Fuzzy Queries over NoSQL Graph Databases: Perspectives for Extending the Cypher Language

Arnaud Castelltort and Anne Laurent

LIRMM, Montpellier, France
{castelltort,laurent}@lirmm.fr

Abstract. When querying databases, users often wish to express vague concepts, as for instance asking for the *cheap* hotels. This has been extensively studied in the case of relational databases. In this paper, we propose to study how such useful techniques can be adapted to NoSQL graph databases where the role of fuzziness is crucial. Such databases are indeed among the fastest-growing models for dealing with big data, especially when dealing with network data (e.g., social networks). We consider the Cypher declarative query language proposed for Neo4j which is the current leader on this market, and we present how to express fuzzy queries.

Keywords: Fuzzy Queries, NoSQL Graph Databases, Neo4j, Cypher, Cypherf.

1 Introduction

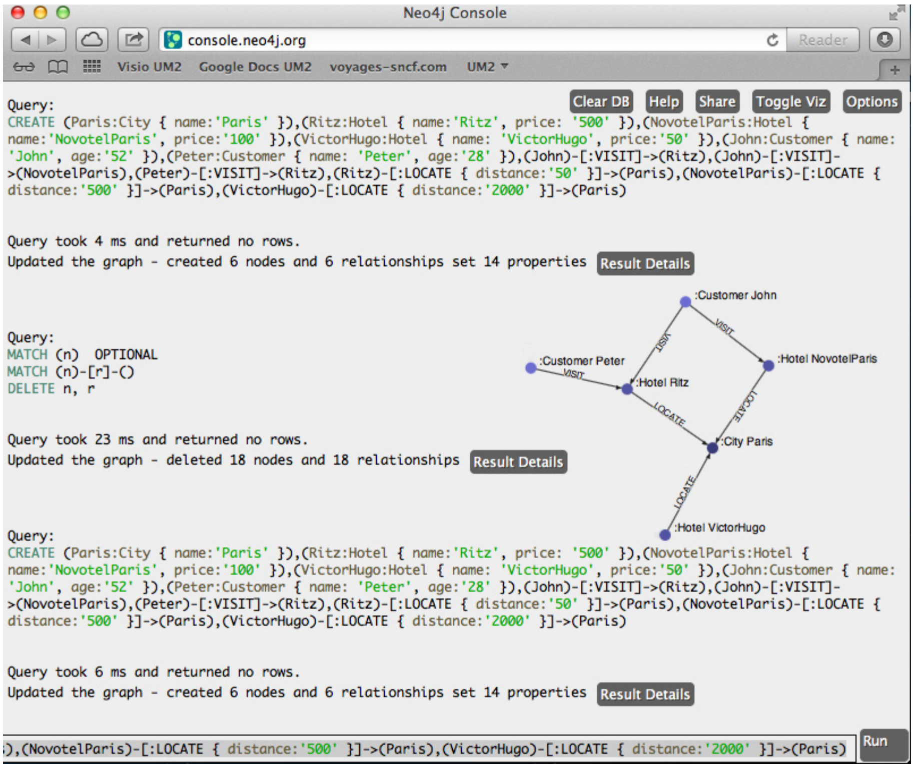
Graph databases have attracted much attention in the last years, especially because of the collaborative concepts of the Web 2.0 (social and media networks etc.) and the arriving Web 3.0 concepts.

Specific databases have been designed to handle such data relying on big dense network structures, especially within the NoSQL world. These databases are built to remain robust against huge volumes of data, against their heterogeneous nature and the high speed of the treatments applied to them, thus coping with the so-called Big Data paradigm.

They are currently gaining more and more interest and are applied in many real world applications, demonstrating their power compared to other approaches. NoSQL graph databases are known to offer great scalability [1].

Among these NoSQL graph databases, Neo4j appears to be one of the most mature and deployed [2]. In such databases, as for graphs, nodes and relationships between nodes are considered. Neo4j includes nodes and relationships labeling with the so-called types. Moreover, properties are attached to nodes and relationships. These properties are managed in Neo4j using the *key:value* paradigm.

