



Castelltort, A., & Martin, T. (2017). Handling scalable approximate queries over NoSQL graph databases: Cypherf and the Fuzzy4S framework. *Fuzzy Sets and Systems*. <https://doi.org/10.1016/j.fss.2017.08.002>

Peer reviewed version

License (if available):
CC BY-NC-ND

Link to published version (if available):
[10.1016/j.fss.2017.08.002](https://doi.org/10.1016/j.fss.2017.08.002)

[Link to publication record in Explore Bristol Research](#)
PDF-document

This is the author accepted manuscript (AAM). The final published version (version of record) is available online via Elsevier at <https://www.sciencedirect.com/science/article/pii/S0165011417303093?via%3Dihub> . Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available:
<http://www.bristol.ac.uk/pure/about/ebr-terms>

Handling Scalable Approximate Queries over NoSQL Graph Databases: Cypherf and the Fuzzy4S Framework

Arnaud Castellort¹, Trevor Martin²

¹ *LIRMM, CNRS-University of Montpellier, France*

² *Department of Engineering Mathematics, University of Bristol, UK*

Abstract

NoSQL databases are currently often considered for Big Data solutions as they offer efficient solutions for volume and velocity issues and can manage some of complex data (e.g., documents, graphs). Fuzzy approaches are yet often not efficient on such frameworks. Thus this article introduces a novel approach to define and run approximate queries over NoSQL graph databases using Scala by proposing the Fuzzy4S framework and the Cypherf fuzzy declarative query language. NoSQL Graph databases are currently gaining more and more interest and are applied in many real world applications. The Fuzzy4S framework is defined with an open DSL (Domain Specific Language) allowing it to define scalable approximate queries at an abstract level. Cypherf is an extension of Cypher which runs over the Neo4J NoSQL graph databases. This work consists of a complete approach embedding the whole chain from end-user declarative query level to implementation issues within the database engine. We provide both the formal definitions for defining approximate graph NoSQL queries and the experimental results which demonstrate the interest and efficiency of our proposition.

Keywords: Approximate Queries, Fuzzy Logic, NoSQL Graph Databases.

1. Introduction

Graphs are often a natural way to represent the world [1]. This is especially true for social networks, but also in biology, chemistry etc. In such areas,

1
2
3
4
5
6
7
8
9 relationships between objects are at least as important as the objects themselves.

10
11 5 For example, retrieving "friend" and "friend of friend" relationships is a key
12 operation in social networks or in any other human relations. In [2], we have
13 highlighted the power of the study of relations in order to retrieve fraud rings
14 which are defined as *sophisticated chains of indirect links between fraudsters*
15 *representing successive transactions (money, communications, etc.) from which*
16 *rogue behaviours are detected.*
17
18
19 10

20 Graphs have been extensively studied in the literature and have recently
21 gained attention with the development of Semantic Web and ontologies. Many
22 modelizations, tools and frameworks have been suggested to represent, manage
23 and analyze graphs with XML, OWL etc.

24
25
26 15 At the same time, the volume and complexity of information concerning
27 these objects and their relationships are growing dramatically, leading to huge
28 databanks.
29
30

31 This has led to the development of the so-called NoSQL graph databases
32 that embed both the performance of NoSQL databases and the representativity
33 of graphs. Several engines exist (OrientDB, Neo4J, HyperGraphDB etc.) [3]. In
34 20 of this article, Neo4J is used since it has been considered the top graph database
35 management for several years¹ (Figure 1, [4]²).
36
37
38

39 Such database management systems embed query processing that can be
40 used at different levels, from the declarative level to the programmatic level
41 within the engine. The Neo4J system offers the declarative Cypher language
42 25 that allows the users to query the graph in a very intuitive manner. These
43 queries include clauses over both nodes and relationships. Properties can be
44 defined in the same style that NoSQL databases *i.e.* with (*key : value*) lists.
45 The value can be of any type, including collections.
46
47
48
49

50 30 For instance, it is possible to consider nodes representing people having
51 some properties such as age (*key = age, value = 28*) or hobbies (*key = hobby,*
52
53

54
55 ¹<http://db-engines.com/en/ranking/graph+dbms>

56 ²<http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf>, 2013
57
58

27 systems in ranking, July 2017

Rank	Rank			DBMS	Database Model	Score		
	Jul 2017	Jun 2017	Jul 2016			Jul 2017	Jun 2017	Jul 2016
1.	1.	1.		Neo4j	Graph DBMS	38.52	+0.65	+4.83
2.	2.	4.		Microsoft Azure Cosmos DB	Multi-model	7.71	+1.33	+5.29
3.	3.	2.		OrientDB	Multi-model	5.57	-0.24	-0.26
4.	4.	3.		Titan	Graph DBMS	4.93	-0.33	-0.34
5.	5.	6.		ArangoDB	Multi-model	2.96	-0.12	+1.21
6.	6.	5.		Virtuoso	Multi-model	1.99	-0.01	-0.31
7.	7.	7.		Giraph	Graph DBMS	1.05	+0.00	+0.16
8.	8.	8.		AllegroGraph	Multi-model	0.61	+0.00	+0.14
9.	9.	9.		Stardog	Multi-model	0.55	+0.01	+0.09
10.	10.	12.		GraphDB	Multi-model	0.53	+0.01	+0.36
11.	11.	10.		Sqrrl	Multi-model	0.49	+0.02	+0.24
12.	16.			Graph Engine	Multi-model	0.36	+0.10	
13.	13.	11.		InfiniteGraph	Graph DBMS	0.29	-0.01	+0.10
14.	15.	18.		Blazegraph	Multi-model	0.29	-0.00	+0.21
15.	14.			Dgraph	Graph DBMS	0.28	-0.02	
16.	19.			JanusGraph	Graph DBMS	0.23	+0.11	
17.	21.	16.		Sparksee	Graph DBMS	0.16	+0.05	+0.07
18.	17.	14.		FlockDB	Graph DBMS	0.16	-0.03	+0.03
19.	18.	17.		HyperGraphDB	Graph DBMS	0.15	-0.03	+0.07
20.	20.	15.		InfoGrid	Graph DBMS	0.13	+0.01	+0.04

Figure 1: Comparison of Graph Database Management Systems (db-engines.com)

$value = \{tennis, reading, travel\}$).

In some cases, it is not relevant for users to define queries in a “crisp” way, as for instance to retrieve people having friends aged between 20 and 30 but they rather want to retrieve people even if their age is 31 and not strictly between 20 and 30.

For this reason, we consider approximate queries over such numerical properties. Such queries can for instance be expressed using the fuzzy set theory framework. In our proposition we introduce Fuzzy4S, standing for *Fuzzy for Scala*. Scala is a modern language mixing object-oriented and functional programming [5]. Fuzzy4S is used within NoSQL databases to define fuzzy queries by introducing the Cypherf framework (which stands for *Cypher fuzzy*).

The rest of this article is organized as follows: Section 2 presents the preliminary concepts underlying our work: graph databases and approximate queries. Section 3 introduces the extension of graph NoSQL queries with the Cypherf framework and discusses implementation issues. Cypherf relies on the Fuzzy4S framework which is detailed in Section 4. Section 5 illustrates our contributions

1
2
3
4
5
6
7
8
9 using examples and showing the efficiency of our implementation with experi-
10 mental load tests over benchmarks. Section 6 concludes this paper and discusses
11 some of the many research avenues opened by our work.
12
13
14

15 **2. Preliminary Concepts and Properties**

16
17 This work merges two topics, namely graph oriented NoSQL databases and
18 approximate queries that are briefly introduced below.
19

20 For illustrating these concepts and in the rest of this paper, we rely on an
21
22 55 example where customers are linked to the hotels they have visited. A hotel is
23 described with properties such as the price and location with regards to the city
24 center. Customers are described by their age.
25
26
27

28 *2.1. Graph Databases*

29
30 Graphs have been studied for a long time by mathematicians and computer
31
32 60 scientists. A graph can be directed or not, labeled or not. Graph databases rely
33 on labeled directed graphs.
34

35
36 **Definition 1 (Labeled Directed Graph).** *A labeled oriented graph G , also*
37 *known as oriented property graph, is given by a n -tuple $(V, E, \alpha, \beta, l_V, l_E)$ where*
38 *V stands for a set of nodes and E stands for a set of edges with $E \subseteq (V \times V)$,*
39 *α stands for the set of attributes defined over the nodes, β the set of attributes*
40 *defined over the relations, $l_V : V \rightarrow \mathcal{P}(\alpha)$ is the labeling function for nodes and*
41 *$l_E : E \rightarrow \mathcal{P}(\beta)$ is the labeling function for relationships.*
42
43
44
45

46 In such databases, α and β are not simple labels such as in labeled graphs.
47 They are rather defined by $(key, value)$ pairs commonly used by NoSQL databases [6]
48
49 70 to represent the so-called properties over nodes and relations. Keys can occur
50 in several nodes or relations.
51

52 Figure 2 shows a graph and its anatomy in $(key, value)$ pairs.
53

54 Queries in graph databases are defined as traversals over the graph. They
55 can be run at several levels, either programmatically or by using declarative
56
57
58

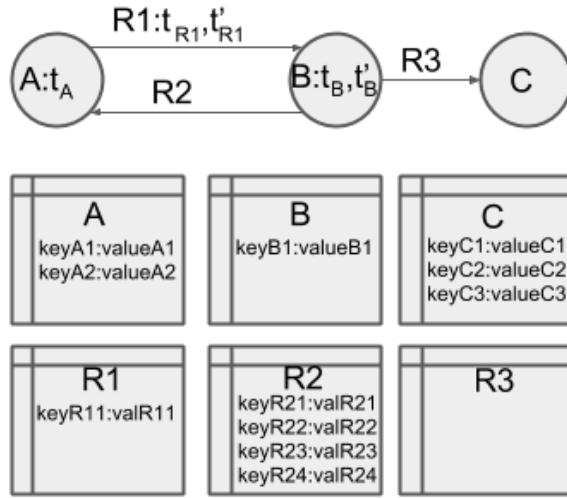


Figure 2: Node and Relationship Properties

75 languages (as done when using SQL in relational databases). No common set
 31 of operators has yet been defined as a standard. It should be noted that the
 32 result of a query over a graph is not always defined as a graph.

33 In Neo4J, declarative queries are run using the so-called *Cypher* language.

34 2.2. Neo4J Cypher Language

35 In Cypher, queries are formulated using the following syntax³:

```

41 [MATCH]
42 [OPTIONAL MATCH WHERE]
43 [WITH [ORDER BY] [SKIP] [LIMIT]]
44 RETURN [ORDER BY] [SKIP] [LIMIT]
45
46
47

```

48 85 Queries are formalized through EBNF Cypher grammar, as shown below.

For the sake of simplicity, we only show some extracts of the grammar.

53 ³<http://neo4j.com/docs/2.2.5/cypher-refcard/>
 54 <http://neo4j.com/docs/2.2.5/cypher-query-lang.html>

1
2
3
4
5
6
7
8
9 An EBNF grammar [7] is an extension of BNF (Backus-Naur Form (BNF) [8])
10 for formalizing the syntax of a language. It has been extensively studied in soft-
11 ware engineering research works. Roughly speaking, every EBNF grammar is
12 described by means of rules, every rule being defined with a left and right part.
13
14 90

15 An open community works on the openCypher project⁴ that aims to deliver
16 a full and open specification of Cypher and to evolve the language through the
17 production and acceptance of CIPs (Cypher Improvement Proposals). Our work
18 is part of this trend and aims to contribute with fuzzy extensions.
19
20
21

22
23 95 **EBNF Grammar Extract 1.** *Cypher requests are based on sequences of state-*
24 *ments that are regular queries. Regular queries are based on single queries that*
25 *can be combined through unions. Single queries are based on 9 possible clauses:*
26 *Match, Unwind, Merge, Create, Set, Delete, Remove, With, Return.*
27
28
29

30 Cypher = [SP], Statement, [[SP], ';'], [SP] ;
31

32 100 Statement = Query ;
33
34

35 Query = RegularQuery ;
36
37

38
39 105 RegularQuery = SingleQuery, { [SP], Union } ;
40
41

42 SingleQuery = Clause, { [SP], Clause } ;
43
44

45 Clause = Match
46

47 110 | Unwind
48

49 | Merge
50

51 | Create
52

53 | Set
54

55 | Delete
56

57 ⁴www.opencypher.org
58

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```
115         | Remove
         | With
         | Return
         ;

120 SP = { whitespace }- ;
```

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

- **MATCH**: the graph pattern to match. The *MATCH* clause is the primary way of getting data from the database into the current set of bindings by providing an example. It allows users to specify the patterns that Cypher will search for in the database.
- **UNWIND**: expands a list into a sequence of rows.
- **MERGE**: ensures that a pattern exists in the graph. Either the pattern already exists, or it is created.
- **CREATE**: creates nodes and relationships.
- **SET**: is used to update labels on nodes and properties on nodes and relationships.
- **DELETE**: removes graphs elements (nodes, relationships and paths).
- **REMOVE**: is used to remove properties and labels from graph elements.
- **RETURN**: what to return. It defines which parts of the pattern the user is interested in. It can be nodes, relationships, or properties on them.
- **WITH**: allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

For instance, it is possible to retrieve which hotels have been visited by customers between the ages of 20 and 30 as shown in listing 1.

1
2
3
4
5
6
7
8
9 Listing 1: Retrieve which hotels have been visited by customers between the ages of 20 and
10 30 Query

```
11 MATCH (hotel:Hotel) <-[:VISITED]-(c: Customer)  
12 WHERE c.age <= 30 and c.age >=20  
13  
14 RETURN DISTINCT hotel  
148
```

15
16
17 Such queries allow the user to answer many questions in a very efficient way.
18 However, they do not allow soft queries, such as retrieving the hotels visited
19 by *young* customers where *young* is defined as a fuzzy set over the numerical
20 universe of all possible ages. For this reason, we claim that introducing ap-
21 proximate queries in NoSQL graph databases is important. The next section
22
23 150 introduces approximate queries. The next section
24 introduces approximate queries.
25
26

27 2.3. Approximate Queries

28
29 Many works have been proposed to deal with approximate data and queries.
30 It should first be noted that the consideration of imperfection may have several
31 semantics, should the data be uncertain, imprecise, vague, incomplete and/or
32 155 inconsistent [9].
33
34

35 This work focuses on approximate queries considered as imprecise queries
36 over precise (classical) data, relying on fuzzy set theory [10].
37
38

39 2.3.1. Fuzzy Queries

40
41 160 The works on fuzzy databases have dealt with the various topics: entity-
42 relationship models, functional dependencies, relational models, object data-
43 bases, queries, etc. We focus here on fuzzy queries over classical databases.
44
45

46 In [11, 12, 13] querying regular databases is considered by extending the SQL
47 language. The FSQL/SQLf and FQL languages have been proposed to extend
48 queries over relational databases in order to incorporate fuzzy descriptions of
49 165 the information being searched for.
50
51

52 In such systems, approximation is basically associated to fuzzy labels, fuzzy
53 comparators (e.g., *fuzzy greater than*) and aggregation over clauses. Thresholds
54 can be defined for the expected fulfillment of fuzzy clauses.
55
56
57
58

1
2
3
4
5
6
7
8
9
10 For instance, when considering a database containing crisp descriptions of
11 hotels, users can ask for *cheap* hotels that are *close to city center*, *cheap* and
12 *close_to_city_center* being fuzzy labels described by fuzzy sets and their mem-
13 bership functions respectively defined on the universe of prices and distance to
14 the city center.

15
16
17 For instance, SQLf allows to integrate fuzzy predicates within the *WHERE*
18 clause of an SQL query:
19
20
21

22 Listing 2: SQLf Query

```
23  
24 1 SELECT *  
25 2 FROM Table_Hotel  
26 188 WHERE Price is cheap;  
27
```

28 Some works have been implemented as fuzzy database engines with fuzzy
29 querying features [14, 15].
30

31 Many works have been proposed to investigate how such fuzzy predicates
32 can be defined by users and computed by the database engine, especially when
33 185 several clauses must be merged. For instance, for retrieving a hotel that is both
34 *cheap AND close to city center*, the membership degrees are aggregated with a
35 t-norm.
36
37
38

39 Such aggregation can consider preferences, for instance for queries where
40 190 price is preferred to distance to city center using weighted t-norms.
41

42 Thresholds can be added when searching for hotels where the *cheap* degree
43 is greater than 0.7 for instance.
44

45 More detailed definitions of the formalism are provided in the next section.
46

47 As we consider graph data, the works on fuzzy ontology querying are very
48 195 close and relevant [16, 17]. f-SPARQL provides a flexible extension for SPARQL
49 by allowing users to apply fuzzy filters and fuzzy operators. Figure 3 shows the
50 syntax of this extension.
51
52

53 However, no model has been proposed for NoSQL graph databases. We
54 thus propose to extend our first work [18] for defining approximate queries over
55 200 NoSQL graph databases, as described in the next section.
56
57
58

