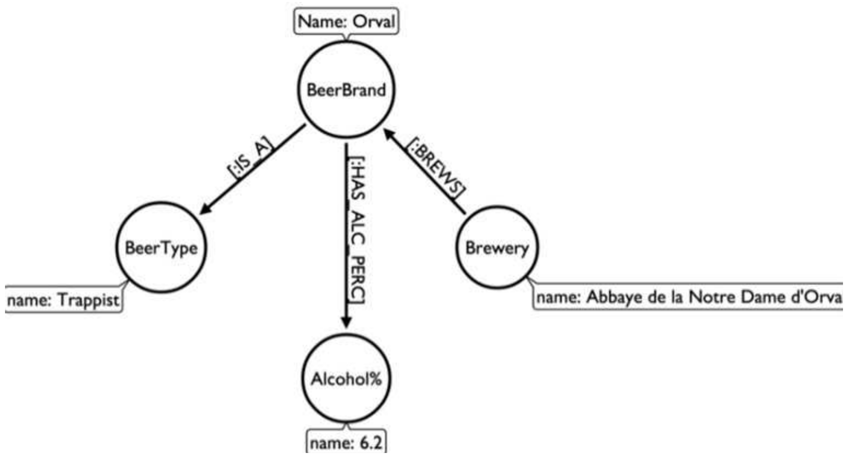

Figure1. A graph data model [Robinson et al. 2013]

A graph database management system (henceforward, GDB) is an online database management system capable of Creating, Reading, Updating and Deleting methods that expose a graph data model. Mostly, graph databases are built for use with transactional systems, henceforth (OLTP). Suitably, they are customarily optimized for not only transactional performance, but also engineered with transactional integrity, in addition of operational availability in sight.

According to [Robinson et al. 2013], there are two properties of graph databases which should be considered when investigating graph database technologies:

1 The underlying storage. Some GDB use native graph storage optimized and designed for storing and managing graphs. However, other GDB technologies do not use native graph storage. Thereby, others serialize the graph data into an object-oriented database, a relational database, or some other general-purpose data store.

2 The processing engine. Some definitions need that a GDB uses index-free adjacency, signifying that connected nodes physically "point" to each other in the database. Hither we take a somewhat broader view: any database that from the user's perspective behaves like a GDB, i.e. exposes a graph data model through CRUD operations, qualifies as a GBD. We do admit even so the notable performance advantages of index-free adjacency; whereby the term native graph processing is used to describe GDB that leverage index-free adjacency.

It becomes essential to point up that native graph processing and native graph storage are neither good nor bad; they are simply classic engineering tradeoffs. Regarding the benefit of native graph storage, its purpose-built stack is managed for performance and scalability. In contrast, the nonnative graph storage, rely on a mature non-graph backend whose production characteristics are well comprehended by operations teams. Native graph processing (index-free adjacency) benefits traversal [Marek et al. 2012, Macko et al. 2013] performance, however at the expense of making some non-traversal queries difficult or memory intensive.

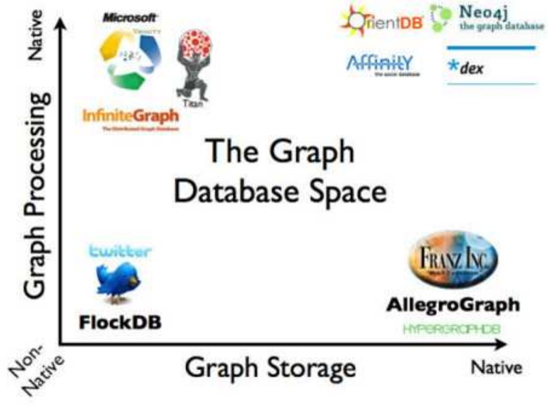

Figure 2. An overview of the graph database space [Robinson et al. 2013]

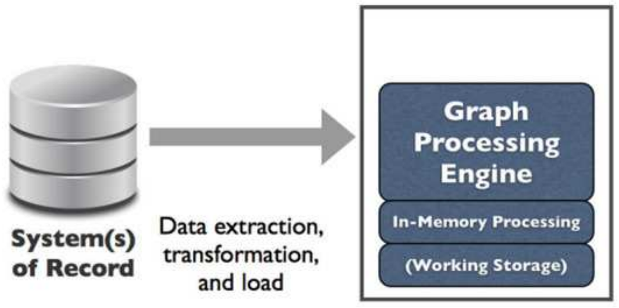

Figure 3. A high level view of a typical graph compute engine deployment [Robinson et al. 2013]

## 4. INDEXES IN GRAPH DATABASE

Native GDB are not decisively conditional on indexes owing to the fact that the graph itself provides a natural adjacency index technique. Moreover, in such GBD the relationships joined to a node naturally supply a direct connection to other related nodes of interest. Wherefrom graph queries may traverse through the graph. Such operations can be performed with utmost

efficiency, traversing very large number of nodes per second, instead of joining data through a global index.
Taking the granularity pattern even further and knowing that most indexing technologies actually use graphs/trees under the hood anyway, one can apply this pattern to create natural indexes for our data models, inside the graph. In accordance with [Bruggen 2014], doing so can be very useful for specific types of query patterns, such as time series and range queries.