Fig. 1 shows an example of hotels and customers database. The database contains hotels located in some cities and visited by some customers. Links are represented by the :LOCATED and :VISIT relationships. The hotels and people and relationships are described by properties: id, price, size (number of rooms) for hotels; id, name, age for people. One specificity is that relationships in Neo4j are provided with types (e.g., type "hotel" or "people" in the example) and can also have properties as for nodes. This allows to represent in a very intuitive and efficient manner many data from the real world. For instance, :LOCATED has property *distance*, standing for the distance to city center.



```
Query:
CREATE (Paris:City { name:'Paris' }),(Ritz:Hotel { name:'Ritz', price: '500' }),(NovotelParis:Hotel {
name:'NovotelParis', price:'100' }),(VictorHugo:Hotel { name: 'VictorHugo', price:'50' }),(John:Customer { name:
'John', age:'52' }),(Peter:Customer { name: 'Peter', age:'28' }),(John)-[:VISIT]->(Ritz),(John)-[:VISIT]-
>(NovotelParis),(Peter)-[:VISIT]->(Ritz),(Ritz)-[:LOCATE { distance:'50' }]->(Paris),(NovotelParis)-[:LOCATE {
distance:'500' }]->(Paris),(VictorHugo)-[:LOCATE { distance:'2000' }]->(Paris)

Query took 4 ms and returned no rows.
Updated the graph - created 6 nodes and 6 relationships set 14 properties

Query:
MATCH (n)  OPTIONAL
MATCH (n)-[r]-()
DELETE n, r

Query took 23 ms and returned no rows.
Updated the graph - deleted 18 nodes and 18 relationships

Query:
CREATE (Paris:City { name:'Paris' }),(Ritz:Hotel { name:'Ritz', price: '500' }),(NovotelParis:Hotel {
name:'NovotelParis', price:'100' }),(VictorHugo:Hotel { name: 'VictorHugo', price:'50' }),(John:Customer { name:
'John', age:'52' }),(Peter:Customer { name: 'Peter', age:'28' }),(John)-[:VISIT]->(Ritz),(John)-[:VISIT]-
>(NovotelParis),(Peter)-[:VISIT]->(Ritz),(Ritz)-[:LOCATE { distance:'50' }]->(Paris),(NovotelParis)-[:LOCATE {
distance:'500' }]->(Paris),(VictorHugo)-[:LOCATE { distance:'2000' }]->(Paris)

Query took 6 ms and returned no rows.
Updated the graph - created 6 nodes and 6 relationships set 14 properties

),(NovotelParis)-[:LOCATE { distance:'500' }]->(Paris),(VictorHugo)-[:LOCATE { distance:'2000' }]->(Paris)
```

Fig. 1. Neo4j database console user interface: Example for Hotels and Customers

All NoSQL graph databases require the developers and users to use graph concepts to query data. As for any other repository, when querying such NoSQL graph databases, users either require specific focused knowledge (e.g., retrieving Peter's friends) or ask for trend detection (e.g., detecting trends and behaviours within social networks).

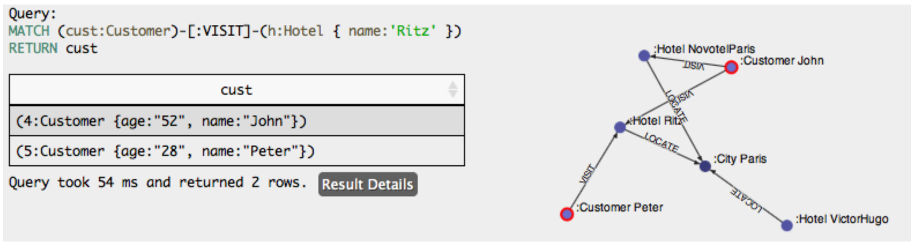
Queries are called *traversals*. A graph traversal refers to visiting elements, *i.e.* nodes and relations. There are three main ways to traverse a graph:

- programmaticaly, by the use of an API that helps developers to operate on the graph;
- by functional traversal, a traversal based on a sequence of functions applied to a graph;
- by declarative traversal, a way to explicit what we want to do and not how we want to do it. Then, the database engine defines the best way to achieve the goal.

In this paper, we focus on declarative queries over a NoSQL graph database. The Neo4j language is called Cypher.

For instance on Fig. 1, one query is displayed to return the customers who have visited the “Ritz” hotel.They are both displayed in the list and circled in red in the graph.

We consider in this paper the manipulating queries in READ mode.



```
Query:
MATCH (cust:Customer)-[:VISIT]-(h:Hotel { name:'Ritz' })
RETURN cust
```

cust
(4:Customer {age:"52", name:"John"})
(5:Customer {age:"28", name:"Peter"})

```
Query took 54 ms and returned 2 rows.
```

Fig. 2. Displaying the Result of a Cypher Query

However, none of the query languages embeds a way for dealing with flexible queries, for instance to get *cheap* hotels or *popular* ones, where *cheap* and *popular* are fuzzy sets.

This need has nevertheless been intensively studied when dealing with other database paradigms, especially with relational databases.

In this paper, we thus focus on the declarative way of querying the Neo4j system with the Cypher query language and we extend it for dealing with vague queries.

The rest of the paper is organised as follows. Section 2 reports existing work from the literature regarding fuzzy queries and presents the Cypher language. Section 3 introduces the extension of the Cypher language to Cypherf and Section 4 shows how such an extension can be implemented. Section 5 concludes the paper and provides some ideas for future work.

2 Related Work

2.1 Neo4j Cypher Language

Queries in Cypher have the following syntax[1]:

```
[START]
[MATCH]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

[1] http://docs.neo4j.org/refcard/2.0/
http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html

As shown above, Cypher is comprised of several distinct clauses:

– START: Starting points in the graph, obtained via index lookups or by element IDs.
– MATCH: The graph pattern to match, bound to the starting points in START.
– WHERE: Filtering criteria.
– RETURN: What to return.
– CREATE: Creates nodes and relationships.
– DELETE: Removes nodes, relationships and properties.
– SET: Set values to properties.
– FOREACH: Performs updating actions once per element in a list.
– WITH: Divides a query into multiple, distinct parts.

2.2 Fuzzy Queries

Many works have been proposed for dealing with fuzzy data and queries. All cannot be reported here. [3] proposes a survey of these proposals.

[4,5] consider querying regular databases by both extending the SQL language and studying aggregating subresults. The FSQL/SQLf and FQL languages have been proposed to extend queries over relational databases in order to incorporate fuzzy descriptions of the information being searched for.

Some works have been implemented as fuzzy database engines and systems have incorporated such fuzzy querying features [6,7].

In such systems, fuzziness in the queries is basically associated to fuzzy labels, fuzzy comparators (e.g., *fuzzy greater than*) and aggregation over clauses. Thresholds can be defined for the expected fulfillment of fuzzy clauses.

For instance, on a crisp database describing hotels, users can ask for $cheap$ hotels that are $close\_to\_city\_center$, $cheap$ and $close\_to\_city\_center$ being fuzzy labels described by fuzzy sets and their membership functions respectively defined on the universe of prices and distance to the city center.

Many works have been proposed to investigate how such fuzzy clauses can be defined by users and computed by the database engine, especially when several clauses must be merged (e.g., $cheap$ **AND** $close\_to\_city\_center$).

Such aggregation can consider preferences, for instance for queries where price is prefered to distance to city center using weighted t-norms.

Thresholds can be added for working with $\alpha-$cuts, such as searching for hotels where the degree $cheap$ is greater than 0.7.

As we consider graph data, the works on fuzzy ontology querying are very close and relevant for us [8,9].

[8] proposes the f-SPARQL query language that supports fuzzy querying over ontologies by extending the SPARQL language. This extension is based on threshold query (e.g., asking for people who are *tall* at a degree greater than 0.7) or general fuzzy queries based on semantic functions.

It should be noted that many works have dealt with fuzzy databases for representing and storing imperfect information in databases: fuzzy ER models, fuzzy object databases, fuzzy relational databases, fuzzy ontologies-OWL [10], etc. Fuzziness can then impact many levels, from metadata (attributes) to data (tuples), and cover many semantics (uncertainty, imprecision, inconsistency, etc.) as recalled in [3]. These works are not reported here as we consider fuzzy queries over crisp data.

3 Fuzzy Queries over NoSQL Graph databases: Towards the Cypherf Language

In this paper, we address fuzzy READ queries over regular NoSQL Neo4j graph databases. We claim that fuzziness can be handled at the following three levels:

- over properties,
- over nodes,
- over relationships.

3.1 Cypherf over Properties

Dealing with fuzzy queries over properties is similar to the queries from the literature on relational databases and ontologies. Such queries are defined by using linguistic labels (fuzzy sets) and/or fuzzy comparators.

Such fuzzy queries impact the $START$, $MATCH$, $WHERE$ and $RETURN$ clauses from Cypher.

In the $WHERE$ clause, it is then possible to search for *cheap* hotels in some databases, or for hotels located $close\_to\_city\_center$[2]. Note that these queries are different as the properties being addressed are respectively linked to a node and a relationship.

Listing 1.1. Cheap Hotels

