Choosing Between Graph Databases and RDF Engines for Consuming and Mining Linked Data

Domingo De Abreu, Alejandro Flores, Guillermo Palma, Valeria Pestana, José Piñero, Jonathan Queipo, José Sánchez, Maria-Esther Vidal

Universidad Simón Bolívar, Caracas, Venezuela
{dabreu, aflores, gpalma, vpestana, jpinero, jqueipo, jsanchez, mvidal}@ldc.usb.ve

Abstract. Graphs naturally represent Linked Data and implementations of graph-based tasks are required not only for data consumption, but also for mining patterns among links. Despite efficient graph-based algorithms and engines have been implemented, there is no clear understanding of how these solutions may behave on Linked Data. We evaluate both general purpose graph database and state-of-the-art RDF engines, and our experimental results reveal characteristics of linked datasets and graph-based tasks that may affect their performance. These results can be considered as a further step for solving the problem of choosing between graph databases to consume and mine Linked Data.

1 Introduction

Graphs are commonly used to represent linked data, and several efficient algorithms have been proposed not only to consume, but also to mine Linked Data. For example, Saha et al. [16] and Thor et al. [18] have defined densest subgraphs and graph summarization techniques to identify patterns between linked datasets of genes. Further, algorithms for finding subgraph isomorphisms and frequent subgraphs have been extensively studied in the literature [1]. The majority of these algorithms are computationally complex, and rely on core graph-based tasks to solve graph reachability, traversal, adjacency and pattern matching.

Additionally, a variety of engines have been developed to manage, store and query graph databases. Particularly, in the context of the Semantic Web, engines have been defined to store and consume RDF graphs. However, existing RDF engines focus on individual triples, more than providing a graph oriented representation of the data, and the implementation of core graph-based tasks can be time-inefficient because of the number of potential self-joins required to execute these tasks. On the other hand, several general purpose graph systems have been defined to manage graph databases; also, APIs are usually available to execute core graph-based tasks on these databases. However, many of the existing graph engines do not exploit the properties of RDF, and pattern matching queries on RDF-based graphs may be inefficient. Therefore, it is important to conduct evaluations that uncover the properties and limitations of existing graph engines, and that provide a further step for solving the problem of choosing the most appropriate engine to execute a given task against linked datasets.

In this paper we empirically evaluate the performance of three general purpose graph database engines: DEX [11], Neo4j [15] and HypergraphDB [8]. Additionally, we study RDF-3x [14] a *best-of-breed* RDF engine [7]. During the evaluation, we analyze the performance of these engines on tasks of adjacency and reachability. Further, we study different implementations of the algorithms to find shortest paths and *k-hops*; also, graph traversals following breadth-first and depth-first search strategies are evaluated. Finally, we analyze the performance of pattern matching queries, and the mining tasks of densest subgraphs and graph summarization algorithms. A variety of synthetic graphs are studied; graphs with different density and size are generated using existing graph generators (e.g., GTgraph[1] and the Berlin SPARQL Benchmark (BSBM) [3]). We describe the variables and configuration setups that impact on the performance of the studied engines when the above mentioned graph tasks are executed against the generated graphs. Finally, we discuss the results of our evaluation, and outline the different conditions that benefit the studied engines. Results of the experiments are published at `http://graphium.ldc.usb.ve`.

To summarize, our contributions are as follows: *i*) a characterization of graph properties and tasks that impact on the performance of existing graph engines; *ii*) benchmarks comprised of graph-based tasks and a variety of graphs to evaluate existing graph engines; and *iii*) an empirical study of the performance of graph engines on tasks for consuming and mining linked data.

This paper is organized as follows: Section 2 summarizes related works, and Section 3 describes the studied graph engines and tasks. Section 4 illustrates parameters that impact existing graph engines, while experimental results are reported in Section 5. Section 6 concludes and outlines our future works.

2 Related Work

Existing RDF engines effectively address the challenges of consuming RDF data. However, independent evaluations [7, 10] suggest that thanks to physical structures and its execution techniques, RDF-3x is able to overcome existing engines. Based on these results, we selected RDF-3x as an exemplar of RDF engines.

Several engines address the problem of managing large persistent graphs (e.g., DEX [11], Neo4j [15], and HypergraphDB [8]). Commonly, these engines provide APIs and rely on physical structures to speed up the execution of graph-based tasks. We evaluate their main functionalities and propose a set of tasks whose evaluation uncovers useful insights that suggest some of their limitations.