Below are listed some graph database technologies which two of them one will use in order to achieve the aim herein proposed:

- Neo4j - Open Source Java Graph Model Awarded
- AllegroGraph - Closed Source, RDF-QuadStore
- Sones - Closed Source oriented .Net
- Virtuoso - Closed Source oriented RDF
- HypergraphDB - Open Source, Java, Hypergraph Model
- OrientDB - Open Source, support RDBMS and NoSQL
- Other such qu'InfoGrid Filament FlockDB, etc ...

## 5. GRAPH DATABASE TECHNOLOGIES AND RELATED INDEXING TECHNIQUES

### 1- Neo4j

Kindred other varieties of databases, Neo4j figures on an index to do an explicit look-up for a specific node or relationship. By the possibility to traverse the graph in order to find the node or relationship, using indexing is every so often more performant to handle the request. As illustration, let suppose one wants to look a specific "Customer" node, one could query the index by a unique identifier such as a customername or other unique key.

Additionally, from its version 2.0 at the end of 2013, Neo4j constructed a fundamental data model called under labels, which once assigned to nodes, Neo4j makes the data model of most users a lot simpler, in other words, there is no longer a need to work with a type property on the nodes, or a need to connect nodes to definition nodes that provide meta-information about the graph [Bruggen 2014]. Labels are a means to quickly and efficiently create sub-graphs. Likewise, labels may primarily be used for indexing and some limited schema constraints.

**Below a simple Cypher Command to Create the Users**

In the example below, one creates a very small graph that represents users in a social network. We consider here the Twitter's user relationships based-method, a bidirectional relationship called "FOLLOWS".

```
WITH
[ " Steve " , " Lucas " ,  " Ana " , " Geoffrey " , " Kenny " ,
" Michael " , " Luiz " , " Heir " ] AS fname ,
[" Seesharp " , " Ataky " , " Stoffolette " , " Rubyster " ,
" Braatz " , " Writesalot " , " Cesar " , " Graphman " ] AS lname
FOREACH ( r IN range ( 0 , 7 ) |
   CREATE ( : User { id : r , username :
    lower ( fname [ r % size ( fname ) ]+" "+ r ) , firstname :
    fname [ r % size ( fname ) ] , lastname : lname [ r % s i z e ( lname ) ] }
) ) ;
```

### Simple code Creating a Unique Index

One needs to create an index for later lookups. Inasmuch as it is known ahead of time that the userId value will be unique, one can add a unique constraint that creates a unique index, which is faster than a standard index.

```
CREATE CONSTRAINT ON ( u : User ) ASSERT u.userId IS UNIQUE
```

### Cypher Command for All Users to Follow All Other Users

```
MATCH (u1: ' User '), (u2: ' User ')
WITH u1, u2
CREATE UNIQUE (u1 - [:FOLLOWS]->u2 )
WITH u1, u2
WHERE u1<> u2
RETURN u1, u2
```

## 2- OrientDB

OrientDB supports 4 kinds of indexes [OrientDB 2014]:

**1- SB-Tree:** Good mix of all and the default one is durable and transactional. As we may perceive, the SB-Tree is based on B-Tree index, optimized concerning data insertion and range queries.

**2- Hash**: Super fast lookup, very light on disk. Works like a HashMap so it's faster on punctual lookup such (select from * from professor where discipline = math) and consumes less resources. Furthermore, hash index does not support range queries, but it's noticeable than SB-Tree index.

**3- Full Text:** Full Text indexes allow to index text as single word its radix. Full text indexes are like a search engine on database.

Syntax:

```
CREATE INDEX City.name ON City ( name ) FULLTEXT
```

**4- Lucene:** Good on full-text and spatial indexes. Lucene indexes can be used only for full-text and spatial.

- **Full text**: Full text index can be created using the OrientDB SQL syntax. Wherein it is needed to specify the index engine to use the Lucene full text capabilities.