```

9      Query                ::= Prologue ( QueryType SelectQuery | ConstructQuery |
10      DescribeQuery | AskQuery )
11      QueryType            ::= '#FQ#' | #top-k FQ# with k
12      Constraint           ::= BrackettedExpression | BuiltInCall | FunctionCall
13      FlexibleExpression   ::= FuzzyTermExpression | FuzzyOperatorExpression
14      FuzzyTermExpression  ::= (Var ['=', '!=', '>', '>=', '<', '<='] FuzzyTerm)? [with threshold]
15      FuzzyOperatorExpression ::= Var FuzzyOperator NumericLiteral
16      FuzzyOperator        ::= (Modifier)*FuzzyOperator
17      FuzzyTerm            ::= FuzzyTerm and FuzzyTerm | FuzzyTerm or FuzzyTerm
18      | not FuzzyTerm | ModifiedFuzzyTerm
19      ModifiedFuzzyTerm    ::= (Modifier)* ModifiedFuzzyTerm | (Modifier)* SimpleFuzzyTerm

```

Figure 3: f-SPARQL modifications to the SPARQL standard grammar

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

3.1. Fuzzy Queries over Graph Databases

This section addresses fuzzy queries over regular NoSQL Neo4J graph databases.

205 We claim that fuzziness can be handled at the following three levels: over nodes, over properties, and over relationships.

The rest of this section presents each possibility along with its associated EBNF grammar.

EBNF Grammar Extract 2. *An expression is defined with sub-expressions*

210 *that can be chained with logical and arithmetical operators.*

Expression = Expression12 ;

Expression12 = Expression11, { SP, (O,R), SP, Expression11 } ;

215 Expression11 = Expression10, { SP, (X,O,R), SP, Expression10 } ;

Expression10 = Expression9, { SP, (A,N,D), SP, Expression9 } ;

Expression9 = { (N,O,T), [SP] }, Expression8 ;

220

Expression8 = Expression7, { [SP], PartialComparisonExpression } ;

```

1
2
3
4
5
6
7
8
9
10      Expression7 = Expression6, { ([SP], '+', [SP], Expression6) | ([SP], '-', [SP], Expression6) } ;
11
12
13
14      225 Expression6 = Expression5, { ([SP], '*', [SP], Expression5) |
15      ([SP], '/', [SP], Expression5) | ([SP], '%', [SP], Expression5) } ;
16
17      Expression5 = Expression4, { [SP], '^', [SP], Expression4 } ;
18
19
20      230 Expression4 = { ('+' | '-'), [SP] }, Expression3 ;
21
22
23      Expression3 = Expression2, { ([SP], '[', Expression, ']') |
24      ([SP], '[', [Expression], '..', [Expression], ']') | ((([SP], '=') |
25      (SP, (I,N)) | (SP, (S,T,A,R,T,S), SP, (W,I,T,H)) |
26      235 (SP, (E,N,D,S), SP, (W,I,T,H)) | (SP, (C,O,N,T,A,I,N,S))), [SP], Expression2) |
27      (SP, (I,S), SP, (N,U,L,L)) |
28      (SP, (I,S), SP, (N,O,T), SP, (N,U,L,L)) } ;
29
30
31
32
33      Expression2 = Atom, { [SP], (PropertyLookup | NodeLabels) } ;
34
35      240
36      Atom = Literal
37      | Parameter
38      | ((C,O,U,N,T), [SP], '(', [SP], '*', [SP], ')')
39      | ListComprehension
40      245 | PatternComprehension
41      | ((F,I,L,T,E,R), [SP], '(', [SP], FilterExpression, [SP], ')')
42      | ((E,X,T,R,A,C,T), [SP], '(', [SP], FilterExpression, [SP], [[SP], '|', Expression], ')')
43      | ((A,L,L), [SP], '(', [SP], FilterExpression, [SP], ')')
44      | ((A,N,Y), [SP], '(', [SP], FilterExpression, [SP], ')')
45      250 | ((N,O,N,E), [SP], '(', [SP], FilterExpression, [SP], ')')
46      | ((S,I,N,G,L,E), [SP], '(', [SP], FilterExpression, [SP], ')')
47      | RelationshipsPattern
48      | ParenthesizedExpression
49      | FunctionInvocation
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```
255 | Variable  
    ;
```

3.1.1. Cypherf over Nodes

Dealing with fuzzy queries over nodes allows the user to retrieve similar
260 nodes.

To achieve that goal, we decided to reuse the grammar definition available in Cypher. Indeed, we decided to use fuzzy functions that take nodes as parameters. Cypher supports function invocations in place of Expression.

EBNF Grammar Extract 3. *FunctionInvocation has a function name followed by zero to any numbers of parameters that are separated by commas.*
265

```
FunctionInvocation = FunctionName, [SP], '(', [SP], [(D,I,S,T,I,N,C,T), [SP]],  
                    [Expression, [SP], { ',', [SP], Expression, [SP] }], ')';
```

For instance, it is possible to retrieve similar hotels:

Listing 3: Getting Similar Hotel Nodes

```
270 1 MATCH (h1:Hotel), (h2:Hotel)  
    2 WITH h1 AS hot1, h2 AS hot2, SimilarTo(hot1, hot2) AS sim  
    3 WHERE sim > 0.7  
    4 RETURN hot1, hot2, sim  
275
```

In this framework, the similarity between nodes is based on the definition of measures that merge the similarities of every property from the two nodes being compared. For example, a similarity measure can be built to compare and rank hotels based on the similarities of both price and size [19].

280 3.1.2. Cypherf over Properties

Dealing with fuzzy queries over properties leads to the consideration of fuzzy linguistic terms and/or fuzzy comparators.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

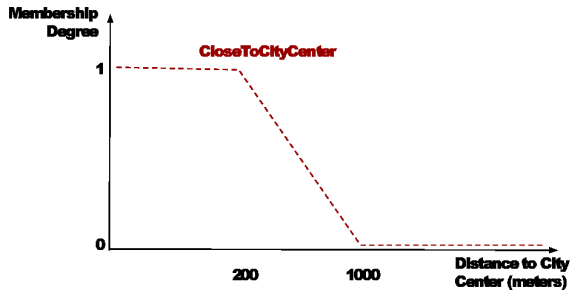


Figure 4: Proximity to City Center

Such fuzzy queries impact the *MATCH*, *WHERE*, *WITH* and *RETURN* clauses from Cypher.

285 In the *WHERE* clause, it is then possible to search for *cheap* hotels in some databases, or for hotels located *close to city center*. Note that the properties being considered can be defined at the node and relationship levels.

Listing 4: Cheap Hotels

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

Listing 5: Hotels Close to City Center

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

300 In the *MATCH* clause, integrating fuzzy criteria are also possible:

Listing 6: Matching Hotels Close to City Center

```

1 MATCH (h:Hotel) -[:LOCATED {CLOSE(distance)>0}]->(c:City)
2 RETURN h
3 ORDER BY CLOSE(h,c) DESC
305

```

To offer this possibility we have added the concept of fuzzy criterion to Cypher and we have extended its grammar. Indeed, Cypher is not able to use functions in “RelationshipPattern”. We thus introduce the following extension.

EBNF Grammar Extension 1. *“FuzzyCriteria” is defined as a function invocation that is an optional member of a “RelationshipDetail”. A Relationship detail is an optional part of a “RelationshipPattern” which is a part of a “PatternElementChain”, and so on to the “Pattern”.*

The extension of the RelationshipDetail possibilities by the addition of the FuzzyCriteria offers by transitivity the possibility to the Match clause to use FuzzyCriteria in a relationship construction.

```

Match = [(O,P,T,I,O,N,A,L), SP], (MATCH), [SP], Pattern, [[SP], Where] ;

Merge = (M,E,R,G,E), [SP], PatternPart, { SP, MergeAction } ;
320

Create = (C,R,E,A,T,E), [SP], Pattern ;

Pattern = PatternPart, { [SP], ',', [SP], PatternPart } ;

325 PatternPart = (Variable,[SP], '=', [SP], AnonymousPatternPart)
                | AnonymousPatternPart
                ;

AnonymousPatternPart = PatternElement ;
330

PatternElement = (NodePattern, { [SP], PatternElementChain })
                | ('(', PatternElement, ')')
                ;

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```

335 NodePattern = '(' , [SP] , [Variable , [SP]] , [NodeLabels , [SP]] , [Properties , [SP]] , ')' ;

PatternElementChain = RelationshipPattern , [SP] , NodePattern ;

RelationshipPattern = (LeftArrowHead , [SP] , Dash , [SP] , [RelationshipDetail] ,
340     [SP] , Dash , [SP] , RightArrowHead)
    | (LeftArrowHead , [SP] , Dash , [SP] , [RelationshipDetail] , [SP] , Dash)
    | (Dash , [SP] , [RelationshipDetail] , [SP] , Dash , [SP] , RightArrowHead)
    | (Dash , [SP] , [RelationshipDetail] , [SP] , Dash)
    ;

345 RelationshipDetail = '[' , [SP] , [Variable , [SP]] , [RelationshipTypes , [SP]] ,
    [RangeLiteral] , [FuzzyCriteria] , [Properties , [SP]] , ']' ;

FuzzyCriteria = '{' , [SP] , FunctionInvocation , [SP] , '}'

```

350 In the *RETURN* clause, no selection will be achieved, but fuzzy terms can be added in order to show the users the degree to which some values possess properties represented by fuzzy sets, as for instance:

Listing 7: Fuzziness in the Return Clause

```

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

```

3.1.3. Cypherf over Relationships

As for nodes, such queries may be based on properties. However it can also
360 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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

365 *MATCH* clause with consecutive links between objects. If a node a is linked to an object b at depth 2, the pattern is written as $(a) - [*2]- > (b)$. If a link between a and b without regarding the depth in-between is searched, then it is written $(a) - [*]- > (b)$. The mechanism also applies for searching objects linked through a range of nodes (e.g. between 3 and 5): $(a) - [*3..5]- > (b)$.

370 In a similar way to [20], we propose thus 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.

EBNF Grammar Extension 2. *FuzzyLinker* is defined as a *FunctionInvocation* that can be used in *RelationshipDetail*.

```
RelationshipDetail = '[' , [SP] , [Variable , [SP]] ,  
                    [RelationshipTypes , [SP]] , [RangeLiteral] , [FuzzyLinker] ,  
                    [FuzzyCriteria] , [Properties , [SP]] , ']' ;  
  
380 FuzzyLinker = '**' , [SP] , FunctionInvocation ;
```

Listing 8: Fuzzy Patterns

```
1 MATCH (c1:customer) -[:KNOWS**almost(3)]->(c2:customer)  
2 RETURN c1,c2
```

385 Fuzzy linker is related to fuzzy tree and graph mining [21] where some patterns emerge from several graphs even if they do not occur exactly the same way everywhere regarding the structure.

Popular hotels may for instance be retrieved when they are chosen by many people. This is similar to the way popular people are detected if they are followed
390 by a large number of people on social networks.

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

Listing 9: Aggregation

```
1 MATCH (c:Customer) -[:VISIT]->(h:Hotel)
2 WITH h AS hotel, count(c) AS nbCustomer
3 WHERE nbCustomer > 1
4 RETURN hotel
```

Such crisp queries can be extended to consider fuzziness:

Listing 10: Aggregation

