Graph Analysis – Do We Have to Reinvent the Wheel?

Adam Welc
Oracle Labs
adam.welc@oracle.com

Raghavan Raman
Oracle Labs
raghavan.raman@oracle.com

Zhe Wu
Oracle
alan.wu@oracle.com

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Hassan Chafi
Oracle Labs
hassan.chafi@oracle.com

Jay Banerjee
Oracle
jayanta.banerjee@oracle.com

ABSTRACT

The problem of efficiently analyzing graphs of various shapes and sizes has been recently enjoying an increased level of attention both in the academia and in the industry. This trend prompted creation of specialized graph databases that have been rapidly gaining popularity of late. In this paper we argue that there exist alternatives to graph databases, providing competitive or superior performance, that do not require replacement of the entire existing storage infrastructure by the companies wishing to deploy them.

1. INTRODUCTION

Many important computational problems from different problem areas, such as circuit analysis, machine learning or biotechnology, can be expressed in the form of graph algorithms. Consequently, the search for convenient and efficient solutions for executing graph algorithms has been enjoying an increased level of attention both in the academia and in the industry. In particular, the graph database systems have been recently gaining a lot of popularity.

Graph databases offer similar functionality as the relational databases (e.g. reliable persistent storage, ACID [5] transactions), but instead of internally representing data as tables, they represent it directly in the form of a graph. The intention is to avoid the data representation mismatch when working with graph-shaped data (e.g. by avoiding translation to and from the relational table and column format), which should provide unbeatable performance when executing graph algorithms.

In this paper we argue that there exist viable alternatives to specialized graph databases, providing competitive or superior performance, that do not require replacement of the entire existing storage infrastructure by the companies wishing to deploy them. We will show that for certain important types of graphs, contrary to popular belief, implementations of graph algorithms in SQL can deliver performance matching or exceeding that of the dedicated graph databases. We will also demonstrate that even larger performance advantage over graph databases can be achieved using in-memory graph analysis engines that can utilize relational database as the storage solution.

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York, NY, USA.
Copyright 2013 ACM 978-1-4503-2188-4 ...$15.00.

One of the prime examples of the graph database technology which we use for a limited performance study presented in this paper is Neo4j [11], "The World's Leading Graph Database" [1] - this claim backed up by a multitude of customers [10] and descriptions of extremely favorable comparisons with solutions based on the relational database technology [12].

We certainly do not question some of the intuitively solid performance benefits of the graph databases. For example, with certain types of data encodings a single graph edge traversal may have to be implemented in SQL as a separate join operation [6], which indeed incurs significant overhead. In fact, this is confirmed by some our own performance evaluation results presented in this paper. It is also not our intention to criticize Neo4j specifically, as it is clearly a useful piece of technology appreciated by its multiple clients. Instead, our main goal is to demonstrate that graph databases are not the only viable solution for solving all graph-related problems. We simply use Neo4j as a representative example of the graph database technology, and contrast its performance with those of a SQL-based solution layered on top of a general-purpose relational database, and with a system utilizing the Green-Marl Domain Specific Language (DSL) for graph analysis [7] featuring custom-built DSL compiler and graph analysis runtime. Due to space limitations, in this work we focus on evaluating implementations of one important graph analysis algorithm - Dijkstra's shortest path [3].

The main contributions of the paper include:

1. We show that performance advantage claims of Neo4j over the SQL-based solutions do not always hold. In particular, we provide experimental evidence demonstrating that SQL-based implementation of Dijkstra's shortest path for graphs representing data from social networks provides at least competitive and in most cases superior performance to that of Neo4j's.

2. We demonstrate that while comprehensive graph storage and analysis solutions utilizing database technology are certainly useful for some use-cases, they cannot

[1]Wording taken verbatim from the first page of Neo4j's web site at http://www.neo4j.org/

match the performance of in-memory graph analysis engines in typical usage scenarios.

3. We discuss trade-offs between solutions for graph analysis utilizing database technology and the in-memory graph analysis engines, followed by a proposal for an integrated architecture aiming at best performance and usability.