```
MATCH (h:Hotel)
WHERE CHEAP(price) > 0
RETURN h
ORDER BY CHEAP(h) DESC
```

Listing 1.2. Hotels Close to City Center

```
MATCH (c:City)<-[:LOCATED]-(h:Hotel)
WHERE CLOSE(c,h) > 0
RETURN h
ORDER BY CLOSE(c,h) DESC
```

In the $START$ clause, it is possible to define which nodes and relationships to start from by using fuzzy labels, as for instance:

Listing 1.3. Starting from Cheap Hotels

```
START h:Hotel(CHEAP(price) > 0)
RETURN h
ORDER BY CHEAP(h) DESC
```

[2] For the sake of simplicity, the fuzzy labels and membership functions are hereafter denoted by the same words.

Listing 1.4. Starting from location links close to city center

```
START l=relationship:LOCATED(CLOSE(distance)>0)
MATCH (h:Hotel) -[ :LOCATED]->(c:City)
RETURN h
ORDER BY CLOSE(h,c) DESC
```

In the $MATCH$ clause, integrating fuzzy labels is also possible:

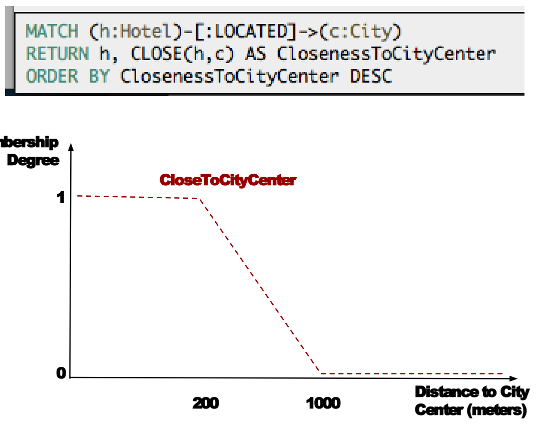
Listing 1.5. Matching Hotels Close to City Center

```
MATCH (h:Hotel) -[ :LOCATED {CLOSE(distance)>0}]->(c:City)
RETURN h
ORDER BY CLOSE(h,c) DESC
```

In the $RETURN$ clause, no selection will be operated, but fuzzy labels can be added in order to show the users the degree to which some values match fuzzy sets, as for instance:

Listing 1.6. Fuzziness in the Return Clause

```
MATCH (h:Hotel) -[ :LOCATED]->(c:City)
RETURN h, CLOSE(h,c) AS 'ClosenessToCityCenter'
ORDER BY ClosenessToCityCenter DESC
```



```
MATCH (h:Hotel)-[:LOCATED]->(c:City)
RETURN h, CLOSE(h,c) AS ClosenessToCityCenter
ORDER BY ClosenessToCityCenter DESC
```

Fig. 3. Fuzzy Cypher Queries: an Example

When considering fuzzy queries over relational databases, the results are listed and can be ranked according to some degrees. When considering graph data, graphical representations are of great interest for the user comprehension and interaction on the data. For instance, Fig 2 shows how a result containing two items (the two customers who went to Ritz hotel) is displayed in the Cypher console, demonstrating the interest of the graphic display.

It would thus be interesting to investigate how fuzzy queries over graph may be displayed, showing the graduality of membership of the objects to the result. For this purpose, we propose to use the work from the literature on fuzzy graph representation and distored projection as done in anamorphic maps [11].

3.2 Cypherf over Nodes

Dealing with fuzzy queries over nodes allows to retrieve similar nodes. It is set at a higher level from queries over properties although it may use the above-defined queries.

For instance, it is possible to retrieve similar hotels:

Listing 1.7. Getting Similar Hotel Nodes

```
MATCH (h1:Hotel),(h2:Hotel)
WITH h1 AS hot1, h2 AS hot2, SimilarTo(hot1,hot2) AS sim
WHERE sim > 0.7
RETURN hot1,hot2,sim
```

In this framework, the link between nodes is based on the definition of measures between the descriptions. Such measures integrate aggregators to deal with the several properties they embed. Similarity measures may for instance be used and hotels may all the more be considered as their prices and size are similar.

It should be noted that such link could be materialized by relationships, either for performance concerns, or because it was designed this way. In the latter case, such query amounts to a query as defined above.

3.3 Cypherf over Relationships

As for nodes, such queries may be based on properties. But it can also be based on the graph structure in order to better exploit and benefit from it.

In Cypher, the structure of the pattern being searched is mostly defined in the $MATCH$ clause.

The first attempt to extend pattern matching to fuzzy pattern matching is to consider chains and depth matching. Chains are defined in Cypher in the $MATCH$ clause with consecutive links between objects. If a node a is linked to an object b at depth 2, the pattern is writen as $(a) - [*2]- > (b)$. If a link between a and b without regarding the depth in-between is searched, then it is writen $(a) - ()- > (b)$. The mechanism also applies for searching objects linked trough a range of nodes (e.g., between 3 and 5): $(a) - [*3..5]- > (b)$.

We propose here to introduce fuzzy descriptors to define extended patterns where the depth is imprecisely described. It will then for instance be possible to search for customers linked through *almost 3* hops. The syntax $**$ is proposed to indicate a fuzzy linker.

Listing 1.8. Fuzzy Patterns

```
MATCH (c1:customer)-[:KNOWS**almost3]->(c2:customer)
RETURN c1,c2
```

It is related to fuzzy tree and graph mining [12] where some patterns emerge from several graphs even they do not occur exactly the same way everywhere regarding the structure.

Another possibility is not to consider chains but patterns where several links from and to nodes.

In our running example, *popular* hotels may for instance be chosen when they are chosen by many people. This is similar as the way famous people are detected if they are followed by many people on social networks.

In this example, a hotel is popular if a large proportion of customers visited it.

In Cypher, such queries are defined by using aggregators. For instance, the following query retrieves hotels visited by at least 2 customers:

Listing 1.9. Aggregation

```
MATCH (c:Customer)-[:VISIT]->(h:Hotel)
WITH c AS cust, count(*) AS cpt
WHERE cpt>1
RETURN cust
```

Such crisp queries can be extended to consider fuzziness:

Listing 1.10. Aggregation

```
MATCH (c:Customer)-[:VISIT]->(h:Hotel)
WITH c AS cust, count(*) AS cpt
WHERE POPULAR(cpt) > 0
RETURN cust
```

All fuzzy clauses described in this section can be combined. The question then risen is to implement them in the existing Neo4j engine.

4 Implementation Challenges

4.1 Architecture

There are several ways to implement fuzzy Cypher queries:

1. Creating an overlay language on top of the Cypher language that will produce as ouput Cypher well formatted queries to do fuzzy work;
2. Extending the Cypher queries and using the existing low level API behind;
3. Extending the low level API with optimized functions, offering the possibility only to developpers to use it;
4. Combining the last two possibilities: using an extended cypher query language over an enhanced low level API.

Every possibility is debated in this section. The reader will find at the end of this section a summary of the debates.

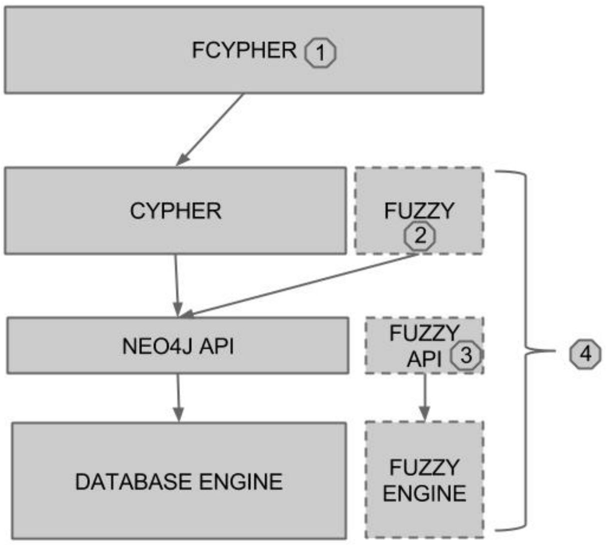


Fig. 4. Implementation Ways

4.2 Creating an Overlay Language

Concept. The concept is to create a high-level fuzzy DSL language that will be used to generate Cypher well-formed queries. The generated Cypher queries will be executed by the existing Neo4j engine.

A grammar must be defined for this external DSL which can rely on the existing Cypher syntax and only enhance it with new fuzzy features. The output of the generation process is pure Cypher code. In this scenario, Cypher is used as a low level language to achieve fuzzy queries.

Discussion. This solution is a cheap and non intrusive solution but has several huge drawbacks:

- Features missing, indeed every fuzzy query shown in Section 3 cannot be expressed by the current cypher language (e.g., listing 1.4);
- Performance issue, Cypher is not designed for fuzzy queries neither for being used as an algorithmic language. All the fuzzy queries will produce Cypher query codes that are not optimized for fuzzy tasks;
- Lack of user-friendliness, Each query cannot be executed directly against the Neo4j environnement, it needs a two-step process: (i) write a fuzzy query, then compile it to get the cypher query; (ii) use the cypher generated queries on the Neo4j database

4.3 Extending the Cypher Queries

Concept. The idea is to extend the Cypher language to add new features. Cypher offers various types of functions: scalar functions, collection functions, predicate functions, mathematical functions, etc. To enhance this language with fuzzy features, we propose to add a new type of functions: fuzzy functions. Fuzzy functions are used in the same way as other functions of Cypher (or SQL) as shown in section 3.

Cypher is an external DSL. Therefore, somewhere it needs to be parsed. The query correctness must be checked and then it should be executed. In the Cypher case, retrieving the results we asked for.