```
1 MATCH (c:Customer) -[:VISIT]->(h:Hotel)
2 WITH h AS hotel, count(c) AS nbCustomer
3 WHERE POPULAR(nbCustomer) > 1
4 RETURN hotel
```

The question then raised is now to implement them in the existing Neo4J engine.

3.2. Cypherf: Implementation Challenges

3.2.1. Architecture

There are several ways to implement fuzzy Cypher queries:

1. Creating an overlay language on top of the Cypher language that will produce well formatted queries expressed as Cypher statements to do fuzzy computations;
2. Extending the Cypher queries and using the existing low level API behind;
3. Extending the low level API with optimized functions only available to advanced users with development skills;
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 and a synthesis is then provided.

3.2.2. Creating an Overlay Language

Concept

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

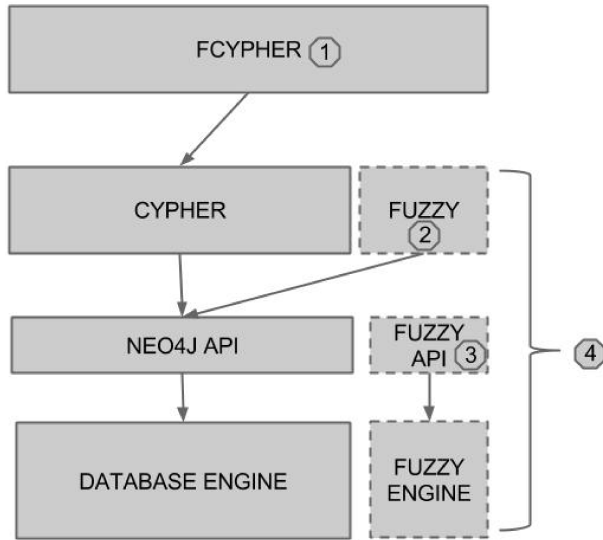


Figure 5: Implementation Ways

The concept is to create a new language dedicated to our framework. Such dedicated languages are also known as Domain Specific Languages (DSL). In this work, we consider a high-level fuzzy DSL that will be used to generate
425 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
430 as a low level language to achieve fuzzy queries.

Discussion

This solution is cheap and non-intrusive but has several serious drawbacks:

- Missing features: fuzzy queries (e.g., retrieving the data matching some fuzzy criterion) cannot be easily expressed by the current Cypher language (e.g., Listing 4) without creating ad hoc queries;
435
- Performance issues: Cypher is neither designed for fuzzy queries nor for being used as an algorithmic language. All the fuzzy queries will produce

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Cypher query codes that are not optimized for computations embedding fuzzy sets;

- 440 • Lack of user-friendliness: some queries cannot be executed directly against the Neo4J environment, it needs a three-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; (iii) reconstruct a “fuzzy answer” from a regular Cypher query result

445 *3.2.3. Extending the Cypher Queries*

Concept

We propose to extend the Cypher language in order to add new features. Cypher offers various types of functions: scalar functions, collection functions, predicate functions, mathematical functions etc. To enhance this language with 450 fuzzy features, we propose to add a new type of function: fuzzy functions. Fuzzy functions are used in the same way as other functions of Cypher (or SQL).

Cypher is an external DSL, which requires to be parsed. The query correctness must be checked and then it has to be executed. In Cypher the execution consists of retrieving the results from the query using the corresponding graph 455 traversal.

The Neo4J’s grammar provides the guidelines of how the language is supposed to be structured and what is and is not valid. In order to implement this solution, the Cypher grammar must be extended regarding the current grammar parser. Cypher uses the Scala Parser Combinator library (that will be 460 covered in more details in Section 4) as parsing engine. 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 requires a deep understanding of the Neo4J engine and more 465 knowledge of Java/Scala programming language (used to write the Neo4J engine and API) than the previous solution. The main advantage of this solution is to

offer an easy and user-friendly way to use the fuzzy feature. The disadvantages of this solution are:

- Performance issue. This solution should have better performance than the previous one. However, it is still 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 merge this contribution to the Neo4J project, it will be required to upgrade every new version of Neo4J with these enhancements. Every new version of Neo4j can make some breaking changes such as API changes. If so, part of the fuzzy extensions will need to be rewritten.

3.2.4. Extending Low Level API

Concept

This scenario consists of enhancing the core database engine with a framework to efficiently handle the fuzzy queries and to extend the programming API that can be used by developers.

Discussion

This solution offers a high performance improvement but needs advanced Neo4J skills, possibly high maintenance costs, a poor user-friendliness experience (only developers can use it) and a costly development process.

3.2.5. Extending Cypher Over an Enhanced Low Level API

Concept

The last possibility is to combine the solutions from Sections 3.2.3 and 3.2.4: adding the features handling the fuzzy queries to the database engine extending the API while also extending the Cypher language.

Discussion

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Scenario	Price	Features	Optimized	User Friendliness
Overlay Language	++	--	--	-
Extending Cypher Queries	+	-	+	++
Extending Low Level API	+	++	--	-
Cypher over an Enhanced Low Level API	--	++	+++	++

Table 1: Discussion Sum-Up

495 Every scenario has pros and cons. Table 1 sums up the above discussion.

The best, but most costly, solution is still the last one: extending Cypher query language and build a low level API framework to enhance the Neo4J database engine with support of fuzzy queries as presented below.

3.3. First Implementation: Naive Implementation

500

Prototype

In [18], a prototype based on the extension of Cypher over an enhanced API has been introduced allowing fuzzy queries to be run as shown in Figure 6.

505 This first implementation is called "naive" because it deliberately rejects sophisticated architecture and is a standard "proof of concept" implementation of what has been specified in previous section.

Discussion

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

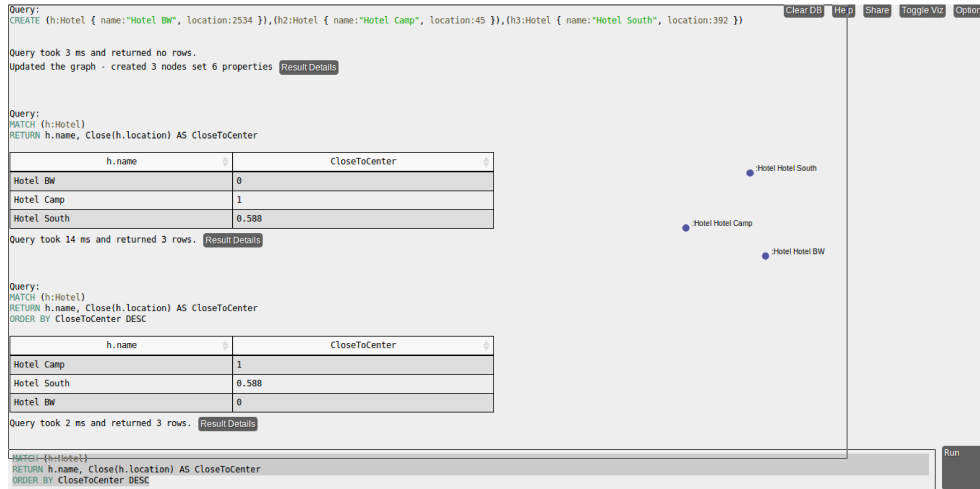


Figure 6: Prototype Developed

This implementation looks fairly user-friendly: a set of fuzzy functions to be used in Cypher queries are available to users. The main drawback of this implementation is that for each specific need a new function has to be developed.

For instance, as defined in section 3.1.1, to calculate the similarity between two nodes a function with two arguments (node1 and node2) has to be written. The calculation of the similarity between two hotels or two cars is based on different properties. This means that two different functions (SimilarityCars and SimilarityHotels) are needed. Each function makes assumption on some node properties (e.g. top speed, max power for a car).

The same problem occurs on fuzzy queries over properties: the μ_{cheap} fuzzy membership function is considered to calculate to which extent a property matches the "cheap" fuzzy term. The cheap function cannot be defined with the same μ_{cheap} membership function for a car or a hotel room: $\mu_{cheap_{car}}$ and $\mu_{cheap_{hotel}}$ have to be implemented in Scala, tested, compiled, and deployed in the Neo4J engine.

This is also a huge drawback when a user wants to define or refine a membership function: it is always a *code, build, run* cycle which requires the help of an developer or the possession of programming skills.

1
2
3
4
5
6
7
8
9 To conclude, this implementation is acceptable only for people who have few
10 fuzzy cases to handle or have development skills to implement fuzzy functions
11 by themselves.

12
13 This lack of genericity has highlighted the need for a new solution, described
14
15 530 in the next section.

16 17 18 *3.4. Cypherf: Extending Cypher to Fuzzy Queries*

19 20 *3.4.1. Overview*

21 This new extension has been proposed and implemented to reduce the draw-
22 backs listed in Section 3.3. The goal of this proposal is to provide an improved
23
24 535 implementation on two axes: it must reduce the need for specific development
25 and increase the capacity for the user to create his or her own membership
26 function.
27
28

29 Experimentation and benchmarking are available in Section 5.
30
31

32 33 *3.4.2. Cypher: an External Domain Specific Language*

34 540 As said before, Cypher is an external Domain Specific Language for express-
35 ing queries on NoSQL graph databases. A Domain Specific Language (DSL) is
36 a language tailored to a specific application domain [22].
37

38 DSLs are everywhere. Among the most popular are *SQL* for relational
39 databases, *Make* for building softwares and *CSS* for styling description.
40

41 545 The main advantages of DSLs are that they are expressive, concise and de-
42 signed at a high level of abstraction. Moreover, during the development cycle,
43 working on DSL is more scalable and tends to produce a higher payoff. The
44 main disadvantages are that the language design is hard and that DSLs lead
45 to performance issues. There are two main categories of Domain Specific Lan-
46 guages:
47
48
49 550

- 51 • The first one is known as "internal" DSL. It uses the infrastructure of
52 an existing programming language to build domain-specific semantics. It
53 is implemented as a particular form of API in a host general purpose
54
55
56
57
58

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

language, often referred to as a “fluent” interface. The term *fluent* is used
555 for an API that is primarily designed to be readable and to flow.

- The second category is named "external" DSL. This type of DSL is language agnostic, meaning that it is independent from a language syntax. It can thus be implemented in any language. One of the strengths of this category of DSL is its expressiveness, as it is not limited to a general/host programming language. The main drawback is that such languages have
560 to be parsed to be manipulated. This requires the writing of a language parser or the use of a tool that uses a grammar to generate a parser for a specific language. For instance, AntLR [23] (ANother Tool for Language Recognition) can be considered for this task.

565 *3.4.3. Cypherf: Adding a Fuzzy DSL to Cypher*

The naive implementation extends the Cypher language with new functions: fuzzy functions.

This approach does not provide an effective way to express membership functions. To provide a more generic implementation, it is necessary to add
570 another Domain Specific Language that allows the end-user to express fuzzy expressions.

The choice has been made to enhance Cypher with fewer functions but ones that allow the user to express fuzzy concerns with a "Fuzzy DSL". This new extension of Cypher is called Cypherf, standing for Cypherfuzzy.

575 *3.4.4. Features*

Cypherf allows the end-user to use fuzzy features such as defining membership function with the use of a fuzzy DSL, using t-norms and t-conorms functions as defined in [24] and using fuzzy linguistic variables.

The available functions are listed below (a formal grammar definition is
580 available in Section 4.2.3):

- Fuzzy(μ_f , value): returns the degree of membership of the μ_f function

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- μ_f describes the membership function with the Fuzzy DSL
- value is expressed as a Double

- FuzzyLT(fuzzyVariable, value): returns a collection that contains for every fuzzy term of the fuzzy linguistic variable two properties: the name of the term and the degree of compatibility of a "value" with the term. For instance, for a value X and a fuzzy linguistic variable $Age = (Age, [0, 130], \{young, middleaged, old\}, \{\mu_{young}, \mu_{middle}, \mu_{old}\})$ the result will be:

```
590 1 {  
2     {name:" young ", degree :  $\mu_{young}(x)$  },  
3     {name:" middle ", degree :  $\mu_{middle}(x)$  },  
4     {name:" old ", degree :  $\mu_{old}(x)$  }  
595 5 }
```

- fuzzyVariable: is expressed as a String that defines the fuzzy variable. This definition is composed of the name of the fuzzy variable and a set of fuzzy-terms. Every fuzzy term is defined by its name and its membership function

- 600 - value is expressed as a Double
- TNorms(tnormName, expression1, expression2): applies a TNorm of name "tnormName" on expression1 and expression2
- TCoNorms(type, expression1, expression2): same as TNorms but this time for t-conorms.

605 *3.4.5. Fuzzy DSL*

The fuzzy DSL is used to define membership functions and fuzzy linguistic variables.

It is processed by an underlying fuzzy framework called "Fuzzy4S" that implements the fuzzy logic. Fuzzy4S has been developed explicitly as part of this new implementation but can be used separately in other projects. The

1
2
3
4
5
6
7
8
9 grammar of the fuzzy DSL is explained in the section 4.2.3 after the presentation
10 of the Fuzzy4S framework and the introduction of some underlying concepts.
11

12
13 In this section, some contribution about defining and using fuzzy queries in a
14 graph database has been proposed. Several implementation scenarios have been
15 discussed and two implementations confronted. Implementations are based on
615 an extension of the declarative graph database query language Cypher and the
17 development of a low level engine that offers fuzzy features. The last implemen-
18 tation is more generic and user-friendly because it does not need programming
19 skills to be used.
20
21
22
23

620 The next section presents the underlying fuzzy framework that has been
25 developed for Cypherf.
26
27
28

29 **4. Fuzzy4S: A Fuzzy Logic Framework for Scala**

30

31 In this contribution, approximate queries are considered in the way they are
32 defined using fuzzy set theory.
33
34

625 *4.1. Overview*

35
36

37 In this paper, we introduce Fuzzy4S which is a fuzzy logic framework written
38 in Scala. Proposed as a library, it includes membership functions, t-norms and
39 t-conorms [10]. Upon this library, an open Domain Specific Language (DSL)
40 has been built to define approximate queries at an abstract level. It relies on
41 the IEC 61131 standard (IEC61131-7) that had been written for fuzzy control
630 programming and on jFuzzyLogic (a java fuzzy logic library) [25, 26].
45
46

47 The need for such contributions in the fuzzy set framework has been high-
48 lighted in [27].
49

50 The rest of this section introduces our Domain Specific Language and the
51 underlying features.
635
52
53
54
55
56
57
58

4.2. Fuzzy4S Grammar

Fuzzy4S can be manipulated directly (by using a Domain Specific Language based on a grammar that will be presented later in this section) or programmatically (by referencing objects and classes of the framework written in Scala). As written in [28], *Scala is implemented as a translation to standard Java bytecode*, which implies that the Scala code can be used in Java.

In the case of the Fuzzy4S DSL, the choice has been made to rely on external DSL which are more expressive than internal DSL. We have thus written a parser based on the concept of parsing expression grammar (PEG) which can be viewed as an extension of EBNF [29] and that is introduced below.

4.2.1. Fuzzy4S: Parsing Expression Grammar (PEG)

A Parsing Expression Grammar (PEG) is a recognition-based grammar formalism. The recognition-based systems have been developed in the 1970s. For decades, the Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), has been used to express the syntax of programming languages and protocols [30].

As said in [31], *the power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs*.

A PEG provides an alternative for describing machine oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express non-deterministic choice between alternatives, PEGs instead use prioritized choice by trying the alternatives in their order and unconditionally consuming the first successful match.

Figure 7 shows the notation of common operators available in PEG and the equivalent notation in Scala.

PEG operators are used in Fuzzy4S to parse the external DSL with a combination of several parsers. This technique is called combinatory parsing.

Description	PEG notation	Scala notation
Literal string	' '	" "
Literal string	" "	" "
Character class	[]	"[]".r
Any character	.	".r
Grouping	(e)	(e)
Optional	e?	(e?) or opt(e)
Zero-or-more	e*	(e*) or rep(e)
One-or-more	e+	(e+) or repl(e)
And-predicate	&e	guard(e)
Not-predicate	!e	not(e)
Sequence	e ₁ e ₂	e ₁ ~ e ₂
Prioritized Choice	e ₁ / e ₂	e ₁ e ₂

Figure 7: PEG operators [31] [32]

665 *4.2.2. Combinatory Parsing*

In functional programming, a common approach to parse domain-specific languages is to model parsers as functions and to define higher-order functions (also called combinators) that implement grammar constructions such as sequencing, choice and repetition. As explained in [33], the basic idea dates back
 670 to 1970s [34] and has become popular since 1980s in a variety of functional programming languages [35, 36, 37].

4.2.3. *Fuzzy4S Parsing Expression Grammar*

The parsing expression grammar for the DSL used in Fuzzy4S is defined in Listing 11.

Listing 11: Fuzzy4S Parsing Expression Grammar

```

675 1
49 2
50 3
51 4
52 5
53 685 def number: Parser[Double] = """-?\d+(\.\d*)?"""
54 7
55 8
56 9
57 10
58 11
59 12
60 13
61 14
62 15
63 16
64 17
65 18

```

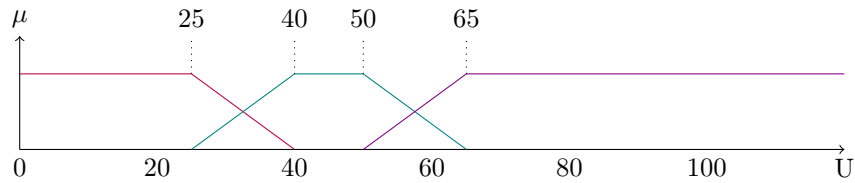


Figure 8: Result of listing 12

```

8   def termName = "[a-zA-Z]\w*"
689 def term = "TERM" ~> termName ~ "==" ~ memberfunction <~ ";"
10  def variableName = "[a-zA-Z]\w*"
11  def fuzzyVariable = "FUZZIFY" ~> variableName ~ rep1(term) <~ "<"
    END_FUZZIFY

```

Listing 12: Fuzzy variable example: fuzzy linguistic definition of Age

```

690 1 FUZZIFY age
    2     TERM young:= (25,1) (40,0);
    3     TERM middle:= trape 25 40 50 65;
    4     TERM old:= (50,0) (65,1);
695 END_FUZZIFY

```

As defined in Listing 11, a point can be expressed as a pair of two numbers, the first one defining the x coordinate and the second one the y coordinate. A membership function has an optional function name and some parameters (point or number). A Term defines a fuzzy set, it is composed of a name and a membership function. A fuzzy linguistic variable (named `fuzzyVariable` in Listing 11) is composed of a name, for instance *age*, and all the terms that compose this fuzzy linguistic variable (*young*, *old*, ...) such as defined in Listing 12 and represented by Figure 8.

Fuzzy linguistic variable definition can be stored in files and can be loaded in Fuzzy4S by using the file path. This choice has been made in order to provide a simple way for users to create their own library to define their specific linguistic terms.

One of our potential future project's feature will be to provide a web ap-

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

710 plication where users can publish their definition of fuzzy linguistic variable to
compose an open source repository. Users will be able to load definition from
this site by using HTTP URIs.

4.3. Epitomization

715 We claim that fuzzification is a challenge for graph databases. Our contribu-
tion is to propose an extension of the Cypher language, one of the most widely
adopted graph database query language, to handle fuzzy queries.

Our first implementation attempt required to create a new function for each
usecase and context (e.g., Cheap, Far, Cold, etc.). Each function implemen-
tation has to be invoked by the use of a *FunctionInvocation* (in Cypher's
720 EBNF grammar).

We think that this approach was lacking of genericity. We thus decided to
offer a better support of fuzzy features to the end users through the definition of
a domain specific language (DSL) called Fuzzy4S that provides the user with the
ability to define his own fuzzy linguistic vocabulary and terms with associated
725 functional membership functions and to chain them in their Cypherf queries.

Upon that, our choice of extending Cypher (instead of creating a separated
language) offers the user more flexibility by the possibility to use functions (e.g.
contains()) and operations (e.g. head()) of the underlying system.

The next section introduces some examples that show the power of our
730 contribution.

5. Examples and Experimentation

This section illustrates our contributions with examples and shows the effi-
ciency of our implementation with experimental load tests over benchmarks.

5.1. Using Fuzzy4S

735 This section is divided into two parts. The first is a showcase of some
classical examples of what can be done with Fuzzy4S. The second introduces a
web application that offers a web console to try out the Fuzzy4S framework.

Membership function	Scala code	DSL
PieceWise	PieceWiseMembershipFunction (List(ValuePoint(0,1), ValuePoint(2,1), ValuePoint(4,0)))	(0,1) (2,1) (4,0)
Triangular	TriangularMembershipFunction(0,2.5,5)	trian 0 2.5 5
Trapezoidal	TrapezoidalMembershipFunction(0,2,3,4)	trape 0 2 3 4
Sigmoidal	SigmoidalMembershipFunction(-4, 3)	sigm -4 3
Singleton	SingletonMembershipFunction(SingletonPoint(2))	2
...

Figure 9: Examples of membership functions using Scala4S

5.1.1. Examples

In Figure 9 some examples of using membership functions of Fuzzy4S are provided both in Scala code and with the external DSL.

5.1.2. Try Fuzzy4S: a Web Console

A web application is proposed to test the Fuzzy4S functionality. This application is a web console that offers the opportunity to define and evaluate some membership functions with the Fuzzy4S DSL as shown in Figure 10.

5.2. Using Cypherf

This section uses the example of searching for the best hotel for a trip to Paris. Two criteria will be explored: the price of the hotel for one night and the distance to the center of the city.

This section contains examples which can be seen as a step-by-step tutorial:

- example 1 and example 2 introduce basic Cypher query concepts required for the rest of this section;
- example 3 and example 4 introduce the use of membership functions with Cypherf;
- example 5, example 6 and example 7 illustrate how to use linguistic variables;

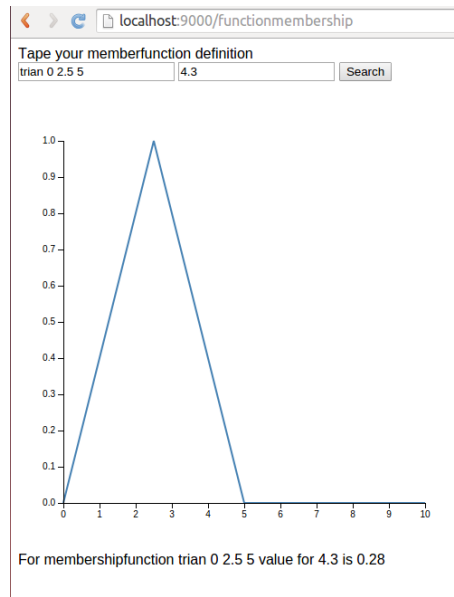


Figure 10: Fuzzy4S web console

- example 8 and example 9 use t-norms and t-conorms.

5.2.1. Data modelization and preliminary examples

Hotel and city center can be modeled as shown in Figure 11.

Using modelization from Figure 11, finding hotels close to the city center of the town requires calculating the distance from the hotel to the center of the city. The Euclidean distance between two points of the plane with Cartesian coordinates (x_1, y_1) and (x_2, y_2) is

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Example 1. Calculate the Euclidean distance in a graph database using Cypher

Listing 13: Calculate the euclidean distance between a hotel and the city center

