Benchmarking database systems for social network applications *

Renzo Angles
Department of Computer Science, Universidad de Talca
Department of Computer Science, VU University Amsterdam
rangles@utalca.cl

Arnau Prat-Pérez
DAMA-UPC
Universitat Politécnica de Catalunya
aprat@ac.upc.edu

David Dominguez-Sal
Sparsity Technologies
david@sparsity-technologies.com

Josep-LLuis Larriba-Pey
DAMA-UPC
Universitat Politécnica de Catalunya
larri@ac.upc.edu

Abstract

Graphs have become an indispensable tool for the analysis of linked data. As with any data representation, the need for using database management systems appears when they grow in size and complexity. Associated to those needs, benchmarks appear to assess the performance of such systems in specific scenarios, representative of real use cases.

In this paper we propose a microbenchmark based on social networks. This includes a data generator that synthetically creates social graphs, and a set of low level atomic queries that model parts of the

*Technical report of the paper presented at the Graph Data-management Experiences and Systems (GRADES) Workshop, co-located with SIGMOD/PODS. June 2013

behavior of social network users. In order to understand how different data management paradigms are stressed, we execute the benchmark over five different database systems representing graph (Dex and Neo4j), RDF (RDF-3X) and relational (Virtuoso and PostgreSQL) data management. We conclude that reachability queries are those that put all the database systems into more difficulties, justifying themselves, and making them good candidates for more complex benchmarks.

1 Introduction

During the last years, there has been a huge increase in the number of applications that query and manipulate graphs. Data from social networks, protein-to-protein networks and the Web, just to cite a few, benefit from being modeled as a graph. Due to the increase in size and complexity of these data sources, it has been essential to have database systems which can handle large graph datasets efficiently.

Although traditional relational database systems can represent a graph as a set of tables, they neither offer a natural interface nor efficient operations. Graph queries, such as finding a path, require complex SQL expressions and produce execution plans with a large number of join operations that can be computationally expensive [3].

Both graph and RDF databases offer more natural graph interfaces than SQL. Graph databases provide the programmer with operations such as getting the neighbors of a node. These systems, such as Dex and Neo4j, offer efficient APIs for these operations and graph oriented query languages (e.g., Cypher). On the other hand, RDF databases store graphs as collections of statements subject-predicate-object called RDF triples. SPARQL is the standard query language for RDF databases, which is based on graph pattern expressions.

Although there are no standard performance benchmarks to assess quantitatively the efficiency of database systems on social network datasets, there are initiatives, like LDBC[1], that work on the design of industry oriented benchmarks. Form the academic point of view, studies on the design framework for graph data base benchmarks including social networks have been performed in the recent years, describing the types of queries that the use cases may require, and showing that the social network use case is the richest in variety [6]. LinkBench [2], is a recent example of a social network

[1]Linked Data Benchmark Council is sponsored by the European Community under ICT-FP7 `http://www.ldbc.eu`

benchmark based on Facebook. In some cases, the load time of the graph is measured [2, 5] and queries based on multi-hop traversals [2, 5] or on the complete traversal of a graph [10] have been used as the paradigm to measure the performance of memory intensive operations. In all those cases, the authors focus on graph topology queries on simple graph schemas, missing some of the characteristics of a specific application such as social networks.

Here we focus on the social network use case, benefiting from its richness to propose a simple microbenchmark that contains very common operations in such networks. The queries of the benchmark represent typical social network usages, such as getting the friends of a friend, looking for similar like pages or finding shortest paths between persons. This benchmark can be used not only for assessing the performance of a graph implementation, but also to build more complex environments. For example, the proposed queries can be combined to build logs of interaction of the users. The queries proposed here are envisioned as the basic primitives from which one could construct more complex graph oriented queries such as page rank, influence or recommendation queries. The benchmark proposes an easy to extend framework that can be implemented by any database system. We also describe the basic social network schema that can be populated with social network-like data, which mimics real-world data distributions found in social networks, such as Facebook. This generation procedure creates both the graph and the query set instances as a data stream, which can be used to build large graphs without large memory requirements.

In this paper, we analyze the proposed queries in order to understand the complexity of social network applications. We execute the microbenchmark on five databases (Dex, Neo4j, RDF-3X, Virtuoso and PostgreSQL) and compare the impact of the queries on those systems. We conclude that graph reachability queries are the most challenging query family in terms of time complexity. We observe that relational databases have performance struggles to compute long paths for such queries and graph and RDF databases show better performance.

The rest of the paper is structured as follows. In Section 2, we present the microbenchmark by describing the data schema, the data generation process, the queries, the generation of test data, and the performance metrics. In Section 3 we describe the experimental setup. In Section 4 we show and discuss the results. Finally, in Section 5 we present some conclusions.

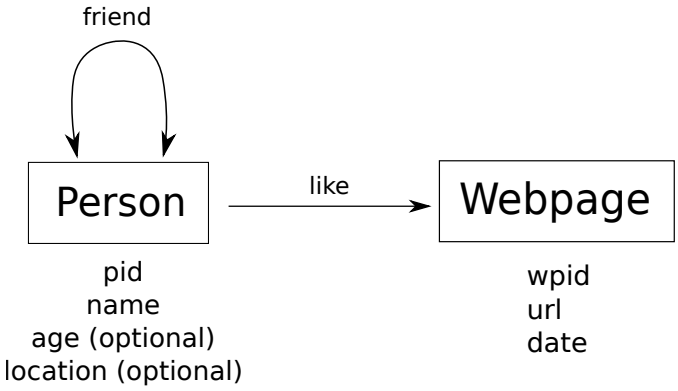

Figure 1: The data schema of the microbenchmark graph.