In order to write Cypher, the Neo4j's team had defined its grammar, which gives the guidelines of how the language is supposed to be structured and what is and isnt valid. In order to express this definition, we can use some variation of EBNF syntax [13], which provides a clear way to expose the language definition. To parse this syntax, Cypher uses Scala language Parser Combinator library.

Then, to extend the Cypher engine, the Cypher grammar must be extended regarding the current grammar parser. Once the cypher query is parsed, the code has to be bound on the current programmatic API to achieve the desired result.

Discussion. This work needs a deeper comprehension of the Neo4j engine and more skills on Java/Scala programming language (used to write the Neo4j engine and API) than the previous solutions. The main advantage of this is to offer an easy and user-friendly way to use the fuzzy feature. The disavantages of this solution are:

- Performance issue. This solution should have better performance than the previous one but it stills built on the current Neo4j engine API that is not optimized for fuzzy queries (e.g., degree computing);
- Cost of maintenance. Until Neo4j accepts to inlude this contribution to the Neo4j project, it will be needed to upgrade each new version of Neo4j with these enhancements. If this feature is built in a plugin, it will be necessary to check that the API has not been broken by the new version (if so an upgrade of the fuzzy plugin will be required).

4.4 Extending Low Level API

Concept The scenario is to enhance the core database engine with a framework to handle efficiently the fuzzy queries and to extend the programming API built on it to provide to developpers access to this new functionnality.

Discussion. This solution offers a high performance improvment but needs high Neo4j skills, possibly high maintenance costs, a poor user friendly experience (only developpers can use it) and a costly development process.

4.5 Extending Cypher over an Enhanced Low Level API

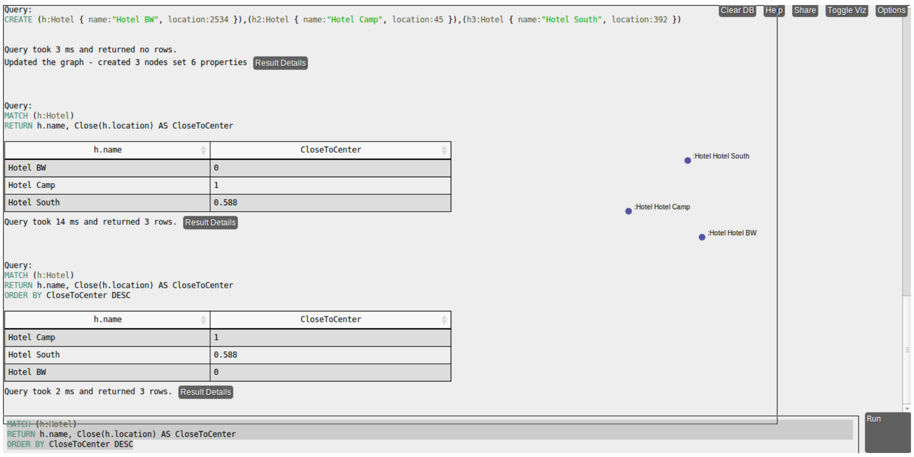
Concept. The last and not the least possibility is to combine the solutions from Sections 4.3 and 4.4: adding to the database engine the feactures to handle the fuzzy queries, extending the API and extending the Cypher language.

Discussion. This solution is user-friendly, provides optimized performance but has a heavy development cost (skills, tasks, etc.) and a high cost of maintenance.

4.6 Summary and Prototype

The first solution is a non intrusive solution with limited perspectives. It is more a hack than a real long termes solution. The best, but most costly, solution still the last one: extend cypher query language and build a low level API framework to extend the Neo4j database engine to support such kind of queries.

A prototype based on the extension of cypher over an enhanced API is under developement, fuzzy queries can be run, as shown in Fig. 5.



```
Query:
CREATE (h:Hotel { name:"Hotel BW", location:2534 }),(h2:Hotel { name:"Hotel Camp", location:45 }),(h3:Hotel { name:"Hotel South", location:392 })

Query took 3 ms and returned no rows.
Updated the graph - created 3 nodes set 6 properties  Result Details

Query:
MATCH (h:Hotel)
RETURN h.name, Close(h.location) AS CloseToCenter
```

h.name	CloseToCenter
Hotel BW	0
Hotel Camp	1
Hotel South	0.588

```
Query took 14 ms and returned 3 rows.  Result Details

Query:
MATCH (h:Hotel)
RETURN h.name, Close(h.location) AS CloseToCenter
ORDER BY CloseToCenter DESC
```

h.name	CloseToCenter
Hotel Camp	1
Hotel South	0.588
Hotel BW	0

```
Query took 2 ms and returned 3 rows.  Result Details

MATCH (h:Hotel)
RETURN h.name, Close(h.location) AS CloseToCenter
ORDER BY CloseToCenter DESC
```

Fig. 5. Protoype Developed

5 Conclusion

In this paper, we propose an extension of the declarative NoSQL Neo4j graph database query language (Cypher). This language is applied on large graph data which represent one of the challenges for dealing with big data when considering social networks for instance. A protoype has been developed and is currently being enhanced.

As we consider the existing Neo4j system which is efficient, performance is guaranteed. The main property of NoSQL graph databases, i.e. the optimized $O(1)$ low complexity for retrieving nodes connected to a given one, and the efficient index structures ensure that performances are optimized.

Future works include the extension of our work to the many concepts possible with fuzziness (e.g., handling fuzzy modifiers), the study of fuzzy queries over historical NoSQL graph databases as introduced in [14] and the study of definition fuzzy structures: Fuzzy Cypher queries for Data Definition or in WRITE mode (e.g., inserting imperfect data). The implementation of the full solution relying on our work, currently in progress, will be completed by these important extensions.

References

1. Rodriguez, M.A., Neubauer, P.: The graph traversal pattern. CoRR abs/1004.1001 (2010)
2. Board, T.T.A.: Technology radar (May 2013), `http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf`
3. Meng, X., Ma, Z., Zhu, X.: A knowledge-based fuzzy query and results ranking approach for relational databases. Journal of Computational Information Systems 6(6) (2010)
4. Bosc, P., Pivert, O.: Sqlf: a relational database language for fuzzy querying. IEEE Transactions on Fuzzy Systems 3(1), 1–17 (1995)
5. Takahashi, Y.: A fuzzy query language for relational databases. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1576–1579 (1991)
6. Zadrożny, S., Kacprzyk, J.: Implementing fuzzy querying via the internet/WWW: Java applets, activeX controls and cookies. In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.) FQAS 1998. LNCS (LNAI), vol. 1495, pp. 382–392. Springer, Heidelberg (1998)
7. Galindo, J., Medina, J.M., Pons, O., Cubero, J.C.: A server for fuzzy SQL queries. In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.) FQAS 1998. LNCS (LNAI), vol. 1495, pp. 164–174. Springer, Heidelberg (1998)
8. Pan, J.Z., Stamou, G.B., Stoilos, G., Taylor, S., Thomas, E.: Scalable querying services over fuzzy ontologies. In: Huai, J., Chen, R., Hon, H.W., Liu, Y., Ma, W.Y., Tomkins, A., Zhang, X. (eds.) WWW, pp. 575–584. ACM (2008)
9. Cheng, J., Ma, Z.M., Yan, L.: f-SPARQL: A flexible extension of SPARQL. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part I. LNCS, vol. 6261, pp. 487–494. Springer, Heidelberg (2010)
10. Stoilos, G., Stamou, G.B., Tzouvaras, V., Pan, J.Z., Horrocks, I.: Fuzzy owl: Uncertainty and the semantic web. In: Grau, B.C., Horrocks, I., Parsia, B., Patel-Schneider, P.F. (eds.) OWLED. CEUR Workshop Proceedings, vol. 188. CEUR-WS.org (2005)
11. Griffin, T.: Cartographic transformations of the thematic map base. Cartography 11(3), 163–174 (1980)
12. López, F.D.R., Laurent, A., Poncelet, P., Teisseire, M.: Ftmnodes: Fuzzy tree mining based on partial inclusion. Fuzzy Sets and Systems 160(15), 2224–2240 (2009)
13. Pattis, R.: Teaching ebnf first in cs 1. In: Beck, R., Goelman, D. (eds.) SIGCSE, pp. 300–303. ACM (1994)
14. Castelltort, A., Laurent, A.: Representing history in graph-oriented nosql databases: A versioning system. In: Proc. of the Int. Conf. on Digital Information Management (2013)