```

765 match (c:Center), (h:Hotel)
2   with h.name as hotel, sqrt((c.posX - h.posX) * (c.posX - h.posX) ←
    + (c.posY - h.posY) * (c.posY - h.posY)) as distance

```

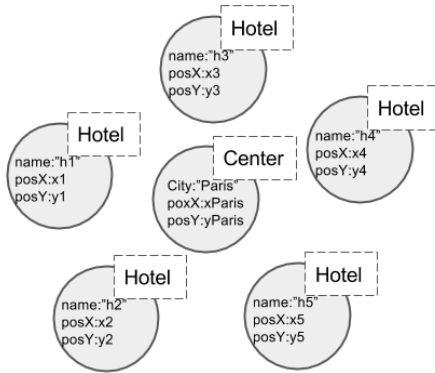


Figure 11: a possible data modelization

```

3  return hotel, distance
770 order by distance

```

Listing 13 shows how to calculate the Euclidean distance. The first line of Listing 13 specifies which node will be searched in the request, in this example all the nodes with label "Hotel" or label "Center". The WITH instruction manipulates the output before passing it to the rest of the query. It is a sort-of "pipe" command. In this example it takes hotels and center nodes and returns the hotel name to the rest of the query as a property called "hotel" and the distance expressed from the euclidean distance between the node and the center of the city.

In Example 1 the calculation of the euclidean distance must be done for each query execution. A better approach could be to store distance information into a relationship between the hotel and the city center.

The same technique can be considered for the result of the execution of a membership function like "close". It can also be stored in the same relationship.

Example 2. Store processed information in a relationship

Listing 14 shows how to store the result of the euclidean distance calculation in a relationship between hotel and city center.

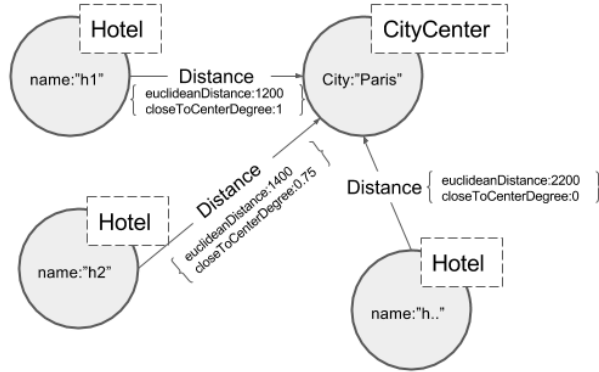


Figure 12: a data modelization with relationships

At the end of the query execution, the graph model looks like: $(h:Hotel)-[r:Distance\ distance:VALUE]->(c:Center)$ as represented in Figure 12.

Listing 14: Adding a relationship with euclidean distance

```

79b MATCH (c:Center), (h:Hotel)
2   WITH h, c, sqrt((c.posX - h.posX) * (c.posX - h.posX) + (c.posY -
36  - h.posY) * (c.posY - h.posY)) as distance
3   WITH h, c, distance, fuzzy("(1200,1) (2000,0)", distance) AS ←
38     closeToCenterDegree
79d CREATE (h)-[r:Distance {euclideanDistance:distance, closeToCenter:←
40     closeToCenterDegree}]->(c)
5   return h,r,c

```

In order to focus on the fuzzy expression of Cypherf language, the model
800 has been transformed. The hotels no longer have *posX* and *posY* properties,
only a distance property that expresses the distance between the hotel and the
center of Paris. The resulting data set is shown in Listing 15 and represented
in Figure 13

Listing 15: Dataset

```

80b CREATE (h1:Hotel { name:"h1", price:60, distance:3200})
2   CREATE (h2:Hotel { name:"h2", price:70, distance:2400})

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

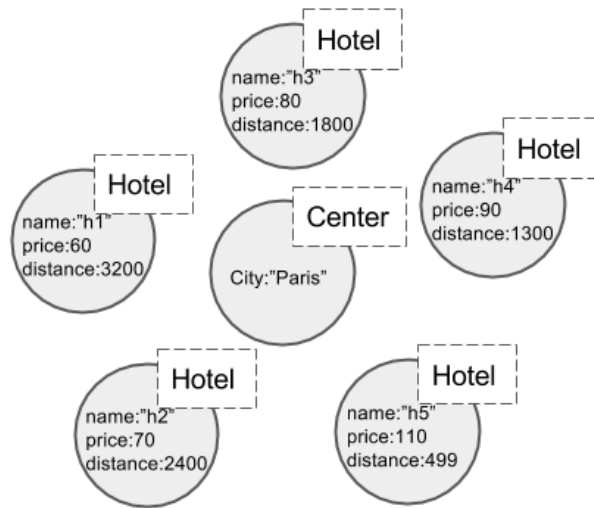


Figure 13: Figure of listing 15

```

3 CREATE (h3:Hotel { name: "h3", price:80, distance:1800})
4 CREATE (h4:Hotel { name: "h4", price:90, distance:1300})
5 CREATE (h5:Hotel { name: "h5", price:110, distance:499})
816 CREATE (c:Center { city: "Paris" })

```

5.2.2. Using a Membership Function

In this section, the goal is to retrieve the hotels close to the city center. To do so a piecewise membership function, illustrated in Figure 14, is defined as:

$$815 \quad f(x) = \begin{cases} 1 & 0 \leq x \leq 1200 \\ \frac{-x}{800} + \frac{5}{4} & 1200 < x < 2000 \\ 0 & x \geq 2000 \end{cases}$$

Example 3. Using a `closeToCenter` piecewise membership function in Cypherf

Listing 16 shows the use of the "fuzzy" Cypherf function that allows the user to use the Fuzzy DSL to describe a membership function. The result is the degree of membership of the hotel's distance to the defined membership function.

Constant flux Field weakening

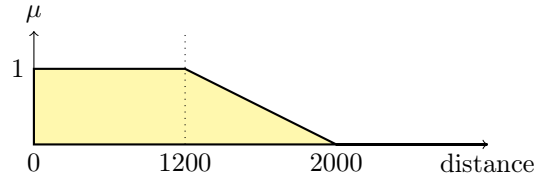


Figure 14: Distance membership function

```
$ match (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance return h.name, distance order by distance desc
```

	h.name	distance
Graph	h5	1
Rows	h4	0.875
	h3	0.25
	h1	0
	h2	0

Returned 5 rows in 64 ms.

Figure 15: Result of querying distance to retrieve hotel

Listing 16: Querying distance to retrieve hotel by closeToCenter degree

```
820
1 MATCH (h:Hotel)
2 WITH h, fuzzy("(1200,1) (2000,0)", h.distance) AS distance
3 RETURN h.name, distance
4
825 ORDER BY distance DESC
```

Figure 15 shows the result of listing 16.

Example 4. Using a threshold on the degree of membership to closeToCenter

Listing 17 retrieves the first ten results with a degree of 0.8 at least. Figure 16 shows the results.

Listing 17: Retrieving hotels close to city center with a threshold on degree of membership to closeToCenter, limit to the top three results

```
830
1 MATCH (h:Hotel)
2 WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance
3 WHERE distance > 0.8
```

```
$ MATCH (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance WHERE distance > 0.8 RETURN h.name, distance 0...
```

h.name	distance
h5	1
h4	0.875

Returned 2 rows in 45 ms.

Figure 16: Result of querying distance with threshold

```
4 RETURN h.name, distance
835 ORDER BY distance DESC
```

We have seen in this section how to use a membership function inside Cypherf query language. The next section shows how to use linguistic variables in Cypherf.

5.2.3. Using a Linguistic Variable

Let a linguistic variable price be defined with 3 terms: cheap, midPrice, expensive.

To allow users to retrieve the degree of membership associated to each term of a fuzzy linguistic variable, all membership functions must be applied. It is thus necessary to provide a "price.fl" (fl stands for FuzzyLogic) file that contains the variable definition as shown in Listing 18.

Listing 18: definition of price linguistic variable

```
1 FUZZIFY price
2     TERM cheap := (70,1) (85,0);
850     TERM midPrice := trape 75 85 90 105;
4     TERM expensive := (95,0) (110,1);
5 END_FUZZIFY
```

Example 5. Using Cypherf with a linguistic variable

If the linguistic variable definition is saved in the /tmp/price.fl location then Cypherf query can be used to build a query as in Listing 19.

```
$ match (h:Hotel) return h.name, fuzzyLT("/tmp/price.fl", h.price)
```

h.name	fuzzyLT("/tmp/price.fl", h.price)
h1	[<ul style="list-style-type: none"> name cheap degree 1 , <ul style="list-style-type: none"> name midPrice degree 0 , <ul style="list-style-type: none"> name expensive degree 0]
h2	[

Returned 5 rows in 77 ms.

Figure 17: Result of querying linguistic variable price

Listing 19: Using Cypherf with a linguistic variable

```
1 match (h:Hotel)
2 return h.name, fuzzyLT("/tmp/price.fl", h.price)
```

As shown in Figure 17, the query returns a collection of terms composed of a name and a degree. This collection is sorted by decreasing degree and (if two terms have the same degree) by term declaration order in the linguistic variable.

Example 6. Retrieving the most appropriate term of a linguistic variable

Listing 20 shows Cypher query to retrieve the **top** term of this linguistic variable for each hotel, meaning the most appropriate term of the price linguistic variable for each hotel.

Listing 20: Retrieving the most appropriate term of a linguistic variable

```
1 MATCH (h:Hotel)
2 WITH h, h.price as price
3 RETURN h.name, head(fuzzyLT("/tmp/price.fl", price))
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```
$ match (h:Hotel) with h, h.price as price return h.name, head(fuzzyLT("/tmp/price.fl", price))
```

Graph	h.name	head(fuzzyLT("/tmp/price.fl", price))				
Rows	h1	<table border="1"><tr><td>name</td><td>cheap</td></tr><tr><td>degree</td><td>1</td></tr></table>	name	cheap	degree	1
name	cheap					
degree	1					
	h2	<table border="1"><tr><td>name</td><td>cheap</td></tr><tr><td>degree</td><td>1</td></tr></table>	name	cheap	degree	1
name	cheap					
degree	1					
	h3	<table border="1"><tr><td>name</td><td>midPrice</td></tr><tr><td>degree</td><td>0.5</td></tr></table>	name	midPrice	degree	0.5
name	midPrice					
degree	0.5					
	h4	<table border="1"><tr><td>name</td><td>midPrice</td></tr><tr><td>degree</td><td>1</td></tr></table>	name	midPrice	degree	1
name	midPrice					
degree	1					
	h5					

Returned 5 rows in 77 ms.

Figure 18: Result of querying the top term of linguistic variable price

Line 3 of Listing 20 shows the fuzzyLT function returning the head of the collection, which is to say the element with the highest degree and declared first in the linguistic terms. Figure 18 shows that each returned element has two properties: a name and a degree.

Listing 21 illustrates the name of the hotels and the top price term name. Results are available in Figure 19.

Listing 21: Display the best price term name

```
88b MATCH (h:Hotel)
2 WITH h, h.price AS price
3 RETURN h.name, head(fuzzyLT("/tmp/price.fl", price)).name
```

Example 7. Storing fuzzy information in nodes

As seen in Listing 14, information processed from the fuzzy functions can be stored back in the graph. For instance, for every hotel, the top-price name can be stored as a property of the hotel node. Listing 22 illustrates how to store the


```
$ match (h:Hotel) with h, h.price as price return h.name, head(fuzzyLT("/tmp/price.fl", price)).name
```

Graph	h.name	head(fuzzyLT("/tmp/price.fl", price)).name
	h1	cheap
	h2	cheap
Rows	h3	midPrice
	h4	midPrice
	h5	expensive

Returned 5 rows in 49 ms.

Figure 19: Result of querying the name of the top term of linguistic variable price

```
$ match (h:Hotel) with h, h.price as price With h, head(fuzzyLT("/tmp/price.fl", price)) as priceTerm set h.priceCat = priceTerm.name
```

Graph	h.priceCat	collect(h.name)
	cheap	[h1, h2]
Rows	expensive	[h5]
	midPrice	[h3, h4]

Set 10 properties, returned 3 rows in 719 ms.

Figure 20: Result of storing category in Hotel node and grouping by category

most appropriate term in hotel node and shows that data can be grouped based on fuzzy information. Figure 20 shows the result of Listing 22.

Listing 22: Store back most appropriate term in a node, group by Category

```

890 1 MATCH (h:Hotel)
    2 WITH h, h.price AS price
    3 WITH h, head(fuzzyLT("/tmp/price.fl", price)) AS priceTerm
    4 SET h.priceCat = priceTerm.name, h.priceDegree = priceTerm.degree
895 RETURN h.priceCat, collect(h.name)

```

5.2.4. Combining Fuzzy Terms

In this example section, the goal is to find the hotels close to the city center and have low price. To do so, two linguistic variables are used: the price and the distance of the hotel to the city center. The price definition remains the same as in listing 18 and the fuzzy linguistic variable **distance** is defined as in Listing 23.

Listing 23: Distance definition

```

1 FUZZIFY distance
2
3 TERM close := (1200,1) (2000,0);
4 TERM medium := trape 1200 2000 2500 3000;
5 TERM far := (2500,0) (3000,1);
6
7 END_FUZZIFY

```

Example 8. *Using t-norms with two fuzzy linguistic variables*

Listing 24 shows the Cypherf query to achieve this goal and Figure 21 shows the results when using the min t-norm.

Listing 24: Using TNorm

```

1 MATCH (h:Hotel)
2 WITH h as hotel, h.price as hotelPrice, h.distance as ←
3 hotelDistance
4 WITH hotel, FuzzyLT("/tmp/distance.fl", hotelDistance) as ←
5 distanceTerms, FuzzyLT("/tmp/price.fl", hotelPrice) as ←
6 priceTerms
7 UNWIND distanceTerms as distance
8 UNWIND priceTerms as price
9 return hotel.name, distance.name, price.name, TNorm("Min", ←
10 distance.degree, price.degree) as minR order by TNorm("Min", ←
11 distance.degree, price.degree) desc

```

Example 9. *Final example: finding the best hotels*

Listing 25: Using TNorm and where clause

```

1 MATCH (h:Hotel)
2 WITH h AS hotel, h.price AS hotelPrice, h.distance AS ←
3 hotelDistance
4 WITH hotel, FuzzyLT("/tmp/distance.fl", hotelDistance) AS ←
5 distanceTerms, FuzzyLT("/tmp/price.fl", hotelPrice) AS ←
6 priceTerms
7 UNWIND distanceTerms AS distance
8 UNWIND priceTerms AS price
9 With hotel, distance, price
10 WHERE price.name = "cheap" and distance.name = "close"

```

\$ Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...

	hotel.name	distance.name	price.name	minR
Graph	h1	far	cheap	1
Graph	h2	medium	cheap	1
Rows	h5	close	expensive	1
Rows	h4	close	midPrice	0.875
Rows	h3	medium	midPrice	0.5
Rows	h3	medium	cheap	0.3333333333333337
Rows	h3	close	midPrice	0.25
Rows	h3	close	cheap	0.25
Rows	h4	medium	midPrice	0.125
Rows	h1	far	midPrice	0
Rows	h1	far	expensive	0
Rows	h1	close	cheap	0
Rows	h1	close	midPrice	0
Rows	h1	close	expensive	0
Rows	h1	medium	cheap	0
Rows	h1	medium	midPrice	0

Returned 45 rows in 81 ms.

Figure 21: Using t-norm

\$ Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl... 

	hotel.name	distance.name	price.name	minR
Graph	h3	close	cheap	0.25
Rows	h1	close	cheap	0
Rows	h2	close	cheap	0
Rows	h4	close	cheap	0
Rows	h5	close	cheap	0

Returned 5 rows in 78 ms.

Figure 22: Result of using TNorm with where clause

```

8 RETURN hotel.name, distance.name, price.name, TNorm("Min", ←
940 distance.degree, price.degree) AS minR ORDER BY TNorm("Min", ←
distance.degree, price.degree) DESC

```

Figure 22 shows that only one hotel, the cheapest one, could fit with our request. Nevertheless, sometimes the least expensive option is not the best match. We thus change the price criterion from "cheap" to "midPrice" as shown in Listing 26.