2 Microbenchmark description

This section presents the microbenchmark including the graph data schema, the data generation method, the query set, test data selection, and the performance metrics.

2.1 Graph data schema

Our data model defines two types of entities as shown in Figure 1: person and webpage. Persons are linked to other persons by a "friend" undirected relationship, and are linked to webpages by a "like" directed relationship. The attributes of a person are the pid (person identifier), the name, and two optional fields, the age and the location. A webpage has attributes wpid (webpage identifier), URL, and optionally a creation date. Note that an instance of this schema is an attributed bipartite graph with directed and undirected edges with two types of nodes and edges.

2.2 Graph data generation.

Stream edge data generator. We have developed a general-purpose graph data generator based on the Recursive Matrix (R-Mat) model [5]. The basic idea behind R-Mat is a recursive procedure to add edges in the adjacency matrix of the graph (starting from an empty matrix), until a given number of edges have been added. The recursive procedure subdivides the adjacency matrix of the graph into four partitions (named a, b, c, d) where the selection of one partition follows a probability ρ_x (e.g., $\rho_a = 0.55$,

$\rho_b = 0.15$, $\rho_c = 0.1$ and $\rho_d = 0.2$), and adds an edge at the end of the recursion (i.e. when a 1×1 matrix has been reached).

We propose a strategy which is based on simulating the R-Mat recursion process. We store the distribution of edges in an array, and construct the graph using such distribution. The advantage of this approach is that it allows the sequential generation of nodes and edges in a streamed fashion, which increases the scalability of the generation process. The explanation is given as follows.

Assume that G denotes the graph to be generated, N the number of nodes, and E the number of edges. First, we construct an array D of length N, such that the value of $D[i]$ defines the number of edges (degree) of node i, for $1 \leq i \leq N$.

Second, we simulate the R-Mat recursive procedure $E = N \log N$ times[2], but we do not materialize the edge. At the end of each recursion we increment the value of both $D[s]$ and $D[t]$ when an edge (s, t) is added. If G is directed, the value of $D[t]$ is not incremented. After the second step, D contains the distribution of degrees for all the nodes in G.

Third, the edges of G are materialized by traversing D. For each node i, we generate a sequence of $D[i]$ edges of the form (i, j) such that $j > i$ and $D[j] > 0$. If G is directed, then the latter condition does not apply. For each generated edge, we decrease the value of $D[i]$ by one. If G is undirected, we also decrease the value of $D[j]$. Note that this procedure avoids generating repeated edges while the original R-Mat algorithm did not avoid such case.

Since the basic R-Mat algorithm produces bumpy histograms, we smooth them by updating the ρ probabilities at each step of the recursion, as suggested in [5]. The probability ρ_x at stage i of the recursion is defined as $\rho_x^i = abs(\rho_x^{i-1} + (0.25 - \rho_x^0) \ / \ \log_2 N)$. Following this procedure, the probability in the last recursion step of the algorithm is $\rho_x = 0.25$. Note that each probability ρ_x is updated $\log_2 N$ times according to the height of the recursion tree.

Social data generation. The generation of social network data is based on the use of information published by current social network applications, for example the Facebook Annual Report of year 2012 [7]. Therefore, we try to produce synthetic data having characteristics present in real-life social networks.

The number of users in Facebook is significantly larger than the number

[2]Studies on the evolution of real networks, and in particular social networks, have shown that the number of edges grows super-linearly in the number of nodes N and below N^2 [8].

of webpages. To simulate this, we set 80% of the nodes as persons and 20% as webpages. The identifier of a person node (i.e., the attribute pid) is an integer value in the range $[1, N \times 0.8]$, where N is the number of nodes in the graph. The names of persons and locations are selected randomly from dictionaries including 5494 first names, 88799 last names, and 656 locations. Hence, the probability of having duplicated pairs of these two attributes depends on the size of the graph and the distributions used. The occurrence of attributes age and location follows probabilities 0.6 and 0.3 respectively. The age of a person is a random integer between 10 and 73. The identifier of a node webpage (i.e., the attribute wpid) is an integer in the range $[(N \times 0.8) + 1, N]$. The attribute URL follows the pattern `http://www.site.org/webpageID.html` where ID is the wpid of the webpage. The probability of including a random creation date for a webpage is 0.6.

Resembling the data generator of the benchmark for the Facebook social graph [2], we use the general-purpose method for graph generation to obtain power-law distributions for the edges corresponding to relationships friend and like.

2.3 Queries

We perform a selection of domain specific queries for the microbenchmark. Our approach is based on the user interaction with Facebook to identify atomic actions that are mapped to the queries of the benchmark. Such interaction includes an analysis of the data displayed in the user profile page, the user wall page, and the profile page of a user's friend.

A microbenchmark is used to evaluate the individual performance of atomic operations (such as joins and aggregations in relational databases), rather than more complex queries [4]. In the context of graphs we find several micro-queries which can be considered atomic and we group them into adjacency, reachability, pattern matching and summarization queries [1]. Also, we add select queries that are relevant in the context of social networks.