Benchmarking has contributed to the improvement of systems and in general, of existing areas by the means of defining fair and neutral benchmarks. Particularly, in the context of Databases, the family of the Transactional Processing Performance Council (TPC) benchmarks[12] were developed for database-centric standards and for the evaluation of system performance under clearly defined dataset conditions and workloads. Recently, the TPC family has been extended

[1] `http://www.cse.psu.edu/~madduri/software/GTgraph/index.html`

to other domains, e.g., virtualization, multi-source data integration, and graph analysis in multiprocessors. Although some of the new released TPC benchmarks attempt to provide graph generation standards [5], the graph-based tasks are mainly focused on network centrality while core graph-based tasks cannot be evaluated, e.g., pattern marching, adjacency. Other initiatives have impulsed the study of performance of graph database engines in current cloud service infrastructures and mostly networking tasks, e.g., [2] and [4]; and recently, the LDBC[2] council was created to work on benchmarks for Linked Data. Finally, benchmarks have been defined to test existing RDF engines: LUBM [6], the Berlin SPARQL Benchmark [3], the RDF Store Benchmarks with DBpedia[3], and the SP^{2}Bench SPARQL benchmark [17]. Similarly, we tailored a family of graph characteristics and tasks that allow us to reveal properties of existing RDF/Graph engines.

3 Existing Graph Databases and Graph-Based Tasks

3.1 Graph Database and RDF Engines

DEX [11] is a database engine that relies on labeled attribute multigraphs to model linked data. To speed up the different graph-based tasks, DEX offers different types of indexing: *i*) attributes. *ii*) unique attributes, *iii*) edges to index their neighborhood, and *iv*) indices on neighborhoods. A DEX graph is stored in a single file; values and resource identifiers are mapped by mapping functions; and maps are modeled as B+-tree. Bitmaps are used to store nodes and edges of a certain type. DEX provides an API to create and manage graph datasets. Different functions are offered to efficiently traverse and select graphs: *i*) sub-graphs whose nodes or edges are associated with labels that satisfy certain conditions, *ii*) explode-based methods that visit the edges of a given node, and *iii*) neighbor-based methods that visit the neighbors of a given node.

Similar to DEX, Neo4j [15] represents graphs as labeled attribute multigraphs implemented in native structures. Neo4j can manage several types of indices on nodes and relationships, and graphs can be traversed by following common policies of breadth- and depth-first. Neo4j provides an API, but Cypher can be also used to specify pattern matching queries.

HypergraphDB [8] models graphs as hypergraphs, a graph generalization where several edges correspond to a hyperedge. Hyperedges connect ordered sets of nodes, and can point to other hyperedges. Data is stored in the form of key-value pairs on top of BerkeleyDB, and is indexed using B-trees. Different access methods and indices are provided to speed up core graph-based tasks, e.g., indices on edges or neighborhoods.

Finally, RDF-3x [14] relies on a native index system to efficiently store RDF data and reduce the overhead of I/O operations. RDF data graphs are implemented as a *Triple* table where each labelled edge of an RDF graph is represented

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

[3] http://www4.wiwiss.fu-berlin.de/benchmarks-200801/

as a tuple of the table. A mapping dictionary is used to encode literals by short number IDs, and compressed clustered B+-trees are used to index data in lexicographic order. Six permutations of subject, predicate and object are considered to create an index on each permutation. Additionally, RDF-3x implements optimization and execution techniques that support efficient and scalable execution of SPARQL queries. Further, it exploits operating system features to manage *cache*, and it is able to load portions of intermediate results in resident memory. Thus, RDF-3x speeds up execution time of queries where previous intermediate results can be reused, and it is considered a *best-of-breed* RDF engine [7].