Listing 26: Requesting with price=midPrice and distance=Close

```

1 MATCH (h:Hotel)

```

\$ Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...

	hotel.name	distance.name	price.name	minR
Graph	h4	close	midPrice	0.875
Rows	h3	close	midPrice	0.25
	h1	close	midPrice	0
	h2	close	midPrice	0
	h5	close	midPrice	0

Returned 5 rows in 1064 ms.

Figure 23: Result of query with price=midPrice and distance=Close

```

2 WITH h AS hotel, h.price AS hotelPrice, h.distance AS ↵
   hotelDistance
950 WITH hotel, FuzzyLT("/tmp/distance.fl", hotelDistance) AS ↵
   distanceTerms, FuzzyLT("/tmp/price.fl", hotelPrice) AS ↵
   priceTerms
4 UNWIND distanceTerms AS distance
5 UNWIND priceTerms AS price
956 WITH hotel, distance, price
7 WHERE price.name = "midPrice" and distance.name = "close"
8 RETURN hotel.name, distance.name, price.name, TNorm("Min", ↵
   distance.degree, price.degree) AS minR ORDER BY TNorm("Min", ↵
960 distance.degree, price.degree) DESC

```

Result of listing 26 is shown in Figure 23.

5.3. Load Testing

5.3.1. Experimental Design

A test protocol has been defined to minimize the side effects (of both the dynamic compilation as well as the JVM memory management) on performance measurement based on works made in [32].

There are some mechanisms in the JVM which are transparent to the programmer, like automatic memory management, dynamic compilation, adaptive optimization and so on [38, 39]. These mechanisms are triggered implicitly and can have an enormous impact on measurements. For example:

- Just in Time (JIT) compilation: during the program the same code might exhibit very different performance characteristics. The HotSpot compiler could potentially compile any part of the code at any point during the

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

runtime. This can happen in the middle of running a benchmark, yielding
an inaccurate running time measurement;

- Classloading: JVM has global program information available (unlike a typical compiler). One method may be optimized based on the information in some unrelated method;
- Automatic memory management: the execution of running code can lead to the triggering of garbage collection cycle which changes the observed performance of the code being tested.

To generate better performance measurement, a test protocol has been defined and applied on every test:

- A separate JVM executor is created. Having a separate JVM ensures that only the bare essentials are needed to run the tests. It prevents the JIT compiler from applying some optimizations to the code.
- This executor has a default heap size set to 2GB
- To ensure that the JIT compiler optimized the code appropriately, the tests code monitors the running time during the warmup to dynamically detect if the running time has become stable
- A warmup is done before every test using statistical data analysis such as confidence interval as described in [40]

```
[info] Running test set for Fuzzy4S.membershipfunction, curve Test-0  
[info] Starting warmup.  
[info] 0. warmup run running time: 60 (covNoGC: NaN, covGC: NaN)  
...  
[info] 20. warmup run running time: 30 (covNoGC: 0.060, covGC: 0.354)  
[info] Steady-state detected.  
[info] Ending warmup.
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- 1000 • 36 repetitions of a measurement⁵ for every input size before moving on to benchmarking wider ranges
- a Min aggregator is used to retrieve the minimum running time of all the benchmarks run for each size

5.3.2. Fuzzy4S Measurement

1005 The goal of this section is to determine if Fuzzy4S performs as well as a state-of-the-art fuzzy engine written on the JVM. To do so, Fuzzy4S is compared to JFuzzyLogic [25] a fuzzy logic framework written in Java.

The performance scenario is described below.

- Use a range generator that generates an inclusive range of integer values. 1010 For instance for an x-axis of 10, it generates values 0,1,...,10.
- Use a triangular membership function.
- Calculate the running time of JFuzzyLogic and Fuzzy4S frameworks to parse the membership function and evaluate each value of x for every range

1015 The result of the benchmark in Figure 24 shows, for this request, better performance for Fuzzy4S than JFuzzyLogic, meaning that Fuzzy4S is an efficient fuzzy engine implementation.

5.3.3. Cypherf Measurement

1020 This section aims at evaluating the additional runtime costs (overhead) of Cypherf. This consists of discussing whether or not the extension of Cypher and the use of the Fuzzy4S engine handling fuzzy queries are acceptable. Listing 27 is a pure Cypher query that retrieves hotels' name whereas query of Listing 28 returns for every hotel its name and the most appropriate price term.

⁵Using 36 measurements increases the probability to have at least 30 measures after removing outliers due to external events (such as garbage collection), which allows us to tighten the confidence interval.

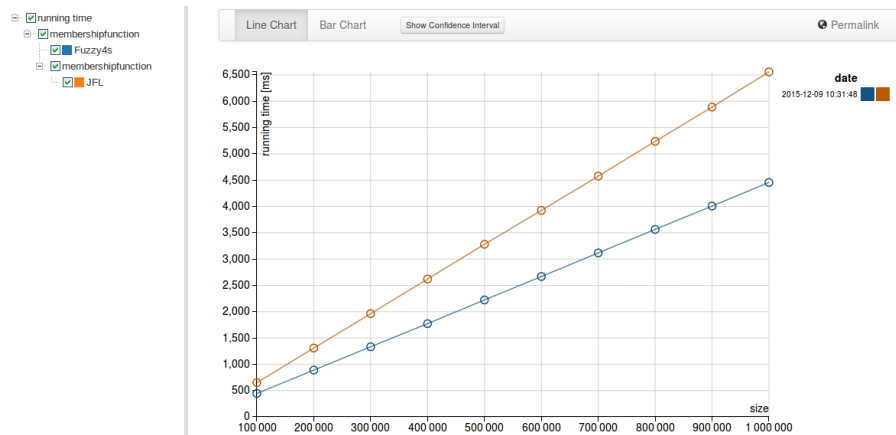


Figure 24: Benchmark comparison on 1 million nodes between Fuzzy4S and JFuzzyLogic frameworks

To better highlight performance measurement of fuzzy runtime overhead, a LIMIT 10 clause has been added. Indeed, the deserialization of query results is an important part of the overall time of each query.

Listing 27: Pure Cypher query in Neo4j

```
1 MATCH (h:Hotel) RETURN h.name LIMIT 10
```

Listing 28: Cypherf query on Neo4j-Fuzzy