Based on an analysis of the user interaction with a social network platform, we defined the query mix shown in Table 1. Query Q1 represents a selection given an attribute value. Queries Q2 and Q3 can be used to test the efficiency when obtaining the adjacent nodes of an edge. They are useful to compare the performance for querying incoming and outgoing edges. Queries Q5, Q6 and Q7 are reachability queries oriented to evaluate the support of a simple path expression with fixed path length. Although the length of the paths is fixed, the direction of the edges can influence the

Q	Description	Type
1	Get all the persons having a name N	Select
2	Get all the persons who like a given webpage W	Adjacency
3	Get the webpages that person P likes	Adjacency
4	Get the name of the person with a given PID	Select
5	Get the friends of the friends of a given person P	Reachability
6	Get the webpages liked by the friends of a given person P	Reachability
7	Get persons that like a webpage which a person P likes	Reachability
8	Is there a connection (path) between persons P1 and P2?	Reachability
9	Get the shortest path between persons P1 and P2	Reachability
10	Get the common friends between persons P1 and P2	Pattern matching
11	Get the common webpages that persons P1 and P2 like	Pattern matching
12	Get the number of friends of a person P	Summarization

Table 1: The queries of the microbenchmark with their description and classification.

performance of a system. Moreover, the evaluation of adjacency queries is strongly influenced by the intermediate results (i.e., the degree of the nodes).

The support for recursive queries is the objective of queries Q8 and Q9. Query Q8 is simpler than query Q9 because the former requests a connectivity test and Q9 searches the shortest path. However, in our experiments we did not detect that any of the systems under test performed such optimization. Note that recursive queries are well-know difficult-to-solve examples for database systems implementing join operations [10].

Queries Q10 and Q11 are common graph pattern matching queries. Note that a simple graph pattern (Q11) can be converted into a complex one (e.g., the cyclic pattern of Q10) just by changing the direction of the edges. Finally, query Q12 is an example of a common aggregate operation that most database systems support.

Note that these essential queries can be composed and/or grouped in order to describe more complex queries. Additionally, the order of the queries can be controlled to construct a query mix that reflects the workflow of a user interacting with a social network platform. Such composite operations are out of the scope of the present article.

2.4 Test data generation

In addition to the social network data file, the generator produces an XML file containing test data, that is data to be used in the creation of query instances. For example, if the benchmark issues 10,000 instances of Q1,

then the data file contains a list of 10,000 names.

The test data is grouped as follows: IDs of people (used for queries Q3, Q4, Q5, Q6, Q7, Q12); names of people (used for query Q1); IDs of webpages (used for query Q2); pairs of IDs <person,person> such that these two people are connected by a "friend" relationship (used for query Q10); pairs of IDs <person,webpage> such that there is a relationship "like" between the person and the webpage; and pairs of IDs <person,person> such that there is a path between them (used for queries Q8 and Q9).

The selection of test data runs in parallel to the graph data generation process. Hence, the data is generated as a stream of query instances. For all queries but Q8-Q9, the selection is based on dividing the data space in equally spaced slices of the stream, taking a sample at the beginning of each slice. Considering that the data space is the set of nodes ordered by degree, this method ensures that we obtain test nodes of several degrees.

In the case of Q8-Q9, the process selects as many nodes as query instances are needed, which are "seeds for the paths", before reading the stream. Then, the process will connect to the stream of edges and will simulate random walks starting from the seeds. Each edge will extend a path if any of the nodes is the end of a seeded path, otherwise the edge is ignored. In Appendix B we show the distribution of such paths.

2.5 Performance metrics and Indexes

As an academic benchmark, we only consider execution times, and not price per transaction or transactions per unit of time. In this paper, we measure the response time as a metric.

Data loading Time. The time required to load the data from the source file. This metric includes any time spent by the system to build index structures and statistical data.

Query execution time. This is the central performance metric of the benchmark. It refers to the time spent by a database system to execute a single query. The execution time of a query Q is given by the average time of executing several instances of Q.

Data indexes. Since the benchmark can be implemented in several environments, it allows indexes for any of the data attributes and relations. We do not limit the type of indexes created as long as their construction time is accounted during the graph load time.

3 Experimental setup

The database systems selected cover graph, RDF and relational databases. The graph databases chosen are Dex (v4.7) [6] and Neo4j (v1.8.2 Community). As a representative of RDF stores, we chose RDF-3X [9]. We chose PostgreSQL (v9.1) and Virtuoso (7.0) as representatives of relational databases. The first is a row based database, while the second is a column store with extensions for expressing graph-like queries in SQL.

The benchmark was implemented for all the tested systems on Java 1.6. The test-drivers for Dex and Neo4j were implemented by using their respective Java APIs, hence the database is embedded into the application. We have two different implementations for Neo4j, named NeoAPI and NeoCypher. In NeoAPI, the queries were implemented by using the query functions of the API. In NeoCypher, the queries were expressed in the Cypher query language. RDF-3X, Virtuoso and PostgreSQL were evaluated as system services through the corresponding Java drivers. The query languages SQL and SPARQL were used in relational and RDF databases respectively. In the case of Virtuoso, we use the TRANSITIVE extension to implement graph traversals by means of transitive queries. We use prepared statements in NeoCypher, Virtuoso and PostgreSQL (i.e., each query was precompiled and parameterized to save the overhead of compilation).