3.2 Graph-Based Tasks

We describe a set of graph-based tasks and the main properties of our implementations. Source code is available at `https://github.com/gpalma/gdbb`.
Graph Creation: consists in measuring the execution time consumed by the engines during the creation and storage of the internal representation of a graph.
Adjacency: checks node adjacency, given that two nodes are adjacent if there is an edge between them. The adjacency tasks are as follows: *i*) *adjacentXP:* given a node x and an edge label p, find the set of adjacent nodes y. *ii*) *adjacentX:* given a node x, find the set of adjacent nodes y. *iii*) *edgeBetween:* given two nodes x and y, find the set of labels of the edges between x and y. These tests are implemented using the nodes/edges adjacency facilities provided by the engines' APIs. In the case of RDF-3x, these tasks are expressed as SPARQL queries. During the evaluation nodes are randomly selected.
Reachability: we study two traversal algorithms: *Breadth-first search* (BFS) and *Depth-first search* (DFS), and implement them using basic adjacency methods of the engines' APIs. We refer to these implementations as `external BFS` and `external DFS`. Additionally, we use reachability methods provided by the engines; we name these implementations `internal BFS` and `internal DFS`. In general, given two nodes x and y, the problem is to *decide* if there is a path between them in the graph. Also, we implement a reachability task named *k-hops*, which given a starting node x, retrieves the set of nodes S such that there is a path of length k from x to y, i.e., paths following out-going links. We implement three versions of *k-hops*. The first is `internal` *k-hops*, and is built using the engines' methods to implement *k-hops*; this implementation relies on graph internal representations and indices to speed up the *k-hops* computation. The second is `external` *k-hops*, implemented with a BFS algorithm with bounded depth, using adjacency methods of the APIs. Finally, *k-hops* is expressed as SPARQL queries in RDF-3x. During the evaluation, nodes are randomly chosen.
Pattern matching: finds all subgraphs that are isomorphic to a pattern graph. The following tasks are implemented in RDF-3x using SPARQL queries: *i*) *adjacentXP*, *ii*) *adjacentX*, *iii*) *edgeBetween*, *iv*) *2-hops*, *v*) *3-hops*, and *vi*) *4-hops*.

Additionally, we implement two main mining algorithms: *densest subgraphs* and *graph summarization*. Conceptually, these tasks are described as follows:

Densest Subgraph: Let $G = (V, E)$ be a directed graph, given subsets S and T of vertices, the density of the subgraph is defined as $d(S,T) = \frac{|E(S,T)|}{\sqrt{(|S||T|)}}$ where *E(S,T)* is the set of edges going from S to T. The *Densest Subgraph problem* is to find a subgraph of maximum density. We implement, for each graph database, the linear time 2-approximation algorithm for the densest subgraph problem in directed graphs proposed by Saha et al.[16]. This implementation relies on nodes/edges adjacency functions provided by the APIs of methods.
Graph Summarization: Let G be a graph, the goal is to produce a graph summary representation of G. A graph summary is an aggregate graph in which each node corresponds to a set of nodes in G, and each edge represents the edges between all pairs of nodes in the two sets. We implement a greedy algorithm proposed by Navlakha et al.[13] to build highly compressed graph representations. This implementation is built on top of the nodes/edges adjacency functions provided by the APIs of the studied graph databases.

4 Analysis of the Benchmark Characteristics

We analyze the main parameters that affect the performance of a graph database engine. Parameters can be independent and dependent. Independent parameters represent characteristics that impact on the behavior of an engine; they need to be specified to ensure reproducibility of an evaluation, as well as, fair and neutral tests. Independent variables have been grouped into three dimensions: graph characteristics and representations, and task implementations. Dependent parameters correspond to the characteristics that are normally impacted by independent parameters. For instance, *i*) *Task Execution Time*: elapsed time between the submission of the task and the generation of the complete answer. *ii*) *Main Memory*: amount of main-memory required to execute a task. *iii*) *Secondary Memory*: amount of secondary-memory required to store the internal representation of a graph. Table 1 summarizes independent and dependent parameters.

First, we can observe that size, density, in- and out-degree of nodes and distribution of the labels, all impact in the time required to create a graph, and in the amount of both main- and secondary-memory consumed during the creation and storage of the graph. Additionally, these properties affect execution time of any graph traversal and reachability task, as well as summarization and densest subgraph. To illustrate how graph characteristics can affect the performance of a graph database engine, lets consider graphs in Table 2. DSJC.1, DSJC.5 and DSJC.9 are randomly generated graphs proposed by Johnson et al. [9] as instances to solve the graph coloring problem[4]; the number of nodes remains the same along the three graphs. Additionally, the family of the graphs Fixed-number-arcs-0.X were generated using GTgraph.[5]; the number of arcs is fixed while density and number of nodes vary. Table 3 reports on the execution time (msecs.), main-memory consumption (MB) and secondary-memory (MB) required to store the

[4] https://sites.google.com/site/graphcoloring/vertex-coloring
[5] `http://www.cse.psu.edu/~madduri/software/GTgraph/index.html`

Table 1. Variables that impact graph engines. Graph Characteristics (GC); Graph Representations (GR); and Task Implementation (TI)

	Independent Parameters	Dependent Parameters		
		Task Execution Time	Main Memory	Secondary Memory
GC	Graph Density	✓	✓	✓
GC	# Nodes	✓	✓	✓
GC	# Edges	✓	✓	✓
GC	Edge Label Distribution	✓	✓	✓
GC	# Node In- Out-Degree	✓	✓	✓
GR	Compressed Structures	✓	✓	✓
GR	Indices	✓	✓	✓
GR	Transactional management	✓	✓	
TI	Engine API Methods	✓	✓	✓
TI	Types of Traversal	✓	✓	
TI	External Implementations	✓	✓	

corresponding Neo4j graphs. We can observe that the behavior of Neo4j can be completely different for graphs where density varies. On one hand, both loading time and memory consumption grow as density for graphs DSJC.1, DSJC.5 and DSJC.9 increases. However, for the rest of the graphs, these values remain almost the same. This suggests that not only the density of a graph impacts on the cost of creating the Neo4j graph representation. The combination of density, number of nodes, edges and labels also seems to affect the behavior of this engine, and different configurations need to be set up during its evaluation.

Table 2. Benchmark of Graphs. Density is defined as $\frac{\#Arcs}{\#Vertices*(\#Vertices-1)}$

Graph	# Vertices	# Arcs	# Labels	Density	File size
DSJC.1	1,000	99,258	1	**0.10**	1.1M
DSJC.5	1,000	499,652	1	**0.50**	5.2M
DSJC.9	1,000	898,898	1	**0.90**	9.3M
Fixed-number-arcs-0.1	10,000	10,000,000	10,000	**0.10**	140M
Fixed-number-arcs-0.5	4,472	10,000,000	10,000	**0.50**	138M
Fixed-number-arcs-0.9	3333	10,000,000	10,000	**0.90**	136M
USA-road-d.NY	264,346	730,100	7,970	**1.04E-5**	13M
USA-road-d.FLA	1,070,376	2,687,902	22,704	**2.35E-6**	48M
Berlin10M	2,743,235	9,709,119	40	**2.58E-6**	2.1G

Transactions need to be carefully configured during the creation of the corresponding graph representations. To explain, graph database engines provide the possibility to configure the size of a transaction, i.e., when *checkpoints* will be executed. Suppose the RDF dataset of Berlin 10M (Table 2) was loaded in Neo4j, and three different transactional models were configured: *i*) one transaction per the whole graph creation process, *ii*) one transaction per 100,000 edges inserted

in the graph, and *iii*) no transactions. The experiment was run on a Sun Fire X4100 M2 machine with two AMD Opteron 2218 processors with 16GB RAM, running a 64-bit Linux CentOS 5.5. When the transactional model '*i*' was followed, the loading process ran out of memory after 10 hours, while the second model allowed to load the whole graph in 7 hours. Finally, when no transactions were considered, Neo4j could upload this graph in 2.1 hours and memory consumption was 7.35GB. This suggests that properly setting up the transactional model is extremely important during the evaluation of a given engine.

Table 3. Neo4j performance for graphs of different density. Dependent variables for the task of Graph Creation: Execution Time (msecs.); Main-Memory (MB); Secondary-Memory (MB). Java methods `System.currentTimeMillis()` and `Runtime.getRuntime()` were used to measured time and main-memory, respectively.

Graph	Execution Time (msecs.)	Main-Memory (MB)	Secondary-Memory (MB)
DSJC.1	12,295	17.27	3.7
DSJC.5	15,719	80.41	16.6
DSJC.9	21,460	142.60	29.5
Fixed-number-arcs-0.1	143,610	1,713.83	651.6
Fixed-number-arcs-0.5	131,621	1,711.09	324.3
Fixed-number-arcs-0.9	130,901	1,709.35	323.9

Finally, we consider the tasks of graph summarization, densest subgraph and shortest paths for DSJC.1, DSJC.5 and DSJC.9, and the engines Neo4j, DEX and HypergraphDB (Table 4). We can observe that tasks as shortest paths and densest subgraph do not seem to be as affected as graph summarization by the characteristics of these three graphs. To explain, both shortest paths and densest subgraph require to iteratively traverse the graphs until the paths with the shortest length are identified. Because in these three graphs the number of nodes is the same, and even when the number of edges increases, the maximal length of the paths does not change considerably. Thus, the time consumed by these two tasks is similar for these three graphs. Contrary, graph summarization requires to traverse all the edges to construct the compact representation of a graph and the corresponding corrections; therefore, graph summarization may be considerably affected when the number of edges increases.

5 Experimental Results

Considering the characteristics previously described, we configured several benchmarks to study the performance of DEX (version 4.8 very large databases), Neo4j (version 1.9), HypergraphDB (version 1.2) and RDF-3x (version 0.3.7). We evaluate graph creation time and memory consumption for a variety of graphs as well as execution time for diverse graph tasks. Indices on unique attributes are

Table 4. Graphs of different density for Graph Summarization Densest Subgraph and Shortest Paths. Timeout 3.5 hours.

	Graphs	Graph Summarization (msecs)	Densest Subgraph (msecs.)	Shortest Path(msecs.)
Neo4j	DSJC.1	**7,573**	6,511	6,774
	DSJC.5	**1,134,155**	8,901	9,466
	DSJC.9	**TimeOut**	14,976	14,436
DEX	DSJC.1	**6,810**	4,786	4,368
	DSJC.5	**1,031,753**	5,898	5,261
	DSJC.9	**12,576,203**	6,174	4,865
HGDB	DSJC.1	**10,974**	15,379	18,226
	DSJC.5	**1,047,470**	85,470	84,017
	DSJC.9	**TimeOut**	159,300	155,730

created for Neo4j, DEX and HypergraphDB; additionally, DEX uses indices on edges. Indices are loaded during the creation phase.

Graphs: DSJC.1, DSJC.5, DSJC.9, USA-road-d.NY, USA-road-d.FLA and Berlin10M are studied (Table 2). USA-road-d.NY and USA-road-d.FLA correspond to the New York City and Florida State road networks; these graphs are part of the 9th DIMACS Implementation Challenge - Shortest Paths[6].

Graph-based Tasks: we evaluate *creation*, *adjacency*, *reachability*, *pattern matching*, *densest subgraph*, and *graph summarization*.

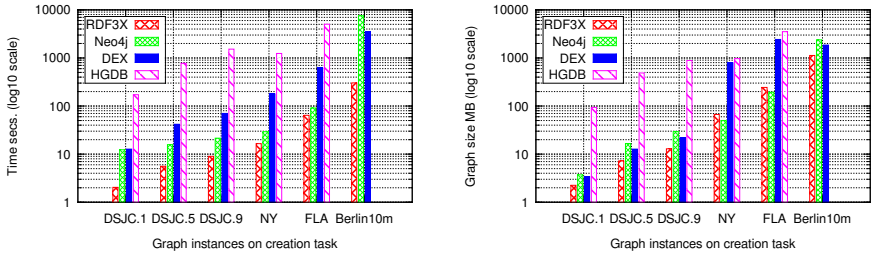
Evaluation Metrics: we report on runtime performance, which is measured by using the *real time* produced by the *time* command of the Linux operation system. Runtime represents the elapsed time between the submission of a task and the output of the answer. Further, we present the size in MB of the internal representation of a graph. Neo4j, DEX and HypergraphDB received graphs in the `sif` format[7], while graphs were passed to RDF-3x as sets of N-triples. Experiments were run on a Sun Fire X4100 M2 machine with two AMD Opteron 2218 processors with 16GB RAM, running a 64-bit Linux CentOS 5.5. All tests were executed in cold cache, i.e., we cleared the cache before running each task by performing the command `sh -c "sync ; echo 3 > /proc/sys/vm/drop_caches"`. Additionally, the machine was dedicated exclusively to run the experiments. Details can be found at `http://graphium.ldc.usb.ve`.