```
1030 1 MATCH (h:Hotel)
1035 2 WITH h, h.price as price
3 RETURN h.name, head(fuzzyLT("/opt/research/data/price.fl", price))
4 LIMIT 10
```

Figure 25 shows that there is no major difference between Cypher and Cypherf execution. The additional cost of using fuzzy statements with Cypherf is thus compatible with the use of NoSQL graph databases.

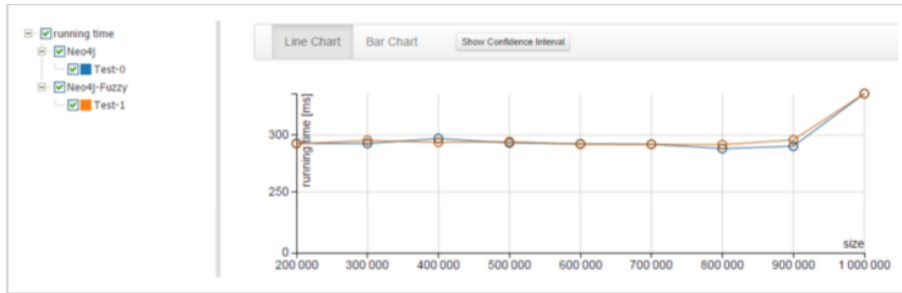


Figure 25: Benchmark comparison on 1 million nodes between a Cypher and a Cypherf query

6. Conclusion

1040 Graph databases are becoming more and more used by both scientists and
 1045 practitioners as the available engines are more and more powerful and scalable.
 They are used in many application domains, for the industry (e.g., logistics),
 ecommerce (e.g., ebay delivery service) or social networks (e.g., LinkedIn). Their
 ability to compute complex queries is a key feature. However, there is not yet
 any native possibility to compute fuzzy queries.

In this article, we thus introduce the necessary frameworks to support ap-
 proximate queries in NoSQL graph databases. Fuzzy4S provides a framework
 via a Domain Specific Language (DSL) to define approximate linguistic vari-
 ables. This DSL can then be used for querying NoSQL graph databases with
 1050 Cypherf which extends the existing declarative language to manage approxi-
 mate queries. Our propositions have been tested, we also have shown that the
 overhead is low assessing the power of the primitives that have been introduced.

This work opens up many research topics and possibilities. Regarding Big
 Data, many real-world applications and uses could be challenging for our ap-
 1055 proach. Fuzzy information may indeed be embedded within the data, especially
 in the (*key; value*) properties, as for instance Hotel *h1* could be located at
around 800 meters from the city center. It is also the case in social networks for
 all numerical properties on people and organizations (age, salary, size, revenue,
 etc.) on which various fuzzy variables may be defined depending on the context

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

1060 and/or application.

In particular, further work may focus on the definition of complex operations in such a context of schema-less databases, as for instance the definition of the *SimilarTo* function on nodes that have to be compared while they may have disjoint sets of properties.

1065 Moreover, Fuzzy4S and Cypherf may be used to retrieve fuzzy linguistic summaries from large NoSQL databases. Such approaches require to apply complex queries and will better highlight the power of graph databases (e.g., paths).

References

- 1070 [1] A. laszlo Barabasi, *Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life*, Basic Books, 2014.
- [2] A. Castelltort, A. Laurent, Rogue behavior detection in nosql graph databases, *J. Innovation in Digital Ecosystems* 3 (2) (2016) 70–82. doi: 10.1016/j.jides.2016.10.004.
1075 URL <http://dx.doi.org/10.1016/j.jides.2016.10.004>
- [3] R. Angles, C. Gutierrez, Survey of graph database models, *ACM Comput. Surv.* 40 (1) (2008) 1–39.
- [4] T. T. A. Board, *Technology radar* (2013).
- 1080 [5] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, M. Zenger, et al., *An overview of the scala programming language (second edition)*, Tech. rep. (2006).
- [6] I. Robinson, J. Webber, E. Eifrem, *Graph Databases*, O’Reilly, 2013.
- 1085 [7] R. E. Pattis, Teaching EBNF first in CS 1, in: R. Beck, D. Goelman (Eds.), *Proceedings of the 25th SIGCSE Technical Symposium on Computer Sci-*

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

ence Education, 1994, Phoenix, Arizona, USA, March 10-12, 1994, ACM, 1994, pp. 300–303. doi:10.1145/191029.191155.

URL <http://doi.acm.org/10.1145/191029.191155>

1090 [8] D. E. Knuth, backus normal form vs. backus naur form, Commun. ACM
7 (12) (1964) 735–736. doi:10.1145/355588.365140.

URL <http://doi.acm.org/10.1145/355588.365140>

[9] P. Smets, Imperfect information: Imprecision and uncertainty, in:
A. Motro, P. Smets (Eds.), Uncertainty Management in Information Sys-
1095 tems, Springer US, 1997, pp. 225–254.

[10] L. Zadeh, Fuzzy sets, Information and Control 8 (3) (1965) 338 – 353.

[11] P. Bosc, O. Pivert, SQLf: a relational database language for fuzzy querying,
Fuzzy Systems, IEEE Transactions on 3 (1) (1995) 1–17.

[12] Y. Takahashi, A fuzzy query language for relational databases, IEEE Trans-
1100 actions on Systems, Man, and Cybernetics 21 (6) (1991) 1576–1579.

[13] D. Dubois, H. Prade, Using fuzzy sets in flexible querying: Why and how?,
in: T. Andreasen, H. Christiansen, H. Larsen (Eds.), Flexible Query An-
swering Systems, Kluwer Academic Publishers, Boston, 1997, pp. 45–60.

[14] S. Zadrozny, J. Kacprzyk, Implementing fuzzy querying via the inter-
1105 net/www: Java applets, activex controls and cookies., in: T. Andreasen,
H. Christiansen, H. L. Larsen (Eds.), FQAS, Vol. 1495 of Lecture Notes in
Computer Science, Springer, 1998, pp. 382–392.

[15] J. Galindo, J. M. Medina, O. Pons, J. C. Cubero, A server for fuzzy sql
queries., in: T. Andreasen, H. Christiansen, H. L. Larsen (Eds.), FQAS,
1110 Vol. 1495 of Lecture Notes in Computer Science, Springer, 1998, pp. 164–
174.

[16] J. Z. Pan, G. B. Stamou, G. Stoilos, S. Taylor, E. Thomas, Scalable query-
ing services over fuzzy ontologies, in: J. Huai, R. Chen, H.-W. Hon, Y. Liu,

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

W.-Y. Ma, A. Tomkins, X. Z. 0001 (Eds.), WWW, ACM, 2008, pp. 575–
1115 584.

[17] J. Cheng, Z. M. Ma, L. Yan, f-sparql: A flexible extension of sparql, in:
P. G. Bringas, A. Hameurlain, G. Quirchmayr (Eds.), DEXA (1), Vol. 6261
of Lecture Notes in Computer Science, Springer, 2010, pp. 487–494.

[18] A. Castelltort, A. Laurent, Fuzzy queries over nosql graph databases: Per-
1120 spectives for extending the cypher language, in: International Conference
on Processing and Management of Uncertainty in Knowledge-Based Sys-
tems, Springer, 2014.

[19] M.-J. Lesot, M. Rifqi, H. Benhadda, Similarity measures for binary and
numerical data: a survey, International Journal of Knowledge Engineering
1125 and Soft Data Paradigms 1 (1) (2008) 63–84. doi:10.1504/ijkesdp.2009.
021985.
URL <https://hal.inria.fr/hal-01072737>

[20] E. Panzeri, G. Pasi, A flexible extension of xquery full-text, in: R. Basili,
F. Sebastiani, G. Semeraro (Eds.), Proceedings of the 4th Italian Infor-
1130 mation Retrieval Workshop, Pisa, Italy, January 16-17, 2013, Vol. 964 of
CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 29–32.
URL <http://ceur-ws.org/Vol-964/paper4.pdf>

[21] F. D. R. López, A. Laurent, P. Poncelet, M. Teisseire, FTMnodes: Fuzzy
tree mining based on partial inclusion, Fuzzy Sets and Systems 160 (15)
1135 (2009) 2224–2240.

[22] M. Fowler, Domain Specific Languages, 1st Edition, Addison-Wesley Pro-
fessional, 2010.

[23] T. Parr, The definitive ANTLR reference: building domain-specific lan-
guages, Pragmatic Bookshelf; 1 edition, 2007.

1140 [24] E. P. Klement, E. Pap, R. Mesiar, Triangular norms, Trends in logic,
Kluwer Academic Publ. cop., Dordrecht, Boston, London, 2000.

- 1
2
3
4
5
6
7
8
9 [25] P. Cingolani, J. Alcalá-Fdez, jFuzzyLogic: a robust and flexible Fuzzy-Logic
10 inference system language implementation, in: FUZZ-IEEE, 2012, pp. 1–8.
11
12 [26] P. Cingolani, J. Alcalá-Fdez, jFuzzyLogic: a java library to design fuzzy
13 logic controllers according to the standard for fuzzy control programming,
1145 International Journal of Computational Intelligence Systems 6 (2013) 61–
14 75.
15
16 [27] B. N. D. Stefano, On the need of a standard language for designing fuzzy
17 systems, in: G. Acampora, V. Loia, C. Lee, M. Wang (Eds.), On the
18 Power of Fuzzy Markup Language, Vol. 296 of Studies in Fuzziness and Soft
19 Computing, Springer, 2013, pp. 3–15. doi:10.1007/978-3-642-35488-5.
20 URL <http://dx.doi.org/10.1007/978-3-642-35488-5>
21
22 [28] M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima Inc,
23 2008.
24
25 [29] R. R. Redziejowski, From EBNF to PEG, in: Proceedings of the 21th
1155 International Workshop on Concurrency, Specification and Programming,
26 Berlin, Germany, September 26-28, 2012, 2012, pp. 324–335.
27
28 [30] N. Chomsky, On certain formal properties of grammars, Information and
29 Control 2 (2) (1959) 137 – 167.
30
31 [31] B. Ford, Parsing expression grammars: a recognition-based syntactic foun-
32 dation, in: ACM SIGPLAN Notices, Vol. 39, ACM, 2004, pp. 111–122.
33
34 [32] P. H. Nguyen, M. Odersky, Scala benchmarking suite-scala performance
35 regression pinpointing.
36
37 [33] G. Hutton, E. Meijer, Monadic parser combinators, School of Computer
38 Science and IT, 1996.
39
1160 [34] H. Burge William, Recursive programming techniques, Reading Mass:
40 Addison-Wesley, 1975.
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

[35] P. Wadler, How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages, in: *Functional Programming Languages and Computer Architecture*, Springer, 1985, pp. 113–128.

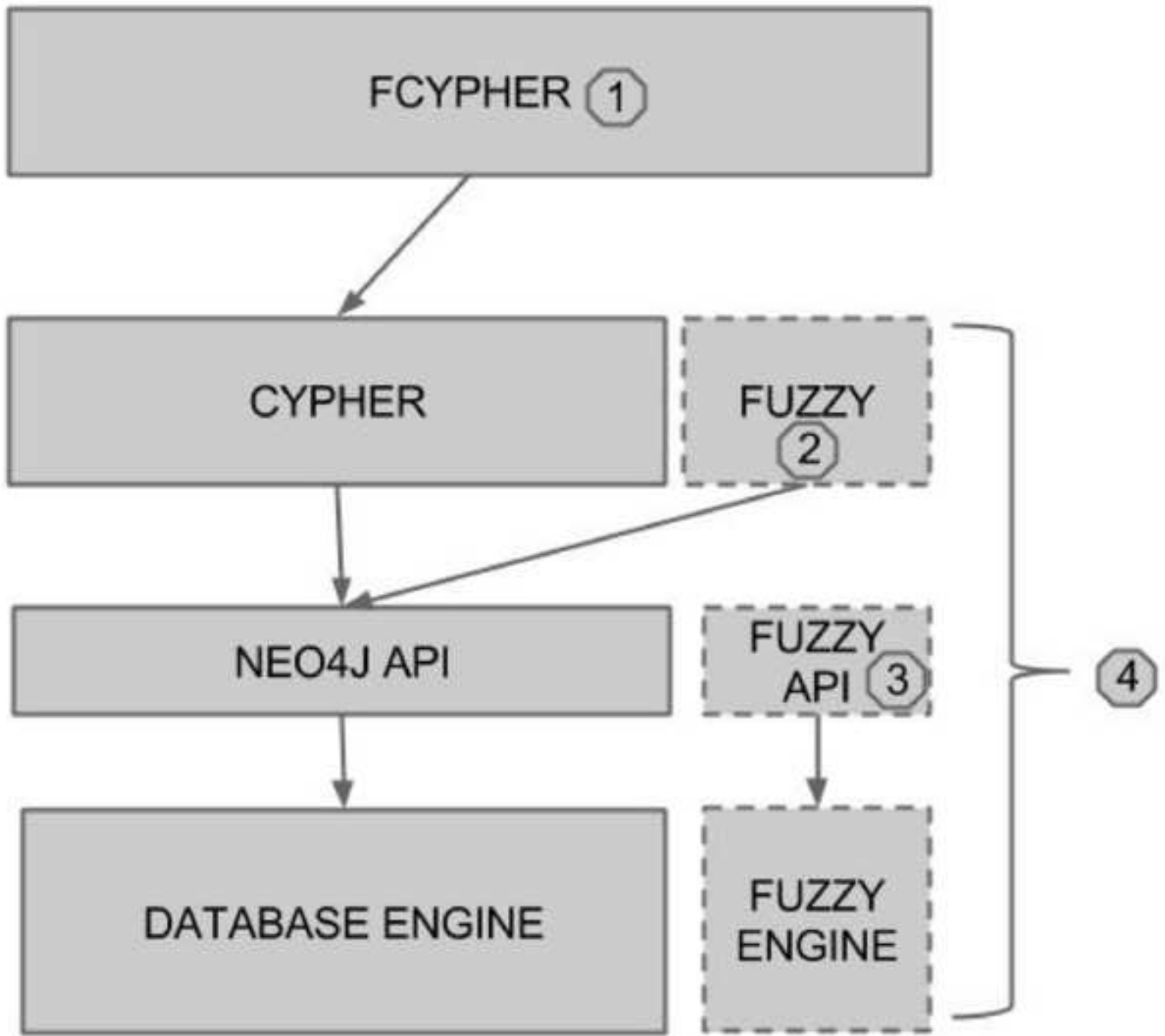
[36] G. Hutton, Higher-order functions for parsing, *J. Funct. Program.* 2 (3) (1992) 323–343.

[37] J. Fokker, Functional parsers, in: *Advanced functional programming*, Springer, 1995, pp. 1–23.

[38] B. Goetz, Java theory and practice: Dynamic compilation and performance measurement, *IBM Developer Works* (2004) 1–8.

[39] B. Goetz, Java theory and practice: Anatomy of a flawed microbenchmark, *IBM developer-Works*.

[40] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, *ACM SIGPLAN Notices* 42 (10) (2007) 57–76.



Query: `CREATE (h1:Hotel { name:"Hotel BW", location:2534 }),(h2:Hotel { name:"Hotel Camp", location:45 }),(h3:Hotel { name:"Hotel South", location:392 })` Clear DB Help Share Toggle Viz Options

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

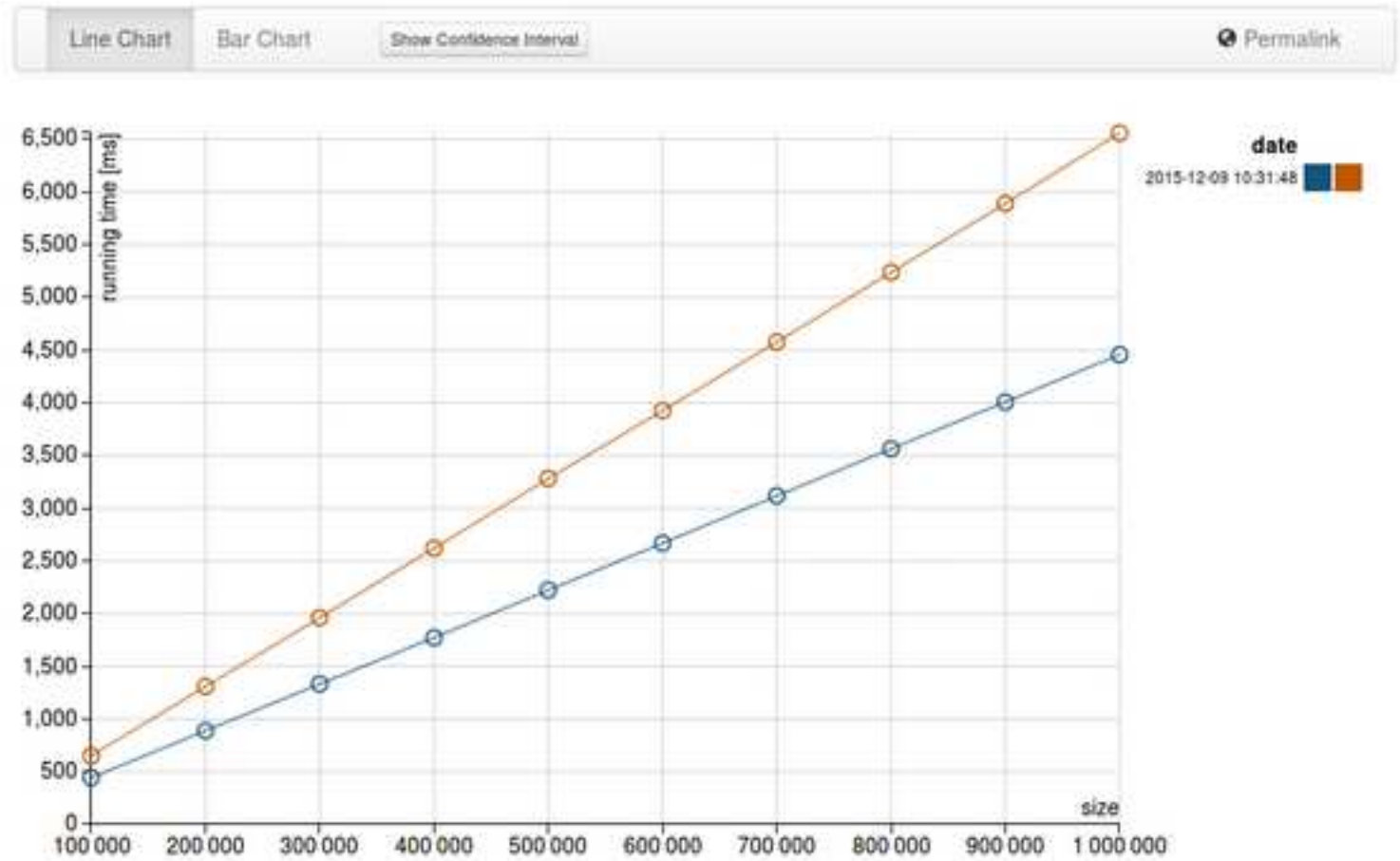
Query took 2 ms and returned 3 rows. Result Details

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



Run

- running time
- membershipfunction
 - Fuzzy4s
- membershipfunction
 - JFL



19 systems in ranking, November 2015

Rank			DBMS	Database Model	Score		
Nov 2015	Oct 2015	Nov 2014			Nov 2015	Oct 2015	Nov 2014
1.	1.	1.	Neo4j	Graph DBMS	34.04	+0.63	+9.39
2.	2.	2.	Titan	Graph DBMS	6.06	+0.54	+3.45
3.	3.	3.	OrientDB	Multi-model	5.50	+0.57	+3.48
4.	4.	7.	ArangoDB	Multi-model	1.61	+0.13	+1.30
5.	5.	6.	Giraph	Graph DBMS	0.92	-0.00	+0.45
6.	6.	5.	AllegroGraph	Multi-model	0.91	+0.01	+0.30
7.	7.	11.	Stardog	Multi-model	0.54	+0.02	+0.41
8.	9.	9.	Sqrrl	Multi-model	0.42	-0.01	+0.24
9.	8.	8.	InfiniteGraph	Graph DBMS	0.38	-0.05	+0.12
10.	10.	4.	Sparksee	Graph DBMS	0.29	-0.02	-0.59
11.	11.	15.	HyperGraphDB	Graph DBMS	0.25	-0.01	+0.22
12.	12.	12.	InfoGrid	Graph DBMS	0.21	+0.00	+0.09
13.	14.		VelocityGraph	Graph DBMS	0.19	+0.02	
14.	15.	16.	GlobalsDB	Multi-model	0.18	+0.02	+0.18
15.	13.	13.	FlockDB	Graph DBMS	0.16	-0.02	+0.04
16.	17.	10.	GraphDB	Multi-model	0.10	+0.06	-0.06
17.	16.	14.	GraphBase	Graph DBMS	0.05	-0.01	+0.02
18.	18.		Blazegraph	Multi-model	0.01	+0.00	
19.	19.	16.	Amisa Server	Multi-model	0.00	±0.00	±0.00

Description	PEG notation	Scala notation
Literal string	<code>/ /</code>	<code>" "</code>
Literal string	<code>" "</code>	<code>" "</code>
Character class	<code>[]</code>	<code>"[]".r</code>
Any character	<code>.</code>	<code>".".r</code>
Grouping	<code>(e)</code>	<code>(e)</code>
Optional	<code>e?</code>	<code>(e?)</code> or <code>opt(e)</code>
Zero-or-more	<code>e*</code>	<code>(e*)</code> or <code>rep(e)</code>
One-or-more	<code>e+</code>	<code>(e+)</code> or <code>rep1(e)</code>
And-predicate	<code>&e</code>	<code>guard(e)</code>
Not-predicate	<code>!e</code>	<code>not(e)</code>
Sequence	<code>e₁ e₂</code>	<code>e1 ~ e2</code>
Prioritized Choice	<code>e₁ / e₂</code>	<code>e1 e2</code>

```
$ Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...
```

	hotel.name	distance.name	price.name	minR
Graph	h1	far	cheap	1
Rows	h2	medium	cheap	1
	h5	close	expensive	1
	h4	close	midPrice	0.875
	h3	medium	midPrice	0.5
	h3	medium	cheap	0.33333333333333337
	h3	close	midPrice	0.25
	h3	close	cheap	0.25
	h4	medium	midPrice	0.125
	h1	far	midPrice	0
	h1	far	expensive	0
	h1	close	cheap	0
	h1	close	midPrice	0
	h1	close	expensive	0
	h1	medium	cheap	0
	h1	medium	midPrice	0

Returned 45 rows in 81 ms.

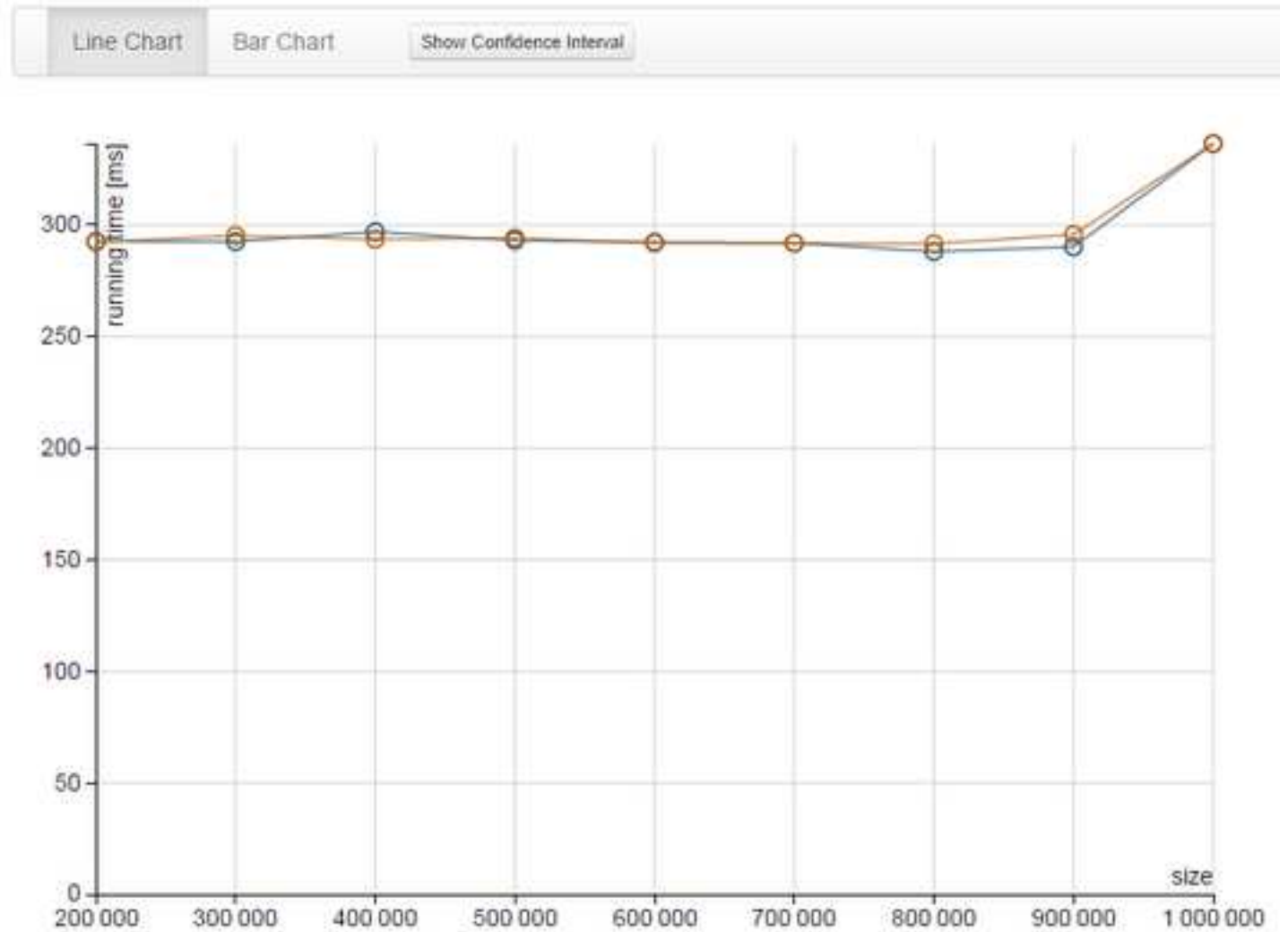
5 Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl... 

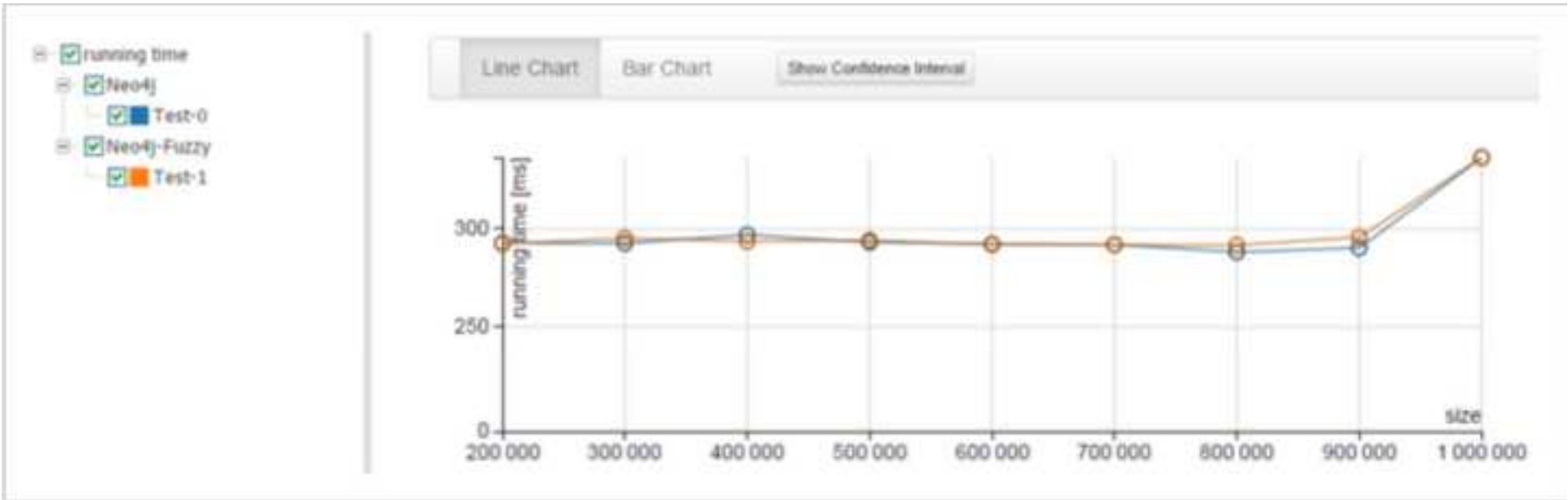
 Graph	hotel.name	distance.name	price.name	minR
	h3	close	cheap	0.25
 Rows	h1	close	cheap	0
	h2	close	cheap	0
	h4	close	cheap	0
	h5	close	cheap	0
Returned 5 rows in 78 ms.				

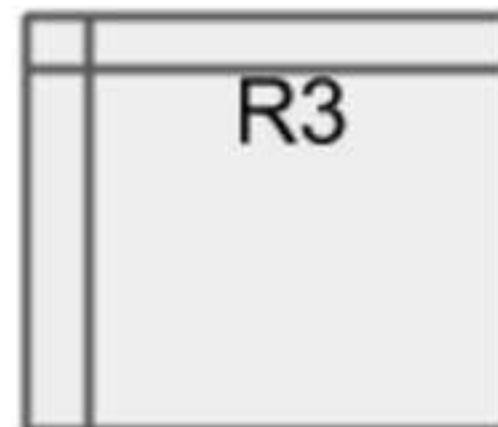
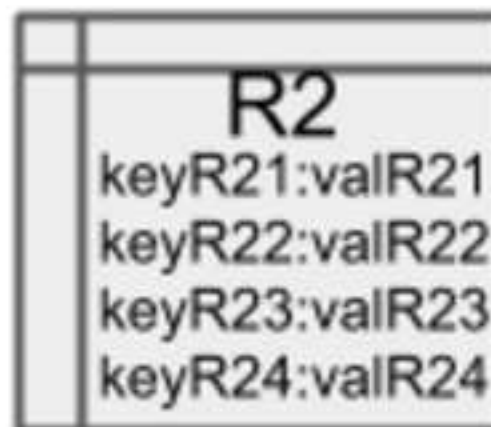
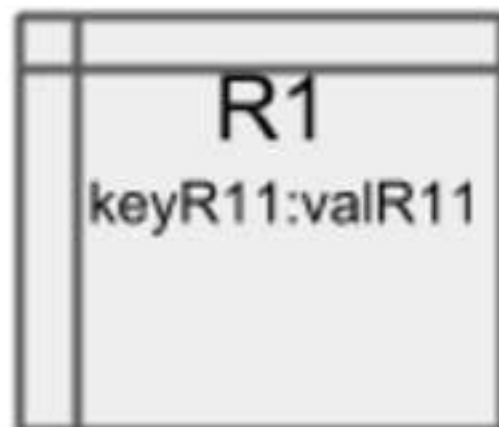
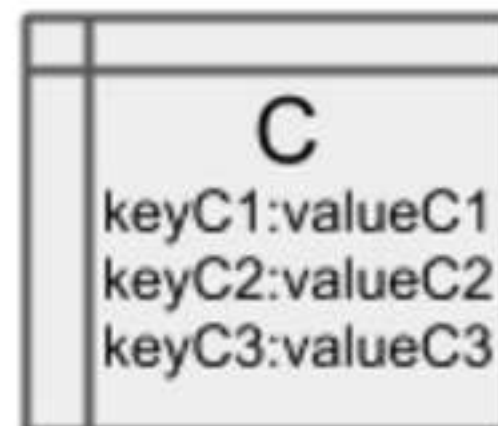
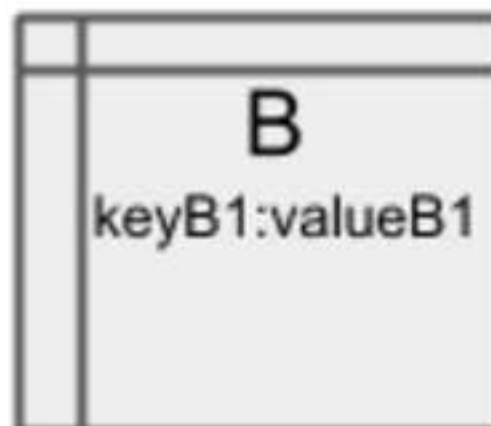
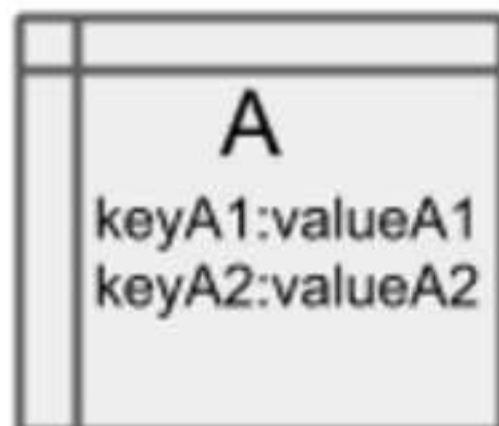
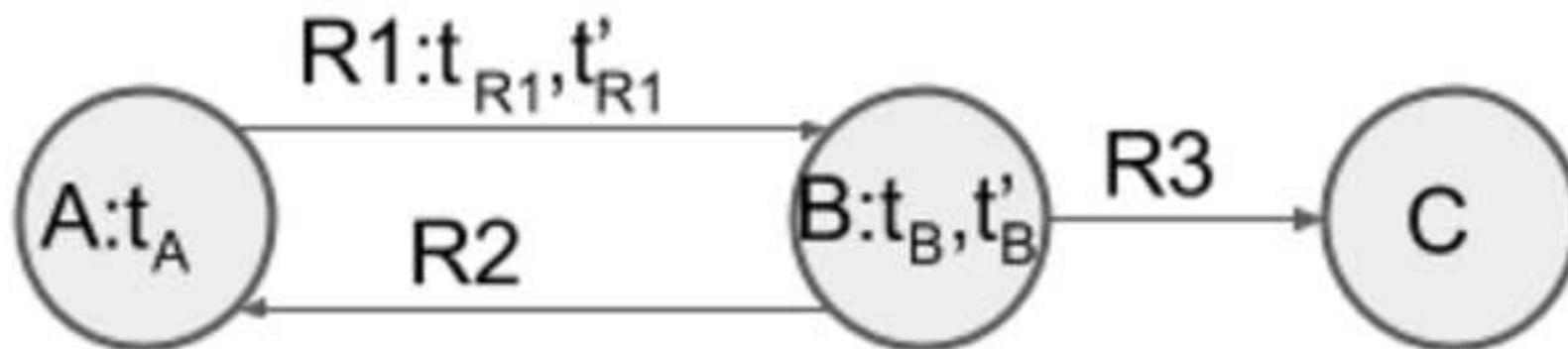
```
$ Match (hiHotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...
```

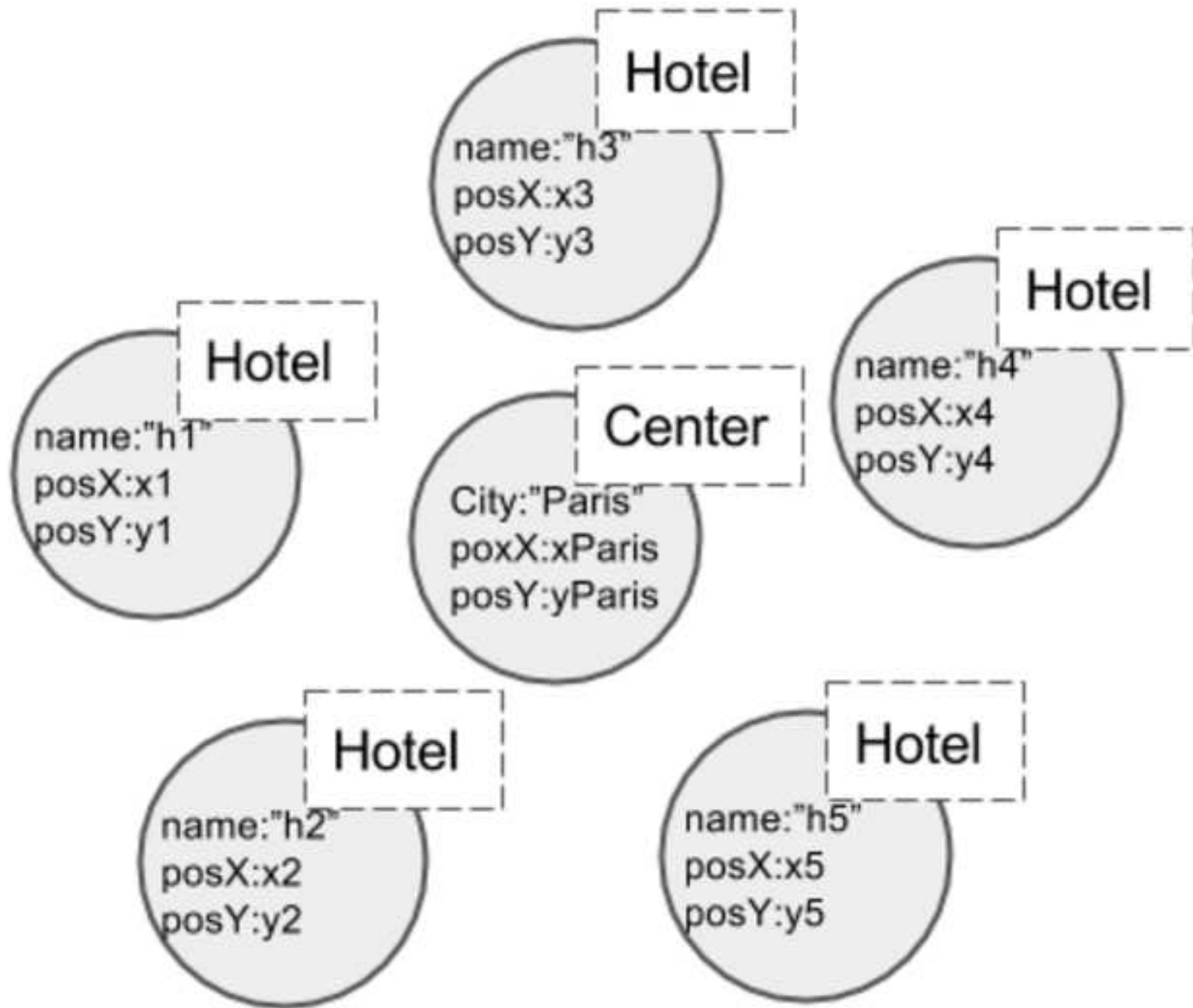
	hotel.name	distance.name	price.name	minR
 Graph	h4	close	midPrice	0.875
 Rows	h3	close	midPrice	0.25
	h1	close	midPrice	0
	h2	close	midPrice	0
	h5	close	midPrice	0
Returned 5 rows in 1064 ms.				

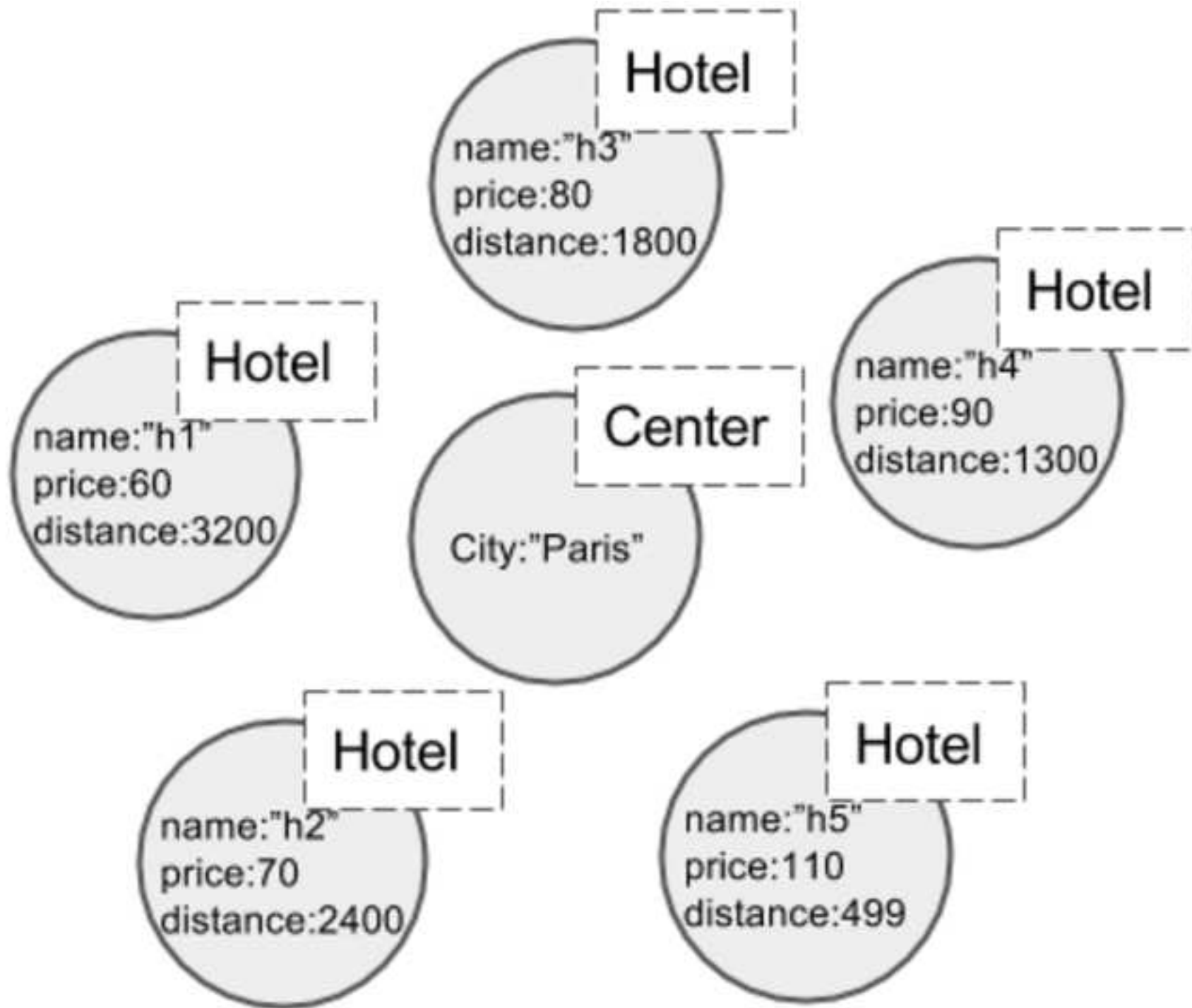
- running time
- Neo4j
- Test-0
- Neo4j-Fuzzy
- Test-1





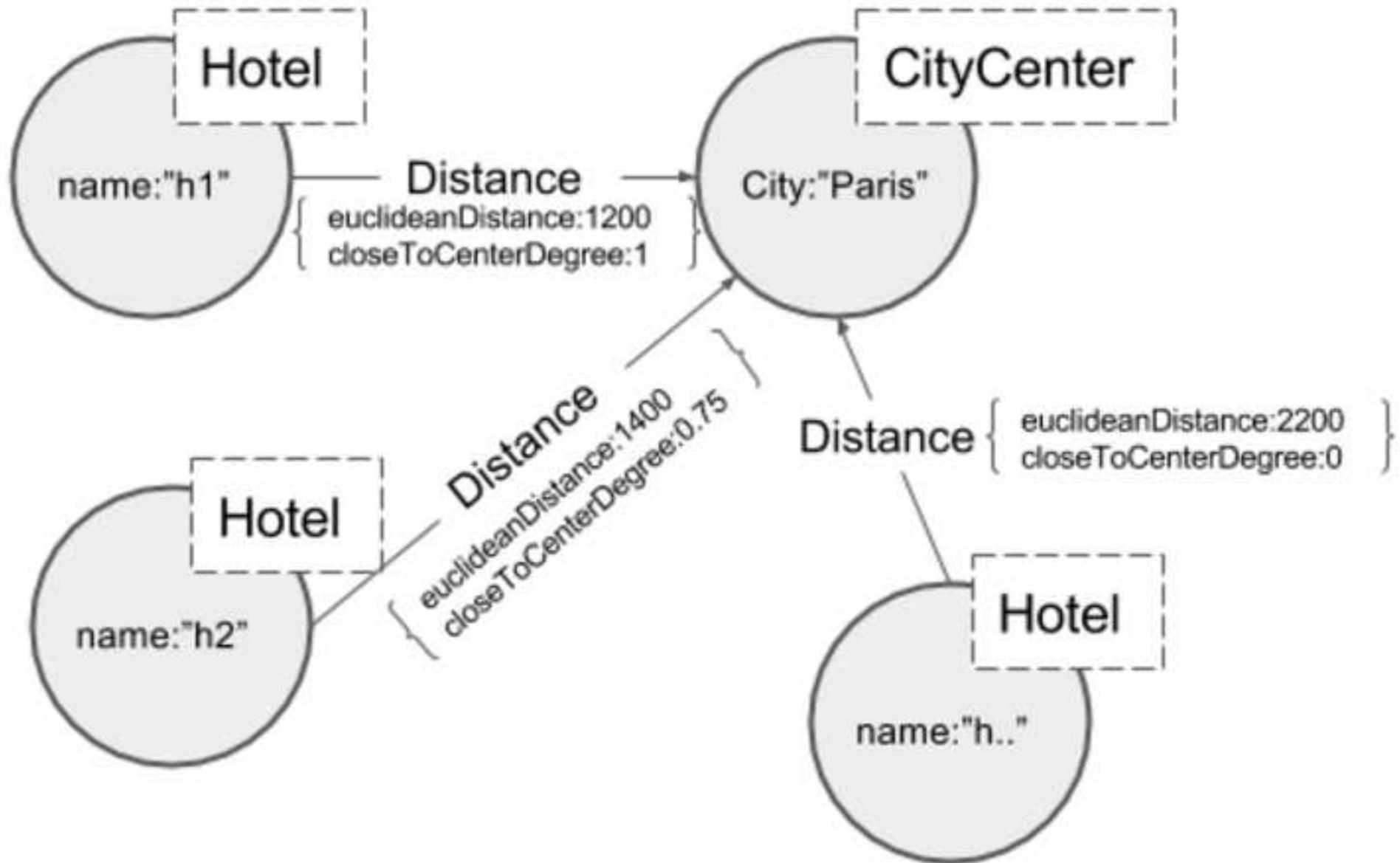






27 systems in ranking, July 2017



Rank			DBMS	Database Model	Score		
Jul 2017	Jun 2017	Jul 2016			Jul 2017	Jun 2017	Jul 2016
1.	1.	1.	Neo4j	Graph DBMS	38.52	+0.65	+4.83
2.	2.	4.	Microsoft Azure Cosmos DB	Multi-model	7.71	+1.33	+5.29
3.	3.	2.	OrientDB	Multi-model	5.57	-0.24	-0.26
4.	4.	3.	Titan	Graph DBMS	4.93	-0.33	-0.34
5.	5.	6.	ArangoDB	Multi-model	2.96	-0.12	+1.21
6.	6.	5.	Virtuoso	Multi-model	1.99	-0.01	-0.31
7.	7.	7.	Giraph	Graph DBMS	1.05	+0.00	+0.16
8.	8.	8.	AllegroGraph	Multi-model	0.61	+0.00	+0.14
9.	9.	9.	Stardog	Multi-model	0.55	+0.01	+0.09
10.	10.	12.	GraphDB	Multi-model	0.53	+0.01	+0.36
11.	11.	10.	Sqrrl	Multi-model	0.49	+0.02	+0.24
12.	16.		Graph Engine	Multi-model	0.36	+0.10	
13.	13.	11.	InfiniteGraph	Graph DBMS	0.29	-0.01	+0.10
14.	15.	18.	Blazegraph	Multi-model	0.29	-0.00	+0.21
15.	14.		Dgraph	Graph DBMS	0.28	-0.02	
16.	19.		JanusGraph	Graph DBMS	0.23	+0.11	
17.	21.	16.	Sparksee	Graph DBMS	0.16	+0.05	+0.07
18.	17.	14.	FlockDB	Graph DBMS	0.16	-0.03	+0.03
19.	18.	17.	HyperGraphDB	Graph DBMS	0.15	-0.03	+0.07
20.	20.	15.	InfoGrid	Graph DBMS	0.13	+0.01	+0.04



Query ::= Prologue (*QueryType* SelectQuery | ConstructQuery | DescribeQuery | AskQuery)
 QueryType ::= '#FQ#' | #top-k FQ# with k
 Constraint ::= BrackettedExpression | BuiltInCall | FunctionCall
 | FlexibleExpression
 FlexibleExpression ::= FuzzyTermExpression | FuzzyOperatorExpression
 FuzzyTermExpression ::= (Var ['=', '!=', '>', '>=', '<', '<='] FuzzyTerm)? [with threshold]
 FuzzyOperatorExpression ::= Var FuzzyOperator NumericLiteral
 FuzzyOperator ::= (Modifier)*FuzzyOperator
 FuzzyTerm ::= FuzzyTerm and FuzzyTerm | FuzzyTerm or FuzzyTerm
 | not FuzzyTerm | ModifiedFuzzyTerm
 ModifiedFuzzyTerm ::= (Modifier)* ModifiedFuzzyTerm | (Modifier)* SimpleFuzzyTerm

Membership function	Scala code	DSL
PieceWise	<code>PieceWiseMembershipFunction (List(ValuePoint(0,1), ValuePoint(2,1), ValuePoint(4,0)))</code>	<code>(0,1) (2,1) (4,0)</code>
Triangular	<code>TriangularMembershipFunction(0,2.5,5)</code>	<code>trian 0 2.5 5</code>
Trapezoidal	<code>TrapezoidalMembershipFunction(0,2,3,4)</code>	<code>trape 0 2 3 4</code>
Sigmoidal	<code>SigmoidalMembershipFunction(-4, 3)</code>	<code>sigm -4 3</code>
Singleton	<code>SingletonMembershipFunction(SingletonPoint(2))</code>	<code>2</code>
...