Syntax:

```
CREATE INDEX <name> ON <class -name> (prop-names)
FULLTEXT ENGINE LUCENE
Example:
CREATE INDEX City.name ON
City (name) FULLTEXT
ENGINE LUCENE
```

- **Spatial**: For the moment, the Index Engine can only index Points.

Sintax:

```
CREATE INDEX Place.1_1on ON
Place ( latitude, longitude)
SPATIAL ENGINE LUCENE
FULLTEXT ENGINE LUCENE
```

## 6. CASE STYDY

The case study herein is a **Social network, Twitter** in the case, whence users, relationships between them, posts, and posts that they have selected as their favorites have been generated. To wit: 5,493 nodes (users), 16,479 properties, which took 788,636ms query time, 49,404 relationships (followers follow) 72,000,000ms query time to be created, in addition of 7,334 'favorites' relationships, statement executed in 1,345,026ms. The figures below illustrate both Neo4j and OrientDB generated graphs from imported and simulated Social Application datas.

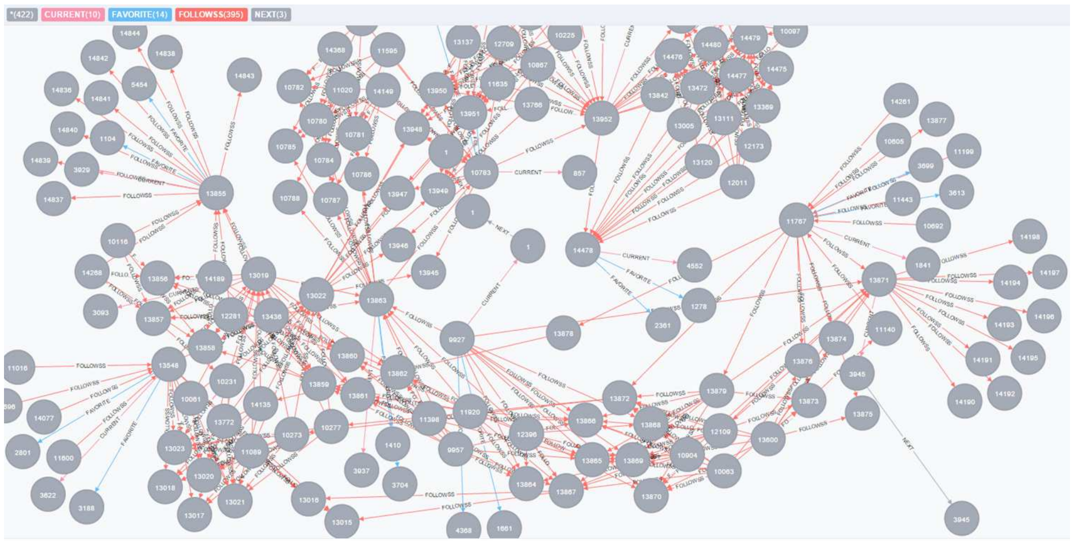

Figure 4. the generated graph in Neo4j

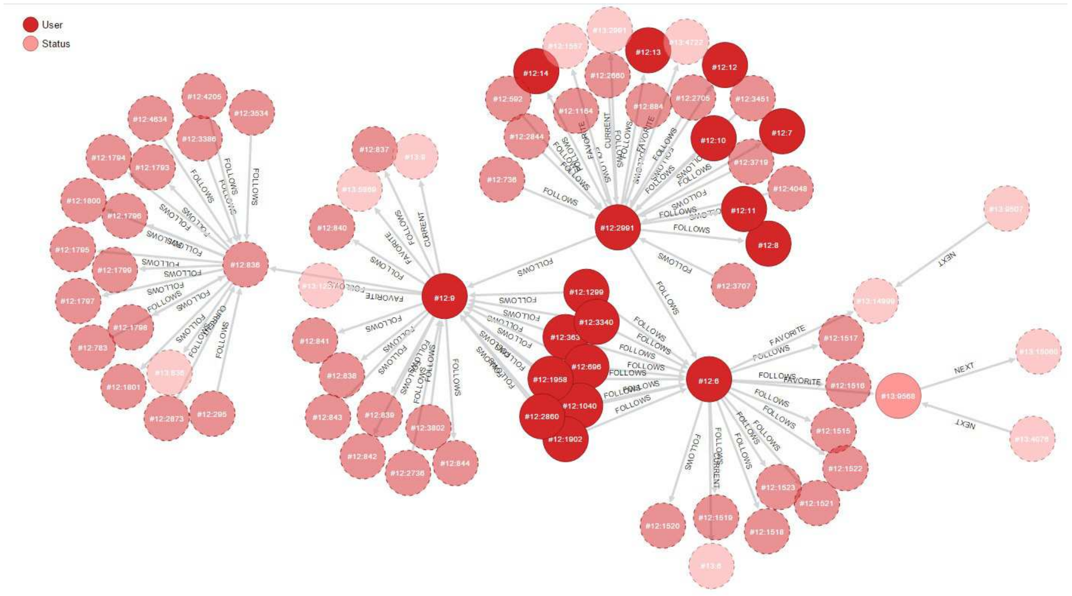

Figure 5. the generated graph in OrientDB

Throughout this case study, where there are 5.500 nodes (users) in a Social Network, one will query to find the node with the searched data, first without index (full scan), and second with index. The experiments below are performed into OrientDB and Neo4j, aiming at comparing the response time (statement query time) of the aforementioned graph databases and find out which of them has a better performance and in which case.

## 7. PERFORMANCE BASED EVALUATION

In this section, one evaluates the graph databases using two experiments that exercise the database with and without indexing.

Experiment setup:

•Performance evaluation was conducted on windows 8, with 4GB RAM and 2.60GHz dual core processor.
•Neo4j (community v2.1.5) and OrientDB (community v2.0.1).
•The test cases were run 10 times.
•Single node setup is used for both databases.

### Experiment I

In this experiment, it is performed five queries to wit: (1) select a user named "nadine", without index; (2) the same query as the first but with index instead; (3) select every single (all) user named "nadine"(but there were performed queries with other names as well), without index - in this case the full sacn is applied-; (4) the same query as the third but with index instead; (5) in the Social network, find the shortest path between to individuals, that is, form the individual **a** with path is the shortest to reach the individual **b**. Neo4j used to use either dijkstra or Floyd's algorithm. The main purpose here is to evaluate the performance of both Neo4j and OrientDB in the same context of data and queries with and without indexing.