The social graph schema was modeled in relational databases with the following tables: Person(id, name, age, location), Webpage(id, url, creation), Friend(pid1, pid2) and Like(pid, wpid). We created indexes for primary keys and attributes, according to the query requirements of the benchmark. For RDF, we generate URIs for nodes and create RDF triples for attributes and edges. Note that we measure the systems with the default configuration provided by the vendor, which means that they might achieve better results if consciously tuned.

We have tested the systems against graphs ranging from 1K nodes to 10M nodes. The datasets used in this paper are described in Appendix A. All systems were able to load the graphs at a speed of 50k-100k objects per second after using bulk loading for RDF-3X, PostgreSQL and Virtuoso, and API loaders for Dex and Neo4j. In this paper, we will not focus on the load time of the systems.

Due to space restrictions, in this paper we show the results for six out of the twelve queries. The detailed results will also be available online in the website of the authors[3]. We have chosen Q1, Q3, Q6, Q9, Q11 and Q12 as

[3] `http://ing.utalca.cl/~rangles/gdbench`

representatives of each of the query types described in Section 2. For each query, we run 10K query instances and we report the average execution time of three consecutive runs. In order to obtain measures similar to those of a working server, the benchmark executes a hot warm-up run with the same query instances and parameters as the main three runs just before them.

To run the benchmark, we have used a computer with the following characteristics: Intel Xeon E5530 CPU at 2.4 Ghz, 32GB of Registered ECC DDR3 memory at 1066 Mhz, a 1Tb hard drive with an ext3 file system. The operating system was a Linux Debian with 2.6.32-5-amd64 kernel.

4 Experimental results

Figure 2 shows the results obtained by the different systems on the selected queries. Broadly speaking, better performance results are obtained on graph databases compared with relational or RDF technology. Among the test implementations, Dex and Neo4j executed the queries in the shortest time and show similar scalability profiles, though Dex is the fastest overall. Furthermore, the introduction of a graph query language in the benchmark is not a prohibitive cost. In general, the results of Neo4j when using the Cypher query language are slower than the API counterpart, but the scalability is similar to the native implementation.

According to the results presented in Figure 2, the queries that stress more all the database systems are Q6 and Q9 which address reachability. While all the systems scale well independently of the graph size for non-reachability queries, the execution time for Q6 and Q9 increases significantly with the number of objects in the graph.

One interesting aspect to consider is how the execution times of the query instances vary with respect to the characteristics of the parameters of the query. For instance, we have analyzed the impact of the degree of the source node in query Q6 (i.e., the number of friends of the source node person). Figure 3 shows the execution times for the different instances of Q6, for the graph with 10M of nodes, for each of the systems. Based on the degree of the source node, we have classified the instances in three groups: degree between 1 and 10; between 11 and 100; and between 101 and 1000 (the degree distribution of the source nodes used in query Q6 is shown in Appendix B). We observe that the execution time of the instances is dependent on the degree of the node for all the tested systems.

In the case of Q9, which is the shortest path query, we see that graph databases obtain the best results. This shows that Dex and Neo4j exploit

their graph-oriented structures, plus a good implementation of the breath-first search algorithm (as described in the documentation of the systems and confirmed by the developers). On the other side, we see that RDF-3X, Virtuoso and PostgreSQL have severe scalability problems, showing that relational and RDF databases are less specialized for path-traversal oriented graph queries.

In order to better understand the behavior of the different systems when computing query Q9, the average cost per path length taken by each database system is shown in Figure 4, and the path length distributions of the shortest-path queries for different graph sizes is shown in Figure 5. Figure 4 shows that all the database systems follow a similar trend: the larger the graph and the longer the path to compute, the larger is the time needed to execute the query. The query workload for Q9 is dominated by short paths (i.e., one and two hops, as seen in Figure 5). This masks the deteriorating performance of PostgreSQL and RDF-3X for long paths, as shown in Figure 2 and Figure 4.

Looking at PostgreSQL, we see a large variability for the largest graphs in Q9 as shown in Figure 2. PostgreSQL is very sensitive to the length of the paths in the query instance set. When these paths are up to three hops long, the system scales well. However, with paths of size four or more, the time needed for the computation increases significantly. The explanation for this behavior is a consequence of the implementation of the shortest-path in SQL. Graph traversal queries in SQL imply the generation of query plans with recursive joins. The reported implementation (which was the fastest in PostgreSQL) is a sequence of six SQL queries, each one asking for a path of a given length, from one to six. An alternative implementation is the usage of the WITH RECURSIVE operator that facilitates the traversal of hierarchies. However, the current implementation computes all the paths and does not stop once a path is found. Since many instances of Q9 contain short paths, the former implementation is more efficient.

Although Virtuoso is not as efficient as graph databases, it exhibits better scalability compared to PostgreSQL, thanks to the use of transitive subqueries implemented by means of the TRANSITIVE operator. Although this operator is not really optimized for memory efficiency (issue confirmed by the developers of Virtuoso), it allows to stop the evaluation when the first solution is found. This feature can explain the best behavior in comparison with PostgreSQL. In Appendix C we discuss results of evaluating PostgreSQL and Virtuoso using stored procedures.