```
5 match (h:Hotel) return h.name, fuzzyLT("/tmp/price.fl", h.price)
```

 Graph	h.name	fuzzyLT("/tmp/price.fl", h.price)
 Rows	h1	<pre>[{ name cheap degree 1 }, { name midPrice degree 0 }, { name expensive degree 0 }]</pre>
	h2	[

Returned 5 rows in 77 ms.

```
$ match (h:Hotel) with h, h.price as price return h.name, head(fuzzyLT("/tmp/price.fl", price))
```

Graph	h.name	head(fuzzyLT("/tmp/price.fl", price))				
Rows	h1	<table border="1"><tr><td>name</td><td>cheap</td></tr><tr><td>degree</td><td>1</td></tr></table>	name	cheap	degree	1
name	cheap					
degree	1					
	h2	<table border="1"><tr><td>name</td><td>cheap</td></tr><tr><td>degree</td><td>1</td></tr></table>	name	cheap	degree	1
name	cheap					
degree	1					
	h3	<table border="1"><tr><td>name</td><td>midPrice</td></tr><tr><td>degree</td><td>0.5</td></tr></table>	name	midPrice	degree	0.5
name	midPrice					
degree	0.5					
	h4	<table border="1"><tr><td>name</td><td>midPrice</td></tr><tr><td>degree</td><td>1</td></tr></table>	name	midPrice	degree	1
name	midPrice					
degree	1					
	h5					

Returned 5 rows in 77 ms.


```
$ match (h:Hotel) with h, h.price as price return h.name, head(fuzzyLT("/tmp/price.fl", price)).name
```

 Graph	h.name	head(fuzzyLT("/tmp/price.fl", price)).name
	h1	cheap
 Rows	h2	cheap
	h3	midPrice
	h4	midPrice
	h5	expensive
	Returned 5 rows in 49 ms.	

```
$ match (h:Hotel) with h, h.price as price With h, head(fuzzyLT("/tmp/price.fl", price)) as priceTerm set h.priceCat = pric...
```



Graph

h.priceCat**collect(h.name)**

cheap

[h1, h2]



Rows

expensive

[h5]

midPrice

[h3, h4]

Set 10 properties, returned 3 rows in 719 ms.

mfdistance.png

[Click here to download high resolution image](#)

```
$ _match (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance return h.name, distance order by distance desc
```



	h.name	distance
 Graph	h5	1
 Rows	h4	0.875
	h3	0.25
	h1	0
	h2	0

Returned 5 rows in 64 ms.

```
$ match (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance return h.name, distance order by distance desc
```

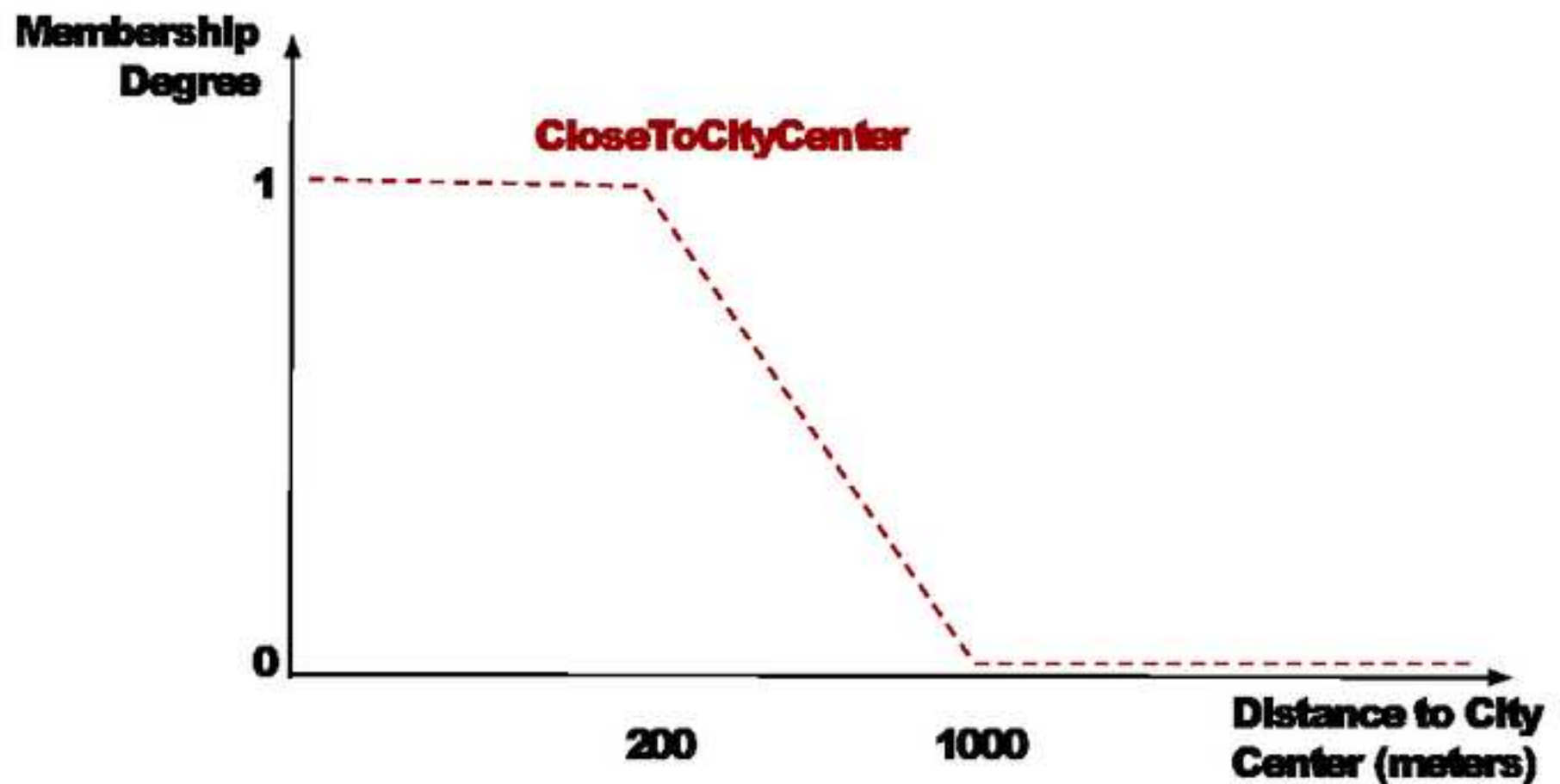
	h.name	distance
 Graph	h5	1
 Rows	h4	0.875
	h3	0.25
	h1	0
	h2	0

Returned 5 rows in 64 ms.

```
$ MATCH (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance WHERE distance > 0.8 RETURN h.name, distance 0...
```

 Graph	h.name	distance
	h5	1
 Rows	h4	0.875
Returned 2 rows in 45 ms.		

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

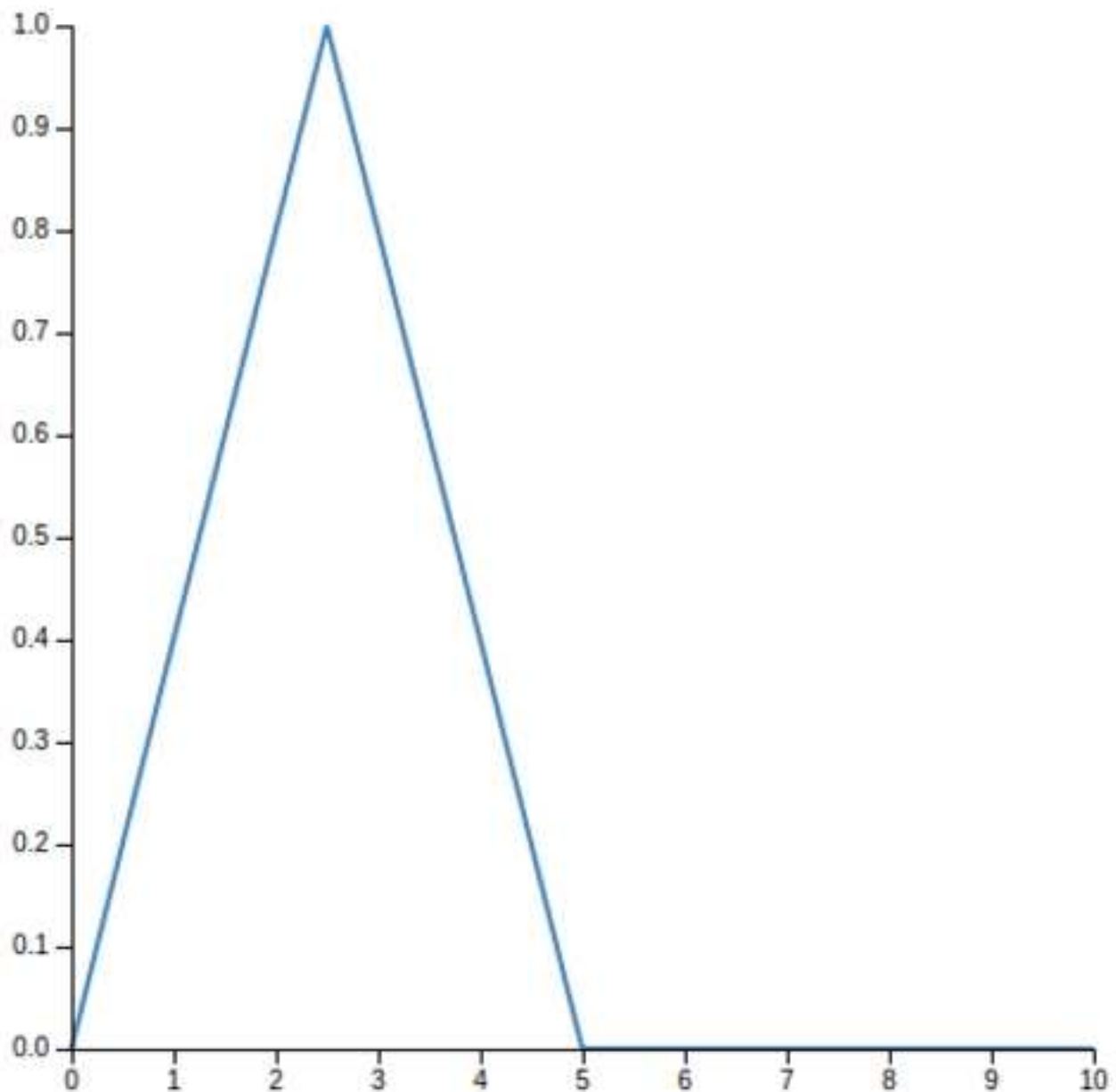


Tape your memberfunction definition

trian 0 2.5 5

4.3

Search



For membershipfunction trian 0 2.5 5 value for 4.3 is 0.28

asc-aca.tex

[Click here to download LaTeX Source Files: asc-aca.tex](#)

biblio-old.tex

[Click here to download LaTeX Source Files: biblio-old.tex](#)

biblio.bib

[Click here to download LaTeX Source Files: biblio.bib](#)

neo4j-peg.tex

[Click here to download LaTeX Source Files: neo4j-PEG.tex](#)

sef.tex

[Click here to download LaTeX Source Files: sef.tex](#)