5.1 Performance During The Graph Creation Task

Figures 1(a) and (b) report on the creation time (secs and logarithm scale) and secondary-memory (MB and logarithmic scale) required to store DSJC.1, DSJC.5, DSJC.9, USA-road-d.NY, USA-road-d.FLA and Berlin10M. HypergraphDB timed out creating Belin10M after 24 hours, and required more time for creating the other five graphs than the rest of the engines. Although Neo4j and RDF-3x were competitive during the creation of USA-road-d.NY and USA-road-d.FLA, in general, we could say that RDF-3x is faster than the other engines in this task. This may

[6] http://www.dis.uniroma1.it/challenge9/download.shtml

[7] `http://wiki.cytoscape.org/Cytoscape_User_Manual/Network_Formats`



(a) Creation Time log-scale secs. (b) Secondary Memory log-scale MB.

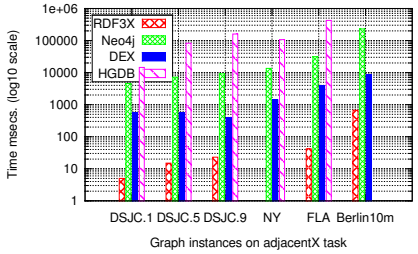
Fig. 1. Comparison of Neo4j, DEX, HypergraphDB and RDF-3x for graph creation of DSJC.1, DSJC.5, DSJC.9, USA-road-d.NY, USA-road-d.FLA and Berlin10M. No transactions were considered. Non-shown bars indicate that the corresponding engine timed out after 24 hours, i.e., HypergraphDB timed out creating Berlin10M after 24 hours.

be the result of exploiting in-memory caching techniques that allow RDF-3x to maintain hash tables to store the internal representation of an RDF graph, e.g., mappings between strings/URIs in RDF data to unsigned integer IDs, a single *Triple* table of these IDs, and the highly compressed indices. Additionally, the `sif` format used to represent the graphs, the size of the labels, and URIs could impact creation time of existing graph databases. We can also observe that the amount of secondary memory of the generated graphs grows as the size of original graphs increases. As previously discussed, several parameters impact on the size of the structure generated by each engine, e.g., density, number of nodes, edges and labels. However, it is important to highlight that RDF-3x graphs are smaller than the rest of the graphs generated by the other engines. This may be because RDF-3x makes use of LZ77 compression in conjunction with byte-wise Gamma encoding for IDs and gap compression [14].

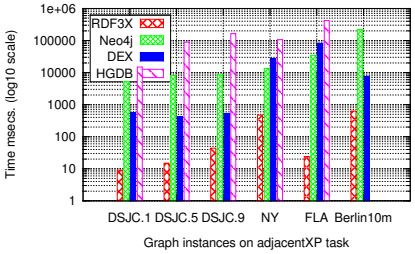
5.2 Performance on Core Graph-Based Tasks

Figures 2(a), (b) and (c) report on the execution time of the adjacency tasks; time is presented in msecs. and logarithmic scale. We recall that these tasks are implemented in the graph engines using their APIs while in the case of RDF-3x, we express them as SPARQL queries. Because RDF-3x implements specialized structures and indices for speeding up SPARQL queries, we can observe that it overcomes the rest of the engines by up to three orders of magnitude. Nevertheless, if the rest of the graph engines are compared, Neo4j and DEX are competitive, while HypergraphDB always consumes at least one order of magnitude more time than the rest of the engines. Finally, Figure 2(d) presents average execution time of the k-hops task, for $k \in \{2, 3, 4\}$; execution time is presented in secs, and logarithmic scale. Similarly, k-hops tasks are expressed in RDF-3x as SPARQL queries. On the other hand, in the rest of the graph engines, k-hops are implemented by using methods provided by their APIs for this task; we name this implementation as internal k-hops. Additionally, we study the effects of im-

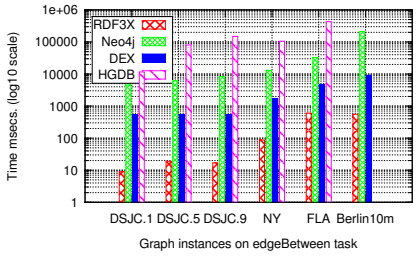
plementing this task by following a BFS strategy and relying on data structures to maintain the neighbors at a distance of m and $m+1$ in main-memory. Basic APIs functions are used to retrieve the adjacent nodes of a given node. This implementation is named external k-hops. First, we can observe that RDF-3x is impacted when graphs are dense, e.g., RDF-3x timed out computing *4*-hops in DSJC.5 and DSJC.9 after 60 minutes; nevertheless, RDF-3x can exploit its internal structures and indices in sparse graphs, e.g., USA-road-d.NY and USA-road-d.FLA. To explain, dense graphs with a fixed number of nodes, have a large number of edges that impact on the size of the k-hops. In consequence, this also affects the size of the intermediate results generated during the execution of the corresponding SPARQL queries; thus, RDF-3x requires more time than the rest of the engines to execute these queries against dense graphs. Contrary, RDF-3x exhibits a better performance on sparse graphs; in fact, RDF-3x overcomes the rest of the engines in both USA-road-d.NY and USA-road-d.FLA. Further, DEX exhibits a better performance in the majority of the graphs than the rest graph engines, and its internal and external implementations are competitive.