RDF-3X shows a very good performance as long as the paths are just one hop long. However, for paths longer than one, its performance decreases

significantly as long as the size of the database increases, standing between Virtuoso and PostgreSQL in terms of performance. An important issue with the evaluation of RDF databases is the cost of translating between external (URIs) and internal identifiers. In the simpler queries, the execution time can be strongly influenced by such translation.

5 Conclusions and Future Work

In this paper, we have proposed and described a microbenchmark for database systems based on social networks. The benchmark proposes and implements a graph generator to synthetically generate graphs with social network characteristics like Facebook, and a set of atomic queries that mimic the typical usages in social network applications.

The query set includes several types of queries that are common in social networks: selection, adjacency, reachability, pattern matching and summarization queries as part of more complex queries or actions by the user. All database systems tested in the paper were able to complete the queries in a reasonable time with the exception of the reachability queries, which were found to be the most stressful queries for all the systems. The relational database systems were not able to compute those queries in a reasonable time when the number of hops for the traversal was larger than 4. Overall, graph databases benefit from the benchmark for all the query types.

In the near future, we will propose a new benchmark based on the queries of this work. This new benchmark will stress the concurrent user sessions that social networks have to process. One such new query will be a composite of several smaller queries that will interact to form a user session like those exercised in a real world environment. Moreover, we expect to include update operations as part of the workload.

6 Acknowledgements

The members of DAMA-UPC thank the Ministry of Science and Innovation of Spain and Generalitat de Catalunya, for grant numbers TIN2009-14560-C03-03 and GRC-1187 respectively, and IBM CAS Canada Research for their strategic research grant. David Dominguez-Sal thanks the Ministry of Science and Innovation of Spain for the grant Torres Quevedo PTQ-11-04970. The members of UPC and VUA would like to thank the European Community's Seventh Framework Programme FP7/2007- 2013 for funding the LDBC project. Renzo Angles is funded by Fondecyt Chile grant 11100364.

Finally, the authors would like to thank Orri Erling for his guiding on implementing the Virtuoso test drivers and stored procedures.

References

[1] R. Angles. A comparison of current graph database models. In *ICDEW*, pages 171–177, 2012.

[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkench: a database benchmark based on the facebook social graph. In *ACM SIGMOD*, 2013 (To appear).

[3] J. Biskup and H. Stiefeling. Transitive closure algorithms for very large databases. In *Workshop on Graph Theoretical Concepts in Computer Science*, 1988.

[4] H. Boral and D. J. DeWitt. A methodology for database system performance evaluation. *SIGMOD Record*, 14(2):176–185, 1984.

[5] D. Chakrabarti, Y. Zhan, and C.s Faloutsos. R-mat: A recursive model for graph mining. In *ICDM*, 2004.

[6] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazan, and J. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *IWGD*, pages 37–48, 2010.

[7] Facebook. Annual report 2012 (10k). 1 Feb. 2013.

[8] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1):2, 2007.

[9] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB Journal*, 19(1):91–113, 2010.

[10] Mihalis Yannakakis. Graph-Theoretic Methods in Database Theory. In *PODS*, pages 230–242. ACM Press, 1990.

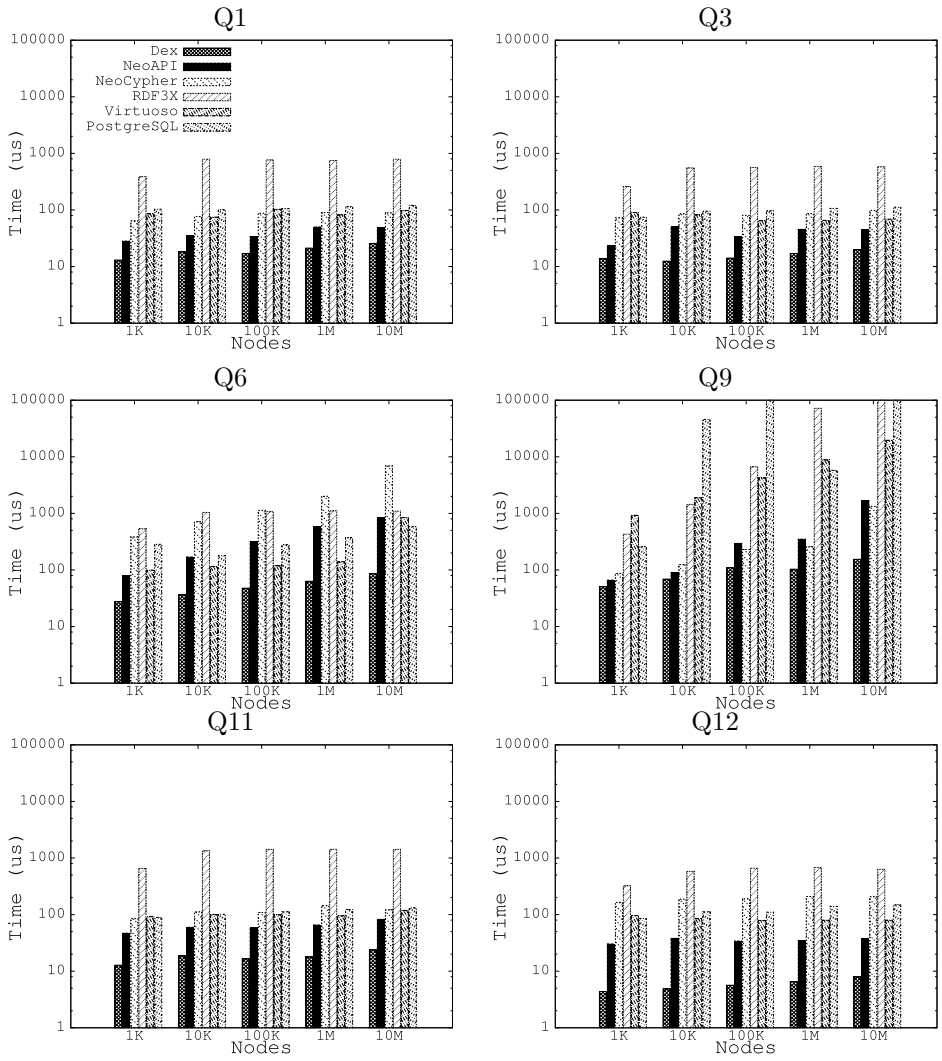

Figure 2: Time in microseconds for queries Q1, Q3, Q6, Q9, Q11 and Q12 for graphs varying from 1,000 to 10M nodes.

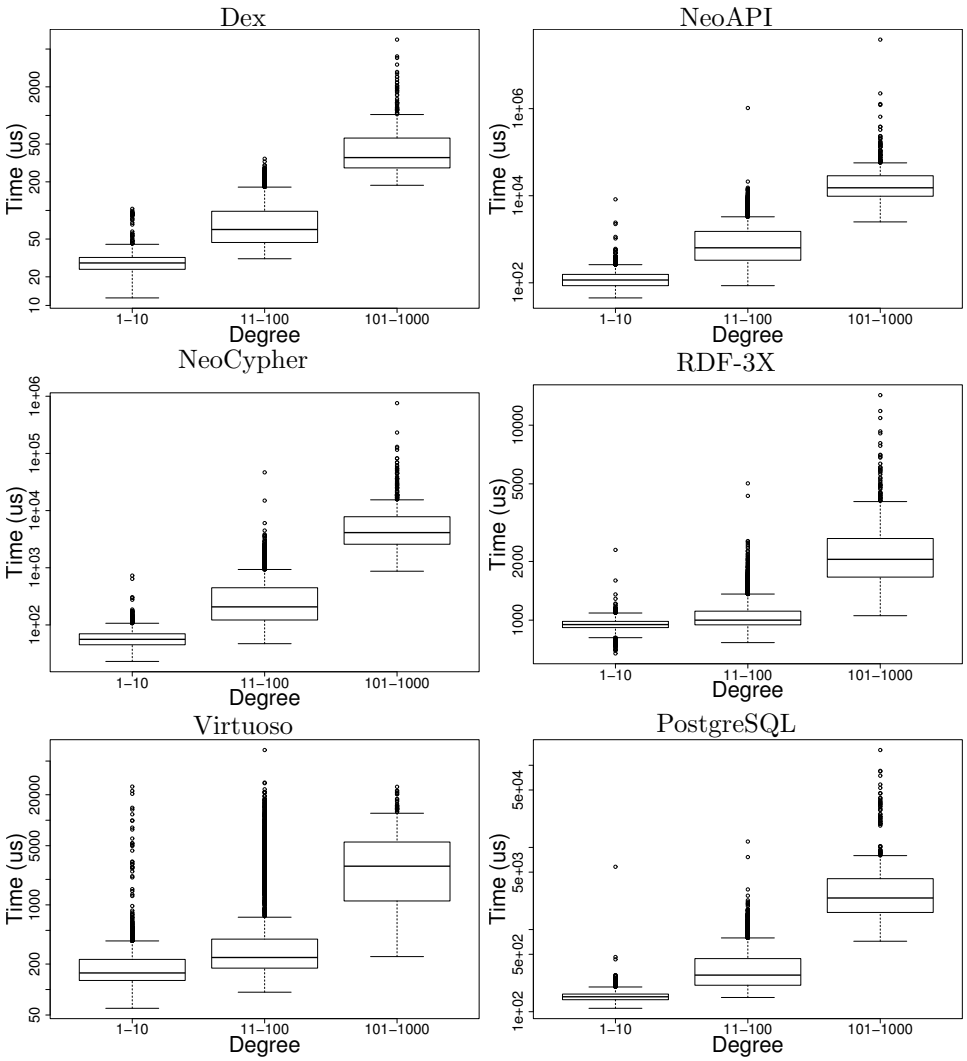

Figure 3: Time in microseconds for Q6 for different systems grouped by the degrees of the input nodes for the 10M graph.

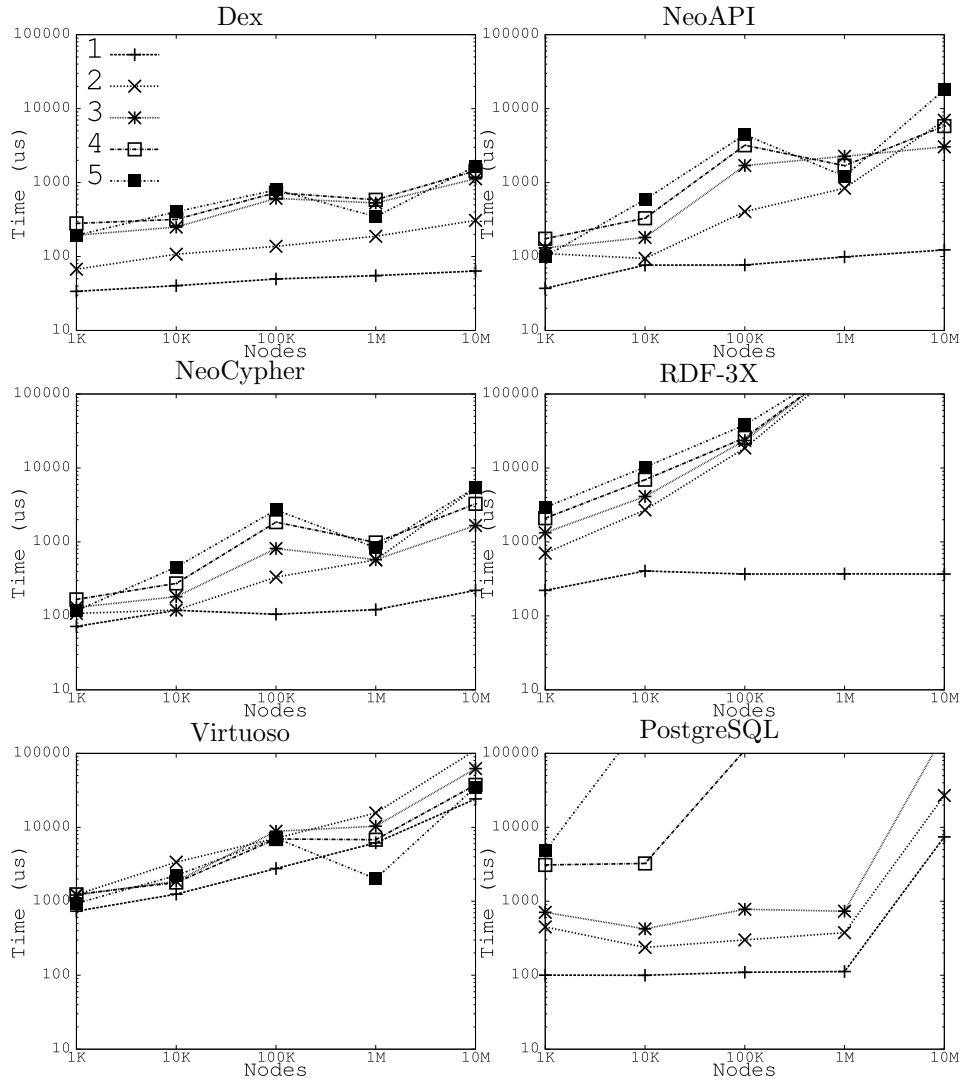

Figure 4: Time in microseconds for computing the shortest path algorithm used in Q9. The plots show different path lengths (from 1 to 5) in the plot lines and graph sizes in the horizontal axis.

A Datasets

Table 2 shows characteristics of the graphs generated for the tests, as well as the size in megabytes for each one.

Nodes	Edges	Dex	Neo4j	PSQL	Virt.	RDF-3X
1K	7K	4M	1M	44M	33M	2M
10K	92K	8M	8M	96M	39M	7M
100K	1.2M	66M	82M	447M	177M	67M
1M	14M	706	942M	4.2G	915M	706M
10M	161M	7.7G	11.5G	38G	8.6G	8.2G

Table 2: Datasets used in the evaluation of the benchmark.

B Path lengths and degree distribution for queries

Figure 5 shows the distributions of the length of the paths, for different graph sizes, used in the evaluation of shortest-path queries (query Q9). We see that the distribution of the paths generated is similar for different graph sizes, and that query instances looking for short paths are more common than those looking for long paths. This simulates that it is more likely to search people who are friends or friends-of-a-friend, than random people.

Figure 6 shows the distribution of the degrees of the nodes used in the query instances of query Q6, for the graph of 10M nodes. We see that there are more query instances querying from nodes with low degree than with large degree, which is expected since the degree of the nodes of the graphs follow a power law distribution.

C Stored Procedures

For the tests described in the above sections, PostgreSQL and Virtuoso are installed as services to which the benchmark connects and sends the query, as opposed to Dex and Neo4j, where the databases are embedded into the executable. This means that the queries' parameters have to be sent to the engine. This places PostgreSQL and Virtuoso in a disadvantageous situation, specially for those simpler queries, where the cost to connect to the database, send the query, and get the results can be high with respect to the cost of actually performing the query. In order to determine the impact of this overhead and to see how this affects the scalability of both systems

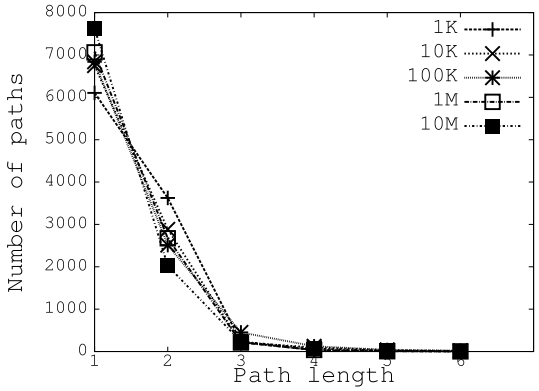

Figure 5: Path distribution, by length and graph size, used in shortest-path queries (Q9).

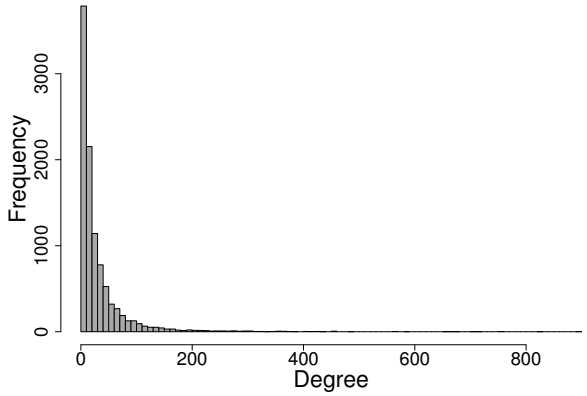

Figure 6: The degree frequency histogram of the nodes used in query Q6, for the 10M graph.

for the different queries, we have implemented them as stored procedures for both relational databases. In this case, the parameters of the queries are stored in a table, from which the stored procedure reads the data, executes the query, and stores the result in the database. Although this execution model does not exactly fit with that proposed by the benchmark (i.e. an environment where the users execute different queries during their session, so we can not have the queries pre stored in the database), this allows us to remove any overhead incurred by the benchmark and focus purely on the cost of the query. Figure 7 shows the results obtained by executing the queries using stored procedures. We see that, for the simpler queries (i.e. Q1, Q3, Q6, Q11 and Q12), both systems have seen a notable improvement in their

average execution time, specially Virtuoso, which obtains results comparable to those obtained by the graph databases. However, in the case of Q9, the times obtained are similar to those obtained by the benchmark. This confirms the results that relational databases can obtain excellent results as far as the queries do not involve traversing the graph further than two hops.

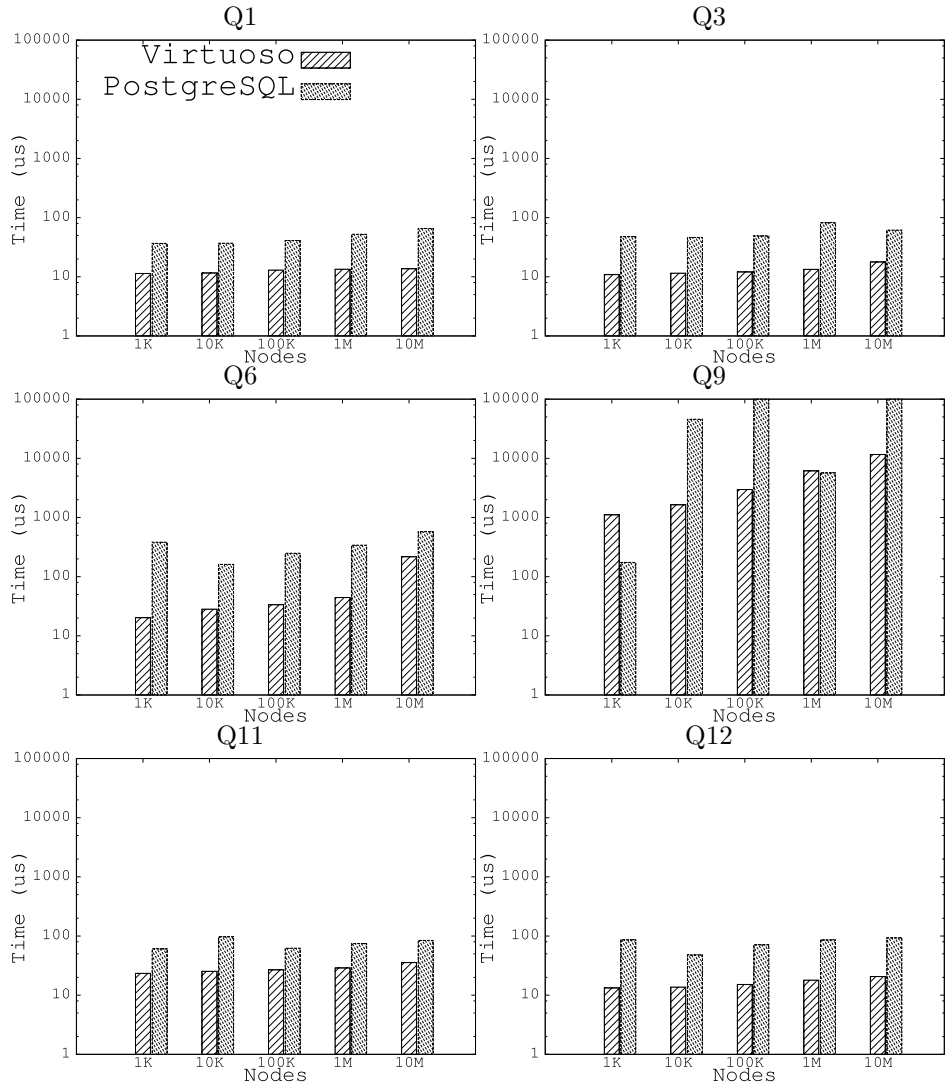

Figure 7: Time in microseconds for queries Q1, Q3, Q6, Q9, Q11 and Q12 for graphs varying from 1,000 to 10M nodes for Virtuoso and PostgreSQL using stored procedures.