## OrientDB

1. SELECT BOTH() FROM User WHERE username = 'nadine'
{Query time: 90 ~ 105 ms}

2. SELECT BOTH() FROM User WHERE @rid = #12:3630
{Query time: 5 ~ 7 ms}

3. SELECT * FROM User WHERE username = 'nadine'
{Query time: 100 ~ 108 ms}

4. SELECT * FROM User WHERE @rid = #12:3630
{Query time: 4 ~ 6 ms}

5. SELECT shortestPath (#12:3630, #12:5354, 'in', 'FOLLOWS')
{Query time: 6 ~ 10 ms}

## Neo4j

1. match ( user { username: 'nadine' }) < - [: FOLLOWS|: FAVORITE|: CURRENT]->(b) return b
{Query time: 45 ~ 48 ms}

2. match ( user {userId: 3631}) < - [:FOLLOWS|:FAVORITE|: CURRENT]->(b) return b
{Query time: 40 ~ 44 ms}

3. match ( n : User ) where n.username = 'nadine' return n
{Query time: 16 ~ 17 ms}

4. match ( n : User ) where n.userId = 3631 return n
{Query time: 7 ~ 8 ms}

5. match path = shortestPath ( ( User1) – [:FOLLOWS *..6] – (User2)) where User1.userId = 3631 and User2.userId = 5355 return path
{Query time: 80 ~ 90 ms}

## Experiment II

In this second experiments, there were a Breadth-first search, or nodes' traversal, with depth equals 5. Traversals are performed from a start node. To decide how the graph should be traversed. One can use whether breadth first or depth first parameters in the request body. As a further matter, it must be decided which relationship types and directions should be followed as well as how uniqueness should be calculated, in addition of whether the traverser should continue down that path or should be pruned so that the traverser will not continue down that path and the current position should be included in the result. The figure 6 shows an example of how the traversal is executed into Neo4j.

## Neo4j and OrientDB

Distinct friend of friends (10k people)

| Depth | OrientDB | Neo4-j |
|---|---|---|
| 1 | …. | …. |
| 2 | 0.01 | 0.016 |
| 3 | 0.168 | 0.267 |
| 4 | 1.359 | 2.506 |
| 5 | 2.132 | 3.210 |

Distinct friend of friends (100k people)

| Depth | OrientDB | Neo4-j |
|---|---|---|
| 1 | 0.010 | 0.010 |
| 2 | 0.010 | 0.018 |
| 3 | 0.072 | 0.138 |
| 4 | 2.544 | 1.600 |

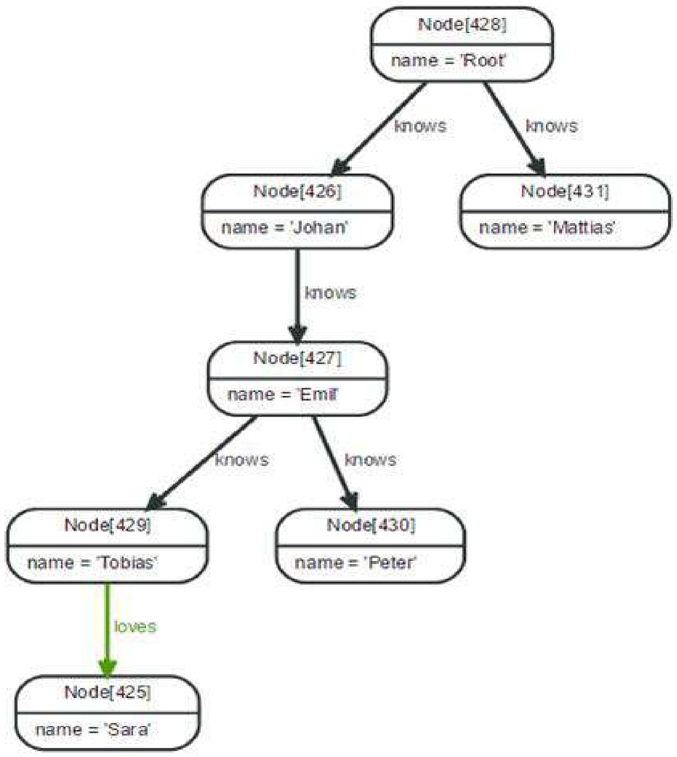

Figure 6. Example of Nodes' traversal [Neo4j 2015]

## 8. CONCLUSION

It is quite evident that the choice of a graph database depends on the application level consideration, as well as the data complexity, whereof several factors are verified. Since the graph is going to get so large overtime, it is recommended the use of indexing to make the search faster for a specific node whatsoever.

With the aim to evaluate, through indexing techniques, the performance of Neo4j and OrientDB, both graph databases technologies and to come up with strength and weaknesses, through the experiments, first of all it has been corroborated, for both technologies, that the whole point of having index speeded up search queries by essentially cutting down the number of nodes in a graph that need to be examined. Moreover, it should be remembered that the application worked in the best possible manner with the least waste of time and effort, considering the allocated resources presented in the experiment section.

Furthermore, comparing both Neo4j and OrientDB, one realized that without indexing techniques, Neo4j shows good queries (retrieval) performance in comparison to OrientDB. Notwithstanding, using indexing techniques, Neo4j's queries retrieval has less performance than OrientDB, especially on several number of nodes. Therefore, having a large number of nodes, whereby indexing techniques are necessary, the experiments have shown that it is recommended to use OrientDB.

## REFERENCES

[1] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. ACM Computing Surveys, 40(1). cited By 166.

[2] Bogdan, G. T. and Bucur, C. (2011). A comparison between several nosql databases with comments and notes. In Roedunet international Conference, pages 1–5.

[3] Brewer, E. (2000). Towards robust distribuited. In Proceedings of the 9th ACM Symposium on principles of distributed computing, New York, NY, USA. ACM.

[4] Bruggen, R. V. (2014). Learning Neo4j. PACKT, 1th edition.

[5] Han, W.-S., Lee, J., Pham, M.-D., and Yu, J. X. (2010). igraph: A framework for comparisons of disk-based graph indexing techniques. Proc. VLDB Endow., 3(1-2):449–459.

[6] Leonard, M. (2014).L'avenir du NoSQL. www.leonardmeyer.com/wpcontent/uploads/2014/06/avenirDuNoSQL.pdf, 1th edition.

[7] Macko, P., Margo, D., and Seltzer, M. (2013). Performance introspection of graph databases. In Proceedings of the 6th International Systems and Storage Conference, SYSTOR '13, pages 18:1–18:10, New York, NY, USA. ACM.

[8] Marek, C., Alex, A., and Ladialav, H. (2012). Benchmarking traversal operations over graph databases. IEEE, 12:1–5.

[9] Neo4j (2015). The Neo4j Manual v2.2.3. Neo4j, version 2.2.3 edition.

[10] OrientDB (2014). OrientDB official Manual. OrientDB, version 2.0 edition.

[11] Robinson, I., Webber, J., and Eifrem, E. (2013). Grapg Databases. O'REILLY, 1th edition.

[12] Steve, A., Luis, M., Marilde, S., and Marcela, R. (2015a). Graph database application using neo4j - railroad planner simulation. In ICEIS (1), pages 399–403.

[13] Steve, A., Patrick, B., and Luis, M. (2015b). From relational database to column-oriented nosql database: Migration process.