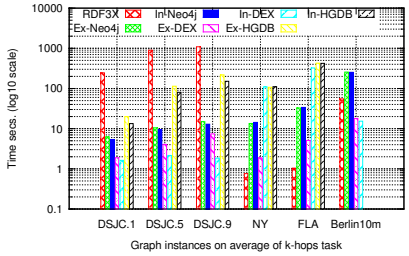
(a) AdjacentX task runtime, msecs. log-scale.



(b) AdjacentXP task runtime, msecs. log-scale.



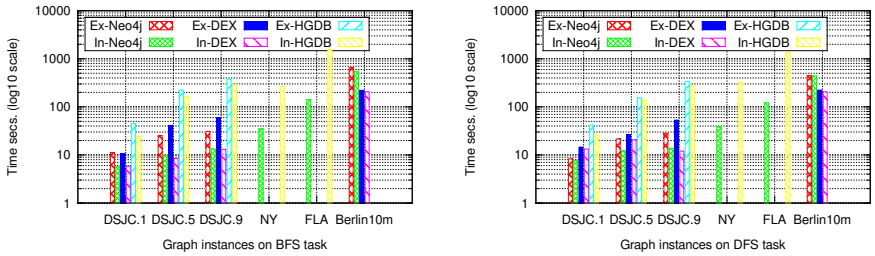
(c) EdgeBetween task runtime, msecs. log-scale.



(d) Average k-hops, secs. log-scale.

Fig. 2. Performance of Adjacency and Average k-hops (average execution time of 2-hops, 3-hops, and 4-hops). Internal k-hops are implemented using APIs (In-DEX, In-Neo4j, In-HGDB); external k-hops follow BFS traversal strategy (Ex-Neo4j, Ex-DEX, Ex-HGDB). Non-shown bars indicate that the corresponding engine timed out after 1 hour.

Finally, we study internal and external implementations of BFS and DFS search strategies. Figures 3(a) and (b) report on the execution time of these implementations when the studied graph engines are run against the five graphs. As expected, internal implementations of both BFS and DFS outperform the external implementations in all the engines. Nevertheless, in general, we can say that Neo4j exhibits a better performance than DEX and HypergraphDB. This may be caused by internal representation of Neo4j graphs and main-memory structures maintained by Neo4j to keep track of the nodes visited during the search.



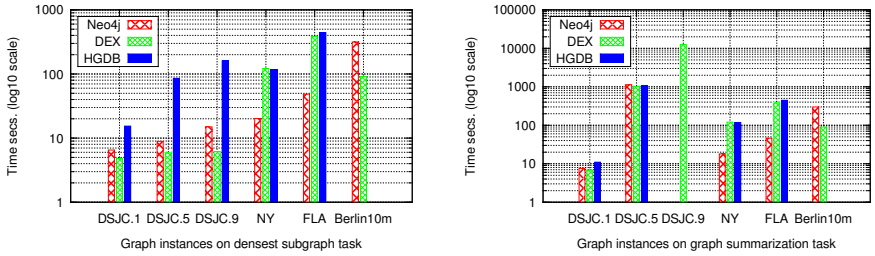
(a) Breadth-First Search task runtime, secs. log-scale.

(b) Depth-First Search task runtime, secs. log-scale.

Fig. 3. Comparison of Neo4j, DEX and HypergraphDB for Internal and External BFS and DFS. Internal tasks are implemented using APIs (In-DEX, In-Neo4j and In-HGDB); external algorithms rely on adjacency functions of the APIs (Ex-DEX, Ex-Neo4j and Ex-HGDB). Execution Time secs. log-scale. Non-shown bars indicate that the corresponding engine timed out after 1 hour.