2. SHORTEST PATH ALGORITHMS

The ability to efficiently compute shortest paths is not only an important graph problem by itself, required for example to find the smallest number of people connecting two members of a social network, but it is also a building block for other important graph problems, such as betweenness centrality where a relative importance of a given graph node is expressed as the number of shortest paths in the graph passing through that node. Consequently, shortest path is a suitable candidate for preliminarily comparing different platforms supporting graph analysis, and it is supported by Neo4j's public API "out of the box".

Many algorithms exist that can be used to find the shortest path between two nodes in a graph. In particular, Neo4j (only) supports "standard" (uni-directional) Dijkstra's shortest path algorithm, Dijkstra's bi-directional shortest path algorithm (all versions of Dijkstra's algorithm used in this paper are described in Section 2.1), and A$*$ shortest path algorithm. Even assuming that all graph analysis engines implement the same exact version of the algorithm, it is difficult to perform a fair direct comparison between different platforms as they may use different graph representations, utilize different auxiliary data structures, or even be implemented in different languages. As the goal of our performance study is to merely demonstrate that performance-wise there exist alternatives to graph databases, it is in our opinion reasonable to compare performance of the best shortest path algorithm available on each platform. Consequently, for SQL we evaluate bi-directional set-based version of Dijkstra's shortest path and for both Neo4j and Green-Marl we evaluate bi-directional non-set-based version of Dijkstra's shortest path.

2.1 Dijkstra's shortest path

In this section we describe various flavors of Dijkstra's shortest path algorithm.

2.1.1 Uni-directional Dijkstra

In the uni-directional version of Dijkstra's algorithm [3] all nodes maintain the shortest distance from the source node (computed so far) – initially 0 for the source node and infinity for all other nodes. Every node whose shortest path from the source node has been found is said to be *finalized*. The algorithm also maintains a *frontier*, which is a set of nodes that have been *touched* (i.e., shortest distance is not infinity) but not *finalized*. The algorithm begins with only the source node in the *frontier* and no *finalized* nodes.

First, the algorithm picks a node with the minimum distance from the *frontier*, which we refer to as the current node. The algorithm *finalizes* the current node because the shortest path from the source node to the current node has been found. It then *expands* the current node to its neighbors – it compares the neighbor's shortest distance with the sum of the current node's shortest distance and the cost of the edge leading from the current node to the neighbor node. If the neighbor's shortest distance is larger than the computed sum then the cost of the neighbor node is *relaxed*, that is replaced with the value of the computed sum, and the neighbor node is inserted into the *frontier* if it is not already in there. The algorithm repeats the above steps until either the destination node is *finalized* (path found) or the *frontier* is empty (path not found).

2.1.2 Bi-directional Dijkstra

The bi-directional version of Dijkstra's shortest path [9] attempts to improve over the uni-directional one by proceeding with the shortest path search both from the source node and from the destination node, picking one of the two directions to expand at every step of the algorithm. Note the shortest path search from the destination node is done on the graph where the direction of all edges are reversed. The bi-directional version of the algorithm is designed to prune the search space, that is limit the number of nodes that would otherwise have to be expanded by the uni-directional algorithm. The basic version of the algorithm terminates when a common node is finalized by both forward (from source) search and reverse (from destination) search.

One common optimization applied to the bi-directional version of Dijkstra's shortest path is the following. The algorithm maintains the *global minimum path cost* – the cost of the shortest path computed so far from the source node to the destination node, which is initially set to infinity. The global minimum path cost may get updated after both forward search and reverse search relax a common node. The algorithm terminates if both forward search and reverse search exhaust their search space without encountering a common node (no path found) or if the sum of minimum paths in both search directions gets larger than the global minimum path (path found – searching in either direction can only result in increasing the global path cost). This optimization is included in all implementations evaluated in this paper.

Multiple additional optimizations exist for the bi-directional version of Dijkstra's shortest path algorithm. These include the choice of direction to expand at every step, different termination conditions, constraints to avoid expansion to nodes that could never lead to shortest paths, etc. Please refer to [4, 9, 15] for the description of some of these optimizations.

2.1.3 Bi-directional set-based Dijkstra

In the non-set based version of Dijkstra's bi-directional shortest path algorithm, by analogy to the uni-directional variant, only one current node is selected from possibly multiple frontier nodes with the same shortest distance, as described in Section 2.1. The bi-directional set-based version of Dijkstra's shortest path [4] groups processing of all frontier nodes with the same shortest distance into a single operation. This strategy relies on the assumption that the cost of the operation on the set of nodes is significantly smaller than the sum of the individual nodes processing costs, and works particularly well in the case of SQL implementation as in this case both the set operation and individual node operation require a single SQL statement each.

3. GRAPH ANALYSIS SOLUTIONS

In this section we will present an overview of the three graph analysis solutions used in our performance case-study.

3.1 Green-Marl

Green-Marl [7] is a Domain Specific Language (DSL) designed specifically to facilitate easy expression of graph algorithms - the notions of graph, node and edge (along with node and edge properties) are explicit in the language as are some of the popular graph operations required for implementation of graph algorithms (e.g. node and edge iterations, breadth-first search, depth-first search, parallel value reductions etc.).

The open-source and publicly available[2] Green-Marl language infrastructure includes a multi-backend optimizing compiler capable of generating code targeting, among others, a shared memory runtime written in C/C++ (parallelism support is provided by the OpenMP [2] library). The runtime features in-memory graph analysis and, as such, does not support durability or database-like transactions, but allows access to both graph structure (nodes and edges) and graph properties. The nodes and edges of the graph are stored in the Compressed Sparse Row (CSR) representation, also used by other popular systems used for graph analysis, such as the SNAP library [1]. This representation consists of two contiguous arrays. One array stores id-s of all nodes in the graph. The other array, for each (source) node in the first array stores information about this node's outgoing edges in the form of (destination) node id-s. The edge cost properties required for shortest path computation are internally represented as an array of integer values where each value in the array is mapped to a single edge id.

The implementation of Dijkstra's shortest path in Green-Marl is a fairly straightforward transliteration of the original algorithm description. Due to space limitations we omit the detailed description – the source code of the bi-directional version of Dijkstra's shortest path used for our experiments is available in Green-Marl's public repository [3]. The Green-Marl code implementing the algorithm gets translated by the Green-Marl compiler to C/C++. Then, it is compiled with a standard C++ compiler and linked together with a simple driver program responsible for loading the graph from the file and for measuring the execution time.

3.2 Neo4j

Neo4j is an open-source NOSQL graph database which offers features similar to relational databases like reliability with full ACID transactions, durability, and scalability. As with all graph databases, the major attraction of Neo4j is that it has an intuitive graph model for data representation. This graph model seems to be a natural fit in the world of graph processing and analytics.

Neo4j stores data in the form of nodes and edges (also known as relationships) in a graph with properties attached to these nodes and edges. With this data model graph traversals are done intuitively by going from a node to its neighbors through direct links. This is unlike relational databases where a graph traversal may require several joins or look-ups on tables.

We use the implementation of Dijkstra's algorithm that comes with the Neo4j distribution for our evaluation. Note that Neo4j comes with two implementation of Dijkstra's algorithm, one is the original uni-directional version and the other is the bi-directional version. In this paper, we show the results of the bi-directional version since it consistently performs better than the uni-directional version at least by an order of magnitude.

We access Neo4j 's functionality through the Java API that comes with the Neo4j distribution. First, our Java program parses the input dataset in adjacency list format and creates a Neo4j database by performing batch insertion. The nodes and edges in the input dataset are added as nodes and relationships in the Neo4j database. The costs of the edges are added as properties of the corresponding relationships in the Neo4j database that is being created.

[2]https://github.com/stanford-ppl/Green-Marl
[3]https://github.com/stanford-ppl/Green-Marl/apps/src/bidir_dijkstra.gm

3.3 SQL

SQL is a standard-based, versatile and powerful language for querying structured data. Many existing real-world applications, including mission critical ones that need to run 24x7, are based on SQL and modern RDBMS, such as Oracle Database (which we use to evaluate SQL queries for the purpose of this paper). This makes integration of graph analysis with the existing applications much easier, particularly considering extensive tooling support that has been built around SQL and RDBMS, including Data Management UI, Business Intelligence, and Data Mining. Furthermore, the RDBMS technology has been in development for decades and all of its components, including transaction support, SQL execution engine, as well as memory, CPU and I/O subsystem management, are highly optimized. For all these reasons, we believe that SQL and RDBMS should be considered as a viable solution for some classes of graph analysis problems.

In our SQL-based approach, we model a directed weighted graph using a simple relational table with three columns: integer-typed source node ID column (`SID`), integer-typed destination node ID column (`DID`), and numeric weight column `W`. Two B-Tree indexes are created to allow easy lookups in both forward and backward directions. The first index is a multi-column index consisting of columns (`SID`, `DID`, `W`) and the second index consists of columns (`DID`, `SID`, `W`). The reason why the weight column `W` is included in both indexes is that these two multi-column indexes provide all necessary information for path calculations. In other words, there is no need to access the base graph table at query time.

Our implementation of bi-directional set-based version of Dijkstra's shortest path algorithm has been inspired and closely resembles implementation in the FEM framework developed by Gao et al. [9]. In particular, the core expansion and relaxation operations of the bi-directional, set-style Dijkstra algorithm are carried against an intermediate working table (`IWT`) which starts as an empty table and gets populated and updated as the search for the shortest path unfolds. The `IWT` consists of a node ID column, two columns for minimum cost path computed by both the forward and the reverse search, shortest path, two columns each node's predecessor (in the forward search) and successor (in the reverse search) [4], and two flag columns denoting if the current node has been finalized in either of the search directions.

Similarly to the version of the algorithm in the FEM framework, the core of our implementation is a single SQL MERGE statement combining the work of selecting, expanding and relaxing the frontier nodes, but the two implementation differ in two notable aspects:

[4]The node's predecessor and successor information is used to retrieve the actual path.

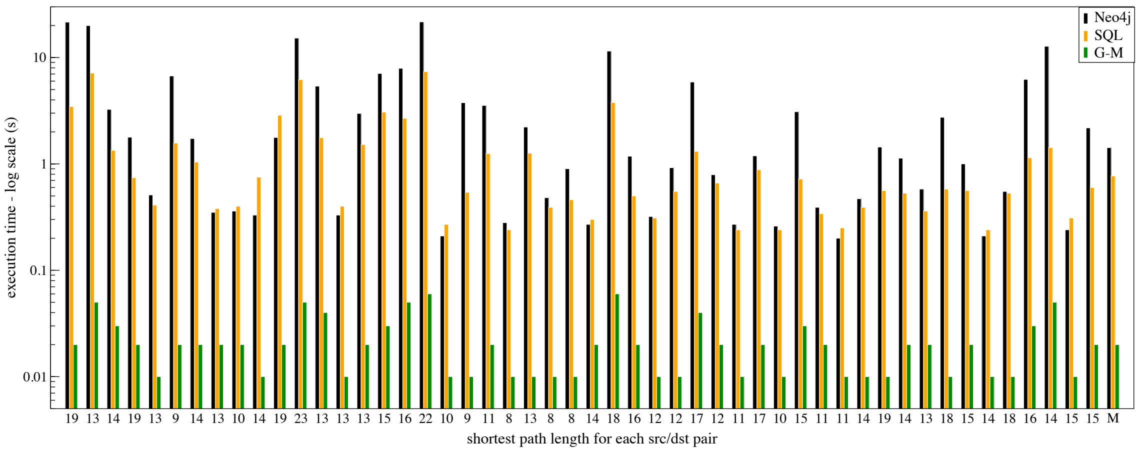

Figure 1: Execution times for LIVEJ graph (50 random src/dst pairs)

1. The IWT is created as a temporary table and no index on this table is used. This is based on a careful evaluation of many different table and index configurations including partitioned table, regular table, temporary table, compressed table, uncompressed table, global index, local index, and no indexes. Empirical results suggest that especially analyzing graphs representing large-scale social networks, a temporary table with no indexes produces the most promising performance.

2. The SQL MERGE statement used in our algorithm implementation not only does the node selection, expansion and relaxation, but also carries out the maintenance of the global minimum path cost (see Section 2.1.3) between the source node and the destination node. In the FEM framework's implementation computation of the global minimum path cost involved execution of an additional SQL statement.

4. PERFORMANCE EVALUATION

As discussed in Section 2, for our performance evaluation we choose to compare the best implementation of Dijkstra's shortest path available on each platform. We are evaluating these algorithm using the following social graphs:

- **LIVEJ** – graph representing members of a free on-line community (available from Stanford's Large Network Dataset Collection [5]): ∼4.8M vertices, ∼69.0M edges, average degree ∼14.2, maximum degree 20293

- **TWITTER** – graph representing Twitter user profiles and social relations between users (available as part of the Twitter site analysis work by Kwak et al. [8] [6]): ∼41.7M vertices, ∼1.47B edges, average degree ∼35.3, maximum degree 2997469

Both of these graphs come with just the connectivity information (nodes and edges) – for the purpose or running the shortest path computations we generated random integer edge costs in the 0–100 range.

[5]http://snap.stanford.edu/data/soc-LiveJournal1.html
[6]http://an.kaist.ac.kr/traces/WWW2010.html

4.1 Experimental setup

All numbers have been collected on an Intel Xeon E5-2660 (Sandy Bridge) machine (2x 8 hyper-threaded 2.2GHz cores, 264GB of RAM) running 64-bit SMP Linux 2.6.39. The machine has been equipped with 4 250GB SSD drives. Configuration options specific to a given graph analysis platform are specified below.

4.1.1 Green-Marl

As described in Section 3.1, algorithms expressed in Green-Marl language are translated to C++ by the Green-Marl compiler – the final binary was compiled using gcc v4.4.7. Default options for both the Green-Marl compiler and the Green-Marl runtime have been used when conducting the experiments. We used the Green-Marl distribution available in the open source repository dated at 28/03/2013. Prior to executing each timed run, the respective graph has been loaded into Green-Marl's in-memory representation from a Green-Marl-specific binary file format (we report graph loading overheads below).

4.1.2 Neo4j

In this paper, we evaluate the open-source community edition of Neo4j version 1.8.2, which is the latest stable release of Neo4j at the time of writing this paper. We use Java JDK v1.6.0_43 running the *server* class JVM in 64 bit mode.

We have experimentally chosen the minimum memory configurations such that the garbage collection (GC) overhead is less than 5% of the total execution time: for the **LIVEJ** graph we set the JVM heap size to 16GB and for the **TWITTER** graph to 32GB. In order to get the best results for the **TWITTER** graph we had to change Neo4j' default cache configuration from "soft" to "none". Otherwise the GC overhead was prohibitive resulting in extremely long execution times – even after increasing the heap size to 128GB, with the soft cache, the GC overhead was ∼48% resulting in a nearly 2x slowdown compared to the 32GB heap size configuration with no cache. The remaining Neo4j parameters have been calculated based on the size of each input graph, as described in Neo4j's documentation.

After graph data stored in the file system has been batch-loaded into Neo4j (see Section 3.2), we shutdown the

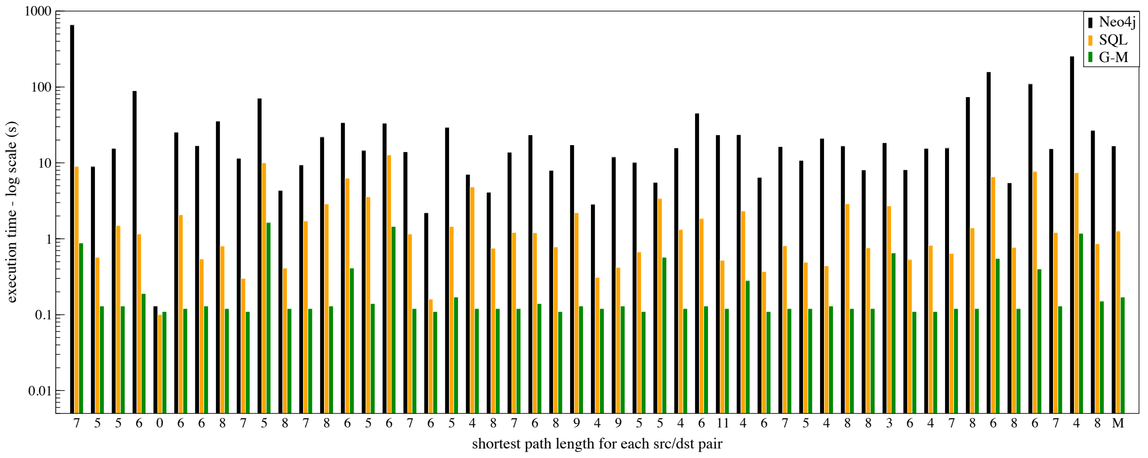

Figure 2: Execution times for TWITTER graph (50 random src/dst pairs)

database so that it is stored to disk. We re-open the same database as an Embedded Read Only Graph Database in Neo4j (we found this configuration to deliver better average performance than the "standard" Embedded Graph Database). Then, we use the bi-directional Dijkstra implementation that comes with Neo4j to compute shortest paths on this database.

4.1.3 SQL

A pre-release version of Oracle Database 12c was used as the RDBMS to run our proposed SQL based implementations. The graph data together with the SQL logic were managed by a 12c Pluggable Database (PDB) [13] which is planned as a new feature for multi-tenancy. Out of the total 264GB of RAM, we allocate 8GB and 8GB for Program Global Area (containing data and control information for database processes) and System Global Area (part of RAM shared by all processes that belong to a database instance), respectively. We have experimentally confirmed that this memory configuration yields the best performance for the SQL runs.

4.2 Results

In Figures 1–2, we plot the execution times for the best available implementation of Dijkstra's bi-directional shortest path algorithm available on each graph analysis platform: non-set-based versions for Neo4j and Green-Marl, and set-based version for SQL.

In our opinion, a typical use of a graph analysis engine is to accept and process multiple requests with a varying input. Consequently, we report execution times for 50 randomly chosen source/destination pairs for each graph, measured within the same invocation of the graph analysis engine [7]. We present both "cold" execution times (to the left of each plot) and "warm" execution times (to the right of each plot) [8]. Execution times are plotted on the logarithmic scale and the labels on the x-axis indicate the length of each shortest path found (0 for cases where no path has been found). The right-most set of bars on each plot (labeled "M") represents a geomean of the execution times for all pairs.

The analysis of the execution times presented in Figures 1–2 confirms the premises underlying this paper:

1. When analyzing social graphs, performance of an optimized SQL-based implementation of Dijkstra's shortest path algorithm is competitive with (Figure 1) or better (Figure 2) than the best implementation of the same algorithm available in Neo4j.

2. The best implementation of Dijkstra's shortest path algorithm by the in-memory graph analysis engine significantly outperforms both database-based solutions.

Please note that we do not claim that SQL-based implementation will always outperform the one provided by Neo4j. We merely observe that the SQL-based version works well for a certain important class of graphs, that is social graphs. As the cost of a single SQL operation is fairly high, performance of the SQL-based solutions is bound to suffer when dealing with algorithms or graphs that trigger execution of a large number of SQL statements. In particular, we observe that the SQL-based shortest path algorithm applied to road network graphs is significantly slower than the Neo4j version due to much longer paths (on the order of thousands) whose discovery can involve hundreds of thousands of SQL operations. However, in the same setting, Green-Marl still significantly outperforms Neo4j.

Please also note that there exists an inherent tradeoff between both database-based solutions and the in-memory solution. The in-memory solution offers superior performance but requires loading the whole graph into memory before the analysis can begin. Consequently, depending on the use-case scenario, the in-memory solution may be favored over a database-based one or vice-versa. The graph loading times (not included on the plots), for Green-Marl-specific binary file format residing in the filesystem are ∼7.7 seconds for the **LIVEJ** graph and ∼153.8 seconds for the **TWITTER**

[7]During our experiments, we have collected numbers for 200 random pairs but the execution times for the remaining 150 pairs do not contribute any additional significant information.

[8]We observed the difference between "cold" and "warm" execution times for the same pair reaching up to ∼65% in favor of the "warm" runs, depending on the analysis engine used and the graph being analyzed.

graph. An additional related trade-off exists with respect to the memory footprint – SQL and Neo4j have to load only portions of the graphs into memory and thus their memory requirements (see Section 4.1.2 and Section 4.1.3) can be significantly lower than dynamic memory footprint we observed for Green-Marl runs (up to ∼39GB for the **TWITTER** graph but, at the same time, only ∼3GB for the **LIVEJ** graph).These trade-offs motivated us to propose a unified solution combining a "traditional" database-based approach with an in-memory approach.

5. UNIFIED GRAPH ANALYSIS

In-memory graph analysis systems typically do not provide a comprehensive data storage solution. Furthermore, as graph analysis requests often do not mutate the graph, the implementation of these systems is optimized towards the read-only workloads. For example, Green-Marl's in-memory graph representation (CSR), described in Section 3.1, is compact and efficient when supporting read-only analysis requests, but it is not designed to support frequent graph mutations. Finally, in-memory solutions may have to load the whole graph into memory before the analysis can begin.

On the other hand, since one of the main tasks of the databases is to handle data mutation in addition to read-only analysis requests, graph analysis solutions utilizing the database technology can handle graph updates in a very natural way. These solutions can also achieve reasonable performance due to their ability to cache data being incrementally loaded into memory, but they have difficulties matching performance of the in-memory solutions as, in addition to guaranteeing that data in the cache and in the persistent store are in-sync, they often moderate access to data using strong consistency models, such as ACID [5].

Based on these observations, we would like to propose an architecture that combines both styles of graph analysis, and plan to build a corresponding system as part of our future work. It is our belief, backed up by performance evaluation presented in this paper, that such system can be layered on top of a relational database. The main idea is simple – use Green-Marl as the high-level abstraction, target the same schema representing graphs in the relational database, and utilize multiple Green-Marl compiler backends to translate the high-level abstraction to different low-level graph execution engines that the system's runtime can dynamically and automatically select based on the current use-case scenario.

Such system would deliver a powerful "knob" providing the ability to dynamically adjust and control the trade-off between different styles of graph analysis. In particular, the compiler could generate both the SQL code and also the C/C++ code targeting custom in-memory graph representation. Our plan also includes building a backend targeting a hybrid graph execution engine that would offer a comprehensive alternative to graph databases by maintaining an update-able in-memory graph representation supporting fine-granularity caching of data retrieved from the relational database. The hybrid solution would then be capable of both reducing the graph loading cost of the current in-memory solution and mitigate SQL's per-operation overhead. This idea is conceptually similar to the approach presented by Sakr et al. [14] who maintain a portion of graph data in memory (nodes and edges) and another portion in a relational database (properties). Their focus, however, is on graph querying as opposed to our target algorithms which are "global" in nature (e.g. pagerank) and it is unclear if the choice of fixed division of graph data would work well in such context. The solution we propose would likely not perform quite as well as the current in-memory systems such as Green-Marl but considering that Green-Marl is from several to several hundred times faster than Neo4j when analyzing graphs of any size, we believe that design-wise it will be possible to find an acceptable trade-off.

6. CONCLUSIONS

In this paper, we presented empirical evidence that for certain classes of graphs, solutions utilizing relational database technology and in-memory graph analysis techniques can offer performance superior to that of the dedicated graph databases. We also observed that these performance improvements come with a certain trade-off and proposed an architecture allowing the runtime to dynamically control this trade-off.

7. REFERENCES

[1] D. A. Bader and K. Madduri. Snap: small-world network analysis and partitioning. `http://snