Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark*

D. Dominguez-Sal P. Urbón-Bayes A. Giménez-Vañó
S. Gómez-Villamor N. Martínez-Bazán
J. L. Larriba-Pey
{ddomings,purbon,agimenez,sgomez,nmartine,larri}@ac.upc.edu
Universitat Politecnica de Catalunya, DAMA-UPC

Abstract

The analysis of the relationship among data entities has lead to model them as graphs. Since the size of the datasets has significantly grown in the recent years, it has become necessary to implement efficient graph databases that can load and manage these huge datasets.

In this paper, we evaluate the performance of four of the most scalable native graph database projects (Neo4j, Jena, HypergraphDB and DEX). We implement the full HPC Scalable Graph Analysis Benchmark, and we test the performance of each database for different typical graph operations and graph sizes, showing that in their current development status, DEX and Neo4j are the most efficient graph databases.

1 Introduction

Relational database models are providing the storage support for many applications. Relational database systems store biological datasets, economic

*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-1087 respectively.

transactions, provide storage for dynamic websites, etc. Although the relational model has proven efficient and scales well for large datasets in table form, it is not adequate for applications that deeply analyze relationships among entities. For example, the computation of a shortest path is not supported by a standard SQL query, and needs a stored procedure with a large number of joins, which are typically expensive to be computed.

On the other hand, the database community has become aware of this need for data storage applications for certain environments that do not require SQL and relational storage. In the recent literature, we find many projects that set alternative constraints to those imposed by the relational model in order to provide more natural query interfaces and improve the system performance. Some examples of this trend are: documental databases such as Lucene [2], which is able to fully index large document collections and support queries that rank the documents according to information retrieval measures; key-value data storages, such as BerkeleyDB [5], which stores the data collections as pairs that map a given key to an object; or even tabular-like representations without SQL support, such as Bigtable [8], which stores data as three dimensional tables that are indexed by two strings plus a timestamp that allows storing temporal sequences.

One type of data representation that is growing in popularity is graph databases (GDB) because many datasets are naturally modeled as a network of relationships: web data, authorship relationships, biological food chains, social networks, protein interaction, etc. Besides, the analysis of these datasets is guided by graph operations such as finding neighborhood of a node, graph traversals, finding minimum paths or community detection. Therefore, the storage and query execution in GDBs become the natural choice to store and develop queries on graphs.

Given that there are several implementations of GDBs, in this paper we will measure the performance of the most popular GDB alternatives. We analyze four different GDB libraries: Neo4j, HypergraphDB, Jena and DEX. We omitted InfoGrid, which is also a popular graph analysis tool, from this comparison because InfoGrid does not support attributes on edges (which are required by the HPC-SGAB) and the recommended workaround is to create additional intermediate nodes as associative entities, which store the weight [10]. This requires to implement specialized implementations for InfoGrid, that would differ from the shared algorithm patterns implemented for the rest of GDBs.

For the evaluation, we apply the recently developed HPC Scalable Graph

Analysis Benchmark [3] (HPC-SGAB). This benchmark was designed by researchers from academia, as well as members of several industrial companies to capture the most representative graph operations such as graph loading, short navigations or full graph traversals. It is out of the scope of this paper to present novel techniques to solve more efficiently the HPC-SGAB kernels: our main focus is to take advantage of the benchmark to evaluate the performance of the most representative GDBs available.

To our knowledge, the present work is the first full implementation of the HPC-SGAB on several GDB engines, running in the same hardware setup. Furthermore, we discuss several aspects from the HPC-SGAB, that we think that should be considered for future refinement of the benchmark. Our results show that some of the current GDBs can handle millions nodes efficiently, proving that DEX and Neo4j are the most efficient current GDB implementations.

The paper is organized as follows: Section 2 presents the four GDBs under study and summarizes its main features, Section 3 describes the HPC-SGAB, Section 4 reports and discusses the experimental results, Section 5 describes our experience implementing the benchmark, and finally, Section 6 concludes the paper.

2 Graph Database Libraries

2.1 Neo4j

Neo4j is a GDB designed for network oriented data, either in tree or general graph form. Neo4j does not rely on a relational layout of the data, but a network model storage that natively stores nodes, relationships and attributes [14]. Although the wiki provides detailed information about configuring Neo4j [16], to our knowledge, there is no research publication or technical report with an extensive description of the internals of Neo4j.

Neo4j is one of the most popular alternatives of GDBs due to its dual free software/commercial license model (AGPL license). Neo4j is fully written in Java and can be deployed on multiple systems. It supports transactions and fulfills the ACID consistence properties [14].

2.2 HypergraphDB

Hypergraph is a GDB, which was designed for artificial intelligence and semantic web projects [9]. As a consequence of the the requirements from these environments, HypergraphDB stores not only graphs but also hypergraph structures. A hypergraph is a mathematical generalization of the concept of a graph, in which the edges are substituted by hyperdges. The difference between edges and hyperedges is the number of nodes that they connect: a regular edge connects two nodes of a graph, but a hyperedge connects an arbitrary set of nodes. Given that the tests in the benchmark only compute operations on regular graphs, we will not take advantage of this feature from HypergraphDB.

HypergraphDB stores all the graph information in the form of key-value pairs. Each object of the graph, either nodes or edges, is identified with a single key that is called atom. Each atom is related to a set of atoms, which can contain zero or any number of elements. These relationships create the topological structure of the graph. HypergraphDB also supports node and edge typing. The types are also implemented as atoms that contain all the elements of a particular type [9].

The key-value structures are stored on an external library, BerkeleyDB [5], which does not support relational queries but provides specialized structures for this type of storage. BerkeleyDB offers three access methods to data: hashes, B-trees and Recno [17]. The first is implemented as a linear hash, which in case of multiple collisions performs multiple hashes to redistribute the colliding keys. The B-trees store the data in the leaves of a balanced tree. Recno assigns a logical record identifier to each pair, which is indexed for direct access.

2.3 Jena (RDF)

The Resource Description Framework (RDF) is a standard for describing relationships among entities. A relationship expressed in RDF is expressed as a triplet: a subject, a predicate and an object. The interpretation is that the subject applies a predicate on the object. Therefore, a RDF description can be viewed as a graph where the subject and the object are the vertices, and the predicate corresponds to the edge that relates them. There are several RDF based graph implementations: Jena [11], Sesame [18], AllegroGraph [1], etc. In our tests, we selected Jena because it is one of the most widely used.

Jena can be run on two storage backends, which store the RDF triplets: SDB and TDB [11]. In this paper, we limit our tests to the TDB backend because it provides better performance according to the Jena documentation. TDB stores the graph as a set of tables and indexes: (a) the *node table* and (b) the *triple indexes*[1].

The node table stores a map of the nodes in the graph to the node identifiers (which are a MD5 hash), and viceversa. The storage is a sequential access file for the identifier-to-node mapping, and a B-tree for the node-to-identifier mapping.

The triple indexes store the RDF tuples, that describe the structure of the graph. Jena does not implement a table with the three attributes of the RDF tuple, but three separate indexes. Each of these three indexes takes as the sorting key, each one of the three attributes of an RDF structure. Nevertheless, the value stored in the index is the full tuple for all the three indexes, in other words an access to any of the three indexes is able to retrieve a RDF relationship.

2.4 DEX

DEX is an efficient GDB implementation based on bitmap representations of the entities. All the nodes and edges are encoded as collections of objects, each of which has a unique oid that is a logical identifier [13].

DEX implements two main types of structures: bitmaps and maps. DEX converts a logical adjacency matrix into multiple small indexes to improve the management of out-of-core workloads, with the use of efficient I/O and cache policies. DEX encodes the adjacency list of each node in a bitmap, which for the adjacent nodes has the corresponding bit set. Given that bitmaps of graphs are typically sparse, the bitmaps are compressed, and hence are more compact than traditional adjacency matrices.

Additionally, DEX is able to support attributes for either vertices and edges, which are stored in a B-tree index. DEX implements two types of maps, which are traversed depending on the query: ones that map from the oid to the attribute value, and others from the attribute value to the list of oids that have this value. This schema fully indexes the whole contents of the GDB.

[1]An additional small table, called prefix table, is also used to export the results to other formats such as XML. This table is not used in a regular query execution [11].

In a nutshell, this implementation model based on multiple indexes favors the caching of significant parts of the data with a small memory usage, reverting in a better efficient storage and query performance [13].

3 Benchmark description

In this paper, we will implement the queries of the HPC Scalable Graph Analysis Benchmark v1.0 for the previously described GDBs. This benchmark was designed from a collaboration between researchers from universities and engineers from industrial partners that analyze different aspects of the performance of a GDB. In the rest of this section, we summarize this benchmark.

3.1 Data generation

In this benchmark, we test the performance of operations on directed and weighted graphs. The dataset is generated using the R-MAT algorithm, which is able to build graphs of any particular size and edge density [7]. Furthermore, the graphs that R-MAT builds follow typical real graph distributions (power law distributions), such as those found in real life situations [7].

The general idea of R-MAT is the following. Suppose that we represent the graph as an adjacency matrix, in which each row and column is a vertex and the coordinates into the matrix indicate if the two vertices are connected. R-MAT divides the matrix in four equal squared sections, which have an associated probability in HPC-SGAB: a, b, c and d. Then, R-MAT picks one of the four sections according to these probabilities. Once the section is selected, R-MAT iterates recursively this process dividing again the selected section in four equal parts. The process finishes when the size of the selection is equal to one position, and then, R-MAT selects this position as the next inserted edge. The process can be iterated arbitrarily to create graphs of the desired density.

In general, the parameters must not be symmetric in order to have power law distributions in the graph (which are the most common distribution in real huge graph datasets [12]). In our benchmark, we take the recommended values by the HPC-SGAB for all the graphs tested: $a = 0.55$, $b = 0.1$, $c = 0.1$ and $d = 0.25$. The number of nodes and edges of the graph is indicated by a parameter called *scale*. The number of nodes is $N = 2^{scale}$,

and the corresponding number of edges is $M = 8 \cdot N$. Finally, the weight of each edge is a positive integer value following a uniform distribution with maximum value 2^{scale}. In the experimental section, we fix the probabilities to the stated parameters and study the performance of the different GDBs for different scales.

The R-MAT generation time is not computed for the benchmark, since it is not part of the GDB performance. Therefore, R-MAT generates a file of tuples with the form <StartVertex, EndVertex, Weight>, that will be loaded into the native GDB format.

3.2 Kernel description

This benchmark is composed of four kernels, where a kernel is an operation whose performance is tested. The first kernel loads the data in the GDB, and the rest of kernels will use this image to compute the queries. The kernels are the following:

- **Kernel 1**: The first kernel measures the edge and node insertion performance of a GDB. This kernel reads the database file, in the format described in Section 3.1, and loads it. The loading includes the creation of all the indexes needed to speedup the computation of the following kernels. Note that the R-MAT file generation process is not included in this timing. In the following queries to the GDB, we use the graph loaded by this kernel, which cannot be modified during the following kernels.

- **Kernel 2**: This kernel measures the time needed to find a set of edges that meet a condition, in this case it finds all the edges with the largest weight. The algorithm output is the list of edges and nodes that connect them. The list of nodes is stored in order to initialize Kernel 3.

- **Kernel 3**: This kernel measures the time spent to build subgraphs in the neighborhood of a node. It computes a k-hops operation, which we implemented using a breadth first search traversal, starting from the edges produced by Kernel 2. We set 2 as the number of additional hops that the algorithm traverses from the tail node of each edge, which adds up to total length of three including the initial edge. Since the graphs follow power law distributions, this operation may access a significant number of nodes.

- **Kernel 4**: This kernel estimates the traversal performance of the GDB over the whole graph. This value is estimated as the Traversed Edges Per Second (TEPS). Kernel 4 estimates the TEPS using a complex graph operation that calculates the Betweness Centrality (BC) of the graph. The BC of a graph gives a score to each node that indicates how far is a node from the center of the graph. It assigns low scores to the nodes that are in the external parts of the graph, and high scores to the nodes that are in the internal parts of the graphs. More formally, the BC of one vertex v accounts for the ratio of shortest paths (σ_{st}) between any pair s and t of nodes that pass through v ($\sigma_{st}(v)$) with respect to those that do not:

  $$BC(v) = \sum_{s \neq v \neq t \in G} \frac{\sigma_{st}(v)}{\sigma_{st}}.$$

  Note that when we refer to the BC of a node we are not considering the weight of the edges. However, this kernel does not compute the BC on the whole graph but on the induced graph that removes all edges with a weight multiple of eight and keeps all the nodes, as indicated by the benchmark. We implement this restriction implementing a BC algorithm that skips the edges that are multiples of eight while exploring the graph, instead of computing this subgraph during Kernel 1.

  We implement the BC using the Bader's algorithm [4] for a single core, which provides an approximated solution based on the Brande's exact algorithm [6]. The complexity of Bader's strategy is $O(k \cdot M)$, where k is the number of samples, and is the recommended algorithm by the HPC-SGAB. The Bader's algorithm calculates the centrality using a sample of multiple BFS traversals, starting from different nodes. In our tests, we pick 8 (in other words, scale 3) as the number of samples for computing the BC.

4 Experiments

4.1 Experimental setup

For each of the databases, we used the latest available versions: Neo4j v.1.0, Hypergraph v.1.0, Jena v.2.6.2 (with TDB 0.8.4), and Dex v.3.0. All the

benchmarks were implemented using the Java interface of the GDBs, which is the recommended programming API for all of them. In order to minimize differences in the performance because of the algorithm implementation, we followed the recommended algorithms described in the HPC-SAGB description, and all the implementations were written following the same generic implementation.[2]

In order to configure each database, we used the default configuration plus the recommendations found in the documentation of the websites. For Neo4j, we implement Kernel 1 with the batch inserter (which disables transactions), configured with the recommended values: $N \cdot 9$ bytes for the edge store, $N \cdot 33$ bytes for the node store [15]. For HyperGraphDB, we disabled the transactional capabilities, which were not required in the benchmark and are an additional overhead.

The kernels are executed in the order indicated by the HPC-SGAB. In Section 4.2, we report the results after a warm up stage, that computes kernels 2 and 3 once. This setup resembles a database system that is computing queries during long timespans. Nevertheless, we also discuss the results without a warm up stage in Section 5. We halt the execution of kernels that take more than 24 hours to compute.

We execute our experiments in a computer equipped with two Quad Core Intel Xeon E5410 at 2,33 GHz, 11 GB of RAM and a LFF 2.25Tb disk. We use the default parameterization of the Java Virtual Machine for all the kernels except for the largest ones: 2GB for scale 20, and 10GB for scale 22 and up.

4.2 Analysis of results

We executed the HPC-SGAB with four different scales for the already described databases: 10, 15 and 20. These setups correspond to databases with 1k, 32k and 1M nodes, which account for a total number of objects ranging from approximately 10k to more than 9.4M objects.

Tables 1, 2 and 3 summarize the results of the benchmark for the different kernels tested. We were not able to scale the comparison over larger scale factors because most GDBs had problems to load the full graph in a reasonable time with our current hardware, and hence we could not continue the

[2]The query implementations are publicly available in the following website: `http://trabal.ac.upc.edu/public/dama-upc-iwgd2010-hpca-sgab-source.zip`

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**0.316**	19.30	7.895	376.9
K2 Scan edges (s)	**0.001**	0.131	0.090	0.052
K3 2-hops (s)	**0.003**	0.006	0.245	0.015
K4 BC (s)	0.512	**0.393**	5.064	1.242
Db size (MB)	2.1	**0.6**	6.6	26.0

Table 1: Scale factor 10

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**7.44**	697	141	+24h
K2 Scan edges (s)	**0.001**	2.71	0.689	N/A
K3 2-hops (s)	**0.012**	0.026	0.443	N/A
K4 BC (s)	14.8	**8.24**	138	N/A
Db size (MB)	30	**17**	207	N/A

Table 2: Scale factor 15

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**317**	32094	4560	+24h
K2 Scan edges (s)	**0.005**	751	18.60	N/A
K3 2-hops (s)	0.033	**0.023**	0.458	N/A
K4 BC (s)	**617**	7027	59512	N/A
Db size (MB)	893	**539**	6656	N/A

Table 3: Scale factor 20.

comparison for even larger datasets. We could not load the scale 22 graph in less than 24 hours in the GDBs tested, except for DEX where it took 27.5 minutes. Furthermore, we were not able to load the scale 24 graph in less than three days in any of the databases, except for DEX, for which it took 28 hours.

We first observe that the scalability of all the databases is not equivalent. Although DEX and Neo4j were able to scale up to the 1M nodes dataset, Jena and HypergraphDB could not load the database in a comparable time. We were not able to load the graphs with 1M nodes in HypergraphDB in 24 hours, and although Jena was able to load the graph with 1M nodes faster than Neo4j, Jena did not scale to the largest scale.

Regarding Kernel 1, which measures the load time, we find that the fastest alternative is DEX that loads the dataset at least one order of magnitude

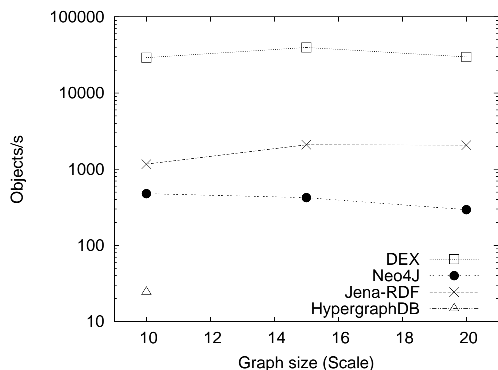

Figure 1: Objects loaded per sec. (Kernel 1)

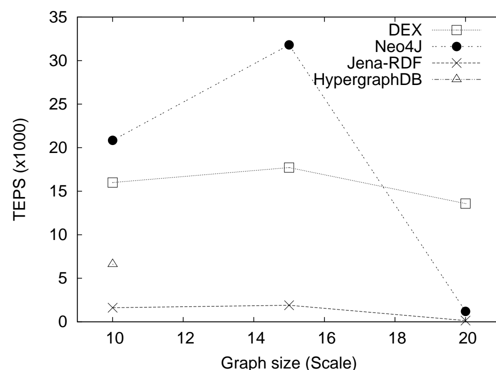

Figure 2: TEPS (Kernel 4)

faster than the rest of algorithms. The insertion rate for DEX scales well for the different benchmarks, adding roughly 30k objects per second as depicted in Figure 1. Jena inserts approximately 2k objects per second, and Neo4j is up to two orders of magnitude slower than DEX for adding data. On the other hand, the image generated by Neo4j is the most compact: it is 40% smaller than the following one, which is DEX, and 90% and 98% with respect to Jena and HypergraphDB.

Kernel 2, measures the time to find a subset of edges. For this task, DEX is the fastest algorithm again for all the tested sizes because it can take advantage of its indexes over edges [13]. It is a fast operation for all the GDBs, because it consists of a single iteration over the weights of all edges. However, for large graphs this operation is expensive for Neo4j because the API does not provide a direct access for iterating over all edges. In order to retrieve the edges, Neo4j iterates over all the nodes in the database, and explores the adjacent edges, which supposes an additional overhead to Kernel 2. This is particularly expensive when the graph is large because since it follows a power law distribution, some nodes have an elevated degree.

All the GDBs are able to compute the third kernel very fast. The operation accesses the local neighborhood of one node and we observe that Neo4j is the best for larger graphs for Kernel 3. On the other hand, DEX is the fastest for the small graph and is very close to Neo4j for the large graph. As we discuss in Section 5, this kernel is heavily influenced by the warm up of the database because of the very short execution time.

Finally, Kernel 4 measures the traversal performance of each algorithm over the whole graph. For the smallest graph size, Neo4j obtains the best performance, but for larger graphs DEX scales better. For small graphs, DEX and Neo4j are significantly faster than the other two GDBs: Neo4j is up to five times faster than HypergraphDB and one order of magnitude faster than Jena. For large graphs, there are big differences among the three databases. DEX obtains a speedup above 60 over Jena and a speedup above 11 over Neo4j.

Figure 2 depicts the discussed results for Kernel 4 as a function of the traversed edges per second, which is more adequate for comparing different database sizes. We observe that DEX and Jena scale with the size of the graph because the slope of the curve is not very pronounced. Nevertheless, DEX is able to traverse 10 times more edges than Jena, which makes DEX a better choice for traversing operations. On the other hand, Neo4J traverses more than 32k TEPS for the small graph but when the graph is large the performance decreases to less than 1.2k TEPS.

In summary, we found that DEX and Neo4j are the only databases that were able to load the largest graphs and compute the queries efficiently. HypergraphDB failed to load the scale 15 and 20 databases in a reasonable time, and in any case, it was not faster than DEX and Neo4j for any kernel. Jena scaled up to factor 20 but it was slower than Neo4j and DEX. We observed that DEX is the fastest for most kernels (9 out of 12 kernels), and in the three cases where Neo4j is the fastest, the difference is small. Furthermore, we detected that Neo4j did not scale as well as DEX for Kernels 1, 2 and 4, in which DEX is up to more than ten times faster than Neo4j.

5 Experience from the benchmark

In this section, we discuss some considerations that we have drawn after implementing the benchmark. We group them into the following points:

- **Imbalance in Kernels 2 and 3 costs**. We found that the computation cost of the different kernel operations is very different, specially for the largest graphs, where the computation of Kernel 4 consumes 10k more time than Kernels 2 and 3 for scale 20. Kernels 2 and 3 measure the performance of operations that are local in nature and whose cost does not grow at the same rate of the graph. For example,

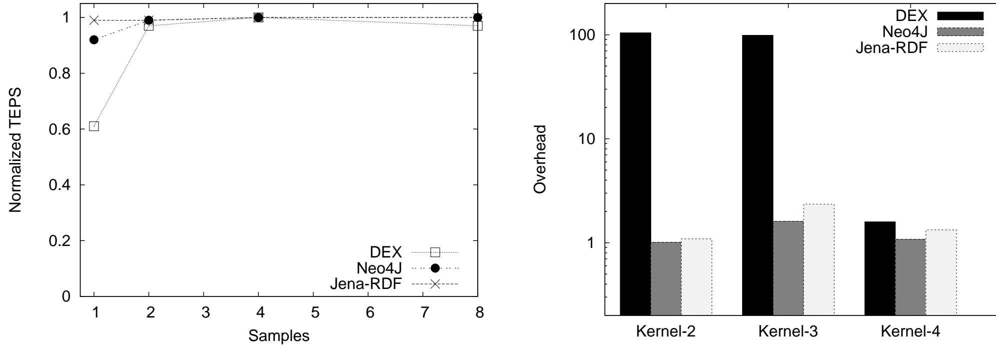

Figure 3: TEPS Normalized to the fastest value for a given GDB.

Figure 4: Warm up overhead (scale 20).

the number of objects grows 1000 times from scale 10 to 20, and thus Kernel 1 building costs are multiplied by this magnitude. But, the cost of computing Kernel 3 is only about ten times larger with the change of scale.

- **TEPS estimation**: On the other side of the spectrum, the BC function calculated in Kernel 4 is a very complex operation. The use of the exact BC is not affordable for large graphs because the number of BFS traversals is equal to the number of nodes. Therefore, the application of the Brandes approximated algorithm for Kernel 4 is necessary for large graphs.

  Furthermore, we found that the number of samples to estimate the TEPS for a particular GDB does not need many traversals. In Figure 3, we depict the average TEPS of the databases for an increasing number of samples of Kernel 4. In this plot, the TEPS is normalized to the TEPS of the fastest sample of the BC algorithm for each database. We observed that the first sample exhibits a warm up effect, such as for DEX and Neo4j, because they load for first time some of the data structures of the graphs. However, once the number of samples is above two the TEPS stabilizes for all the databases. Thus, as Figure 3 shows, Kernel 4 does not need an excessive number of iterations to estimate the TEPS obtained by a GDB, i.e. more than eight iterations would not be necessary to measure the traversal performance of the GDB.

- **Warm up**: An important topic that is not discussed in the benchmark description is the warm up process of the GDB. Typically, the implementation of good caching strategies improves the performance of an application because it does not read information from the secondary storage. In Figure 4, we plot the overhead produced by the first execution of one kernel compared to the following computation of the same kernel. The difference between having all data in memory or loading part from disk is more severe for kernels that measure simple operations that are computed very fast, such as the kernel 2 and 3 of the HPC-SGAB. For example, in DEX the first computation of Kernel 2 takes approximately 100 times more to be computed than the following executions. The reason is that the first execution loads from disk the necessary data pages, which takes half a second, and then computes the query in five milliseconds more. The following executions skip the cost to load from disk, and thus are significantly faster. This effect is clearly visible in DEX and Jena in Figure 4.

As a consequence of the previous issues we believe that some aspects can still be improved for future revisions of the benchmark. From our point of view, the time needed to compute all kernels should be harmonized and scale at a similar rate to the graph growth. For example, we suggest that Kernel 3 should perform the k-hops operation over a larger set of nodes as origin. Instead of only picking the set of edges with the largest weight, we suggest to pick a fixed percentage of edges, such as the set of the 1‰ largest edges.

Furthermore, we think that it is important to include a remark about the warm up of the GDB in the benchmark description. Since GDBs are complex and have their own bufferpool implementation, we think that the fairest option would be to enable a warm up process before the timed kernel executions. This is particularly important for graphs that do not fit in memory and where the bufferpool hit rate plays a significant role in the final system performance.

6 Conclusions

In this paper, we have tested the performance of four of the most popular graph data management applications with the recently developed HPC-SGAB. The results derived from the benchmark show that for small graphs

all four databases are capable to achieve a reasonable performance for most operations. However, in our hardware setup, only DEX and Neo4j were able to load the largest benchmark sizes.

DEX is the fastest database loading the graph data and performing the operations that scan all the edges of the graph, exhibiting an improvement over one order of magnitude than the second best graph database, Neo4j. For Kernel 3, we found that Neo4j is faster than DEX for the large dataset, and the reverse happens for the small dataset. Nevertheless, the difference between both databases is small for all the data sizes tested in Kernel 3. For Kernel 4, Neo4j is able traverse more than 32k TEPS, but they do not scale properly for graphs with a million nodes because the performance drops to 1,2k TEPS. On the other hand, DEX is able to scale better, traversing roughly 15k TEPS either for small and large graphs. All in all, Neo4j obtained a good throughput for some operations, but we found that it had scalability problems for some operations on large data volumes, specially in the full graph traversals. DEX achieved the best performance for most operations and close to Neo4j, in those where Neo4j was the fastest.

Finally, we discussed two aspects to take into account for improving future versions of HPC-SGAB kernels based on our experience implementing and testing it on real hardware: (a) harmonize the costs of Kernels 2, 3 and 4 with operations that scale similarly with the graph size in order to obtain a more balanced benchmark, (b) defining a warm up policy that is able to show the performance of the databases with respect to the secondary storage devices for huge graphs.

References

[1] AllegroGraph. AllegroGraph website. *http://www.franz.com/agraph/*, Last retrieved in May 2010.

[2] Apache Lucene. Lucene website. *http://lucene.apache.org/*, Sept. 2008.

[3] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koetser, E. Loh, K. Madduri, B. Mann, T. Meuse, and E. Robinson. HPC Scalable Graph Analysis Benchmark v1.0. *HPC Graph Analysis*, February 2009.

[4] D. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, pages 539–550, 2006.

[5] BerkeleyDB. BerkeleyDB website. *http://www.oracle.com/database/berkeley-db/index.html*, Last retrieved in March 2010.

[6] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

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

[8] F. Chang and J. Dean and S. Ghemawat et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[9] HypergraphDB. HypergraphDB website. *http://www.kobrix.com/hgdb.jsp*, Last retrieved in March 2010.

[10] Infogrid. Blog. *http://infogrid.org/blog/2010/03/operations-on-a-graph-database-part-4*, Last retrieved in March 2010.

[11] Jena-RDF. Jena documentation. *http://jena.sourceforge.net/documentation.html*, Last retrieved in March 2010.

[12] J. Leskovec, L. Lang, A. Dasgupta, and M. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008.

[13] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, et al . Dex: high-performance exploration on large graphs for information retrieval. In *CIKM*, pages 573–582, 2007.

[14] Neo4j. The neo database. *Available at http://dist.neo4j.org/neo-technology-introduction.pdf*, 2006.

[15] Neo4j. Batch Insert. *http://wiki.neo4j.org/content/Batch_Insert*, Last retrieved in March 2010.

[16] Neo4j. Neo4j wiki documentation. *http://wiki.neo4j.org/content/Main_Page*, Last retrieven in March 2010.

[17] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999.

[18] Sesame. Open RDF website. *http://www.openrdf.org/*, Last retrieved in May 2010.