5.3 Performance on Graph Summarization and Densest Subgraph

We further evaluate the impact on the studied graph database engines of the mining tasks of graph summarization and densest subgraph (Figure 4). It is important to highlight that to best of our knowledge, this is the first evaluation of these tasks on Neo4j, DEX, and HypergraphDB. Graph density, size and number of labels impact on the performance of both graph summarization and densest subgraph in all the engines. Nevertheless, graph summarization seems to be more affected by the graph density and the number of labels than for size of graph, i.e., execution time increases as the density of the DSJC.X family increases, as well as the number of labels grows in USA-road-d.NY and USA-road-d.FLA. Contrary, densest subgraph is more impacted by the size of the graphs. Additionally, we can observe that DEX overcomes the rest of the engines when the graphs are dense, while Neo4j has better performance in sparse graphs even if they have a large number of labels. To conclude, performance of these engines on both mining tasks is highly sensitive to graph characteristics.



(a) Densest Subgraph task runtime. (b) Graph Summarization task runtime.

Fig. 4. Comparison of Neo4j, DEX, HypergraphDB on Mining Tasks. Execution Time in secs, log-scale. Non-shown bars indicate that the corresponding engine timed out after 3.5 hours.

6 Conclusion and Future Work

We evaluated RDF-3x, Neo4j, DEX and HypergraphDB. First, our experimental study suggests that the considered independent variables highly impact on the performance of all these engines. Second, none of the engines outperforms the rest in all graph-based tasks. Nevertheless, we observe that RDF-3x exhibits a better performance than the rest of the engines during the execution of graph creation and pattern matching tasks; but, it is highly sensitive to graph density. HypergraphDB poorly performs in all the tasks of the study. Finally, Neo4j and DEX are quite competitive, and in some cases, one complements the other. Particularly, DEX outperforms Neo4j and HypergraphDB in the adjacency tasks and k-hops; also, DEX performs quite well in mining tasks if graphs are dense. On the other hand, Neo4j has a better performance in BFS and DFS traversals, as well as in mining tasks against sparse large graphs with high number of labels. Based on these results, we conclude that benchmarks that consider these independent variables and graph-based tasks need to be developed to uncover properties and limitations of these engines. In the future we plan to design and implement benchmarks able to measure performance of graph database and RDF engines. Additionally, to better understand cons and pros of existing engines, the analysis of graph structures and core methods is part of our future plans.

References

1. C. C. Aggarwal and H. Wang. Graph data management and mining: A survey of algorithms and applications. In *Managing and Mining Graph Data*, pages 13–68. 2010.
2. R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. In *GRADES 2013*, 2013.
3. C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
4. M. Dayarathna and T. Suzumura. Xgdbench: A benchmarking platform for graph stores in exascale clouds. In *CloudCom*, pages 363–370, 2012.

5. D. Dominguez-Sal, N. Martínez-Bazan, V. Muntés-Mulero, P. Baleta, and J.-L. Larriba-Pey. A discussion on the design of graph database benchmarks. In *TPCTC*, 2010.
6. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
7. J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.
8. B. Iordanov. Hypergraphdb: A generalized graph database. In *WAIM Workshops*, pages 25–36, 2010.
9. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Operations research*, 39(3):378–406, 1991.
10. T. Lampo, , M.-E. Vidal, J. Danilow, and E. Ruckhaus. To cache or not to cache: The effects of warming cache in complex sparql queries. In *ODBASE 2011*, pages 111–111, 2011.
11. N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey. Dex: high-performance exploration on large graphs for information retrieval. In *CIKM*, pages 573–582, 2007.
12. R. O. Nambiar, M. Poess, A. Masland, H. R. Taheri, M. Emmerton, F. Carman, and M. Majdalany. Tpc benchmark roadmap 2012. In *TPCTC*, pages 1–20, 2012.
13. S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *ACM SIGMOD*, pages 419–432. ACM, 2008.
14. T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1):256–263, 2010.
15. I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, 2013.
16. B. Saha, A. Hoch, S. Khuller, L. Raschid, and X.-N. Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *RECOMB*, pages 456–472, 2010.
17. M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. In *ISWC*, pages 82–97, 2008.
18. A. Thor, P. Anderson, L. Raschid, S. Navlakha, B. Saha, S. Khuller, and X.-N. Zhang. Link prediction for annotation graphs using graph summarization. In *ISWC*, pages 714–729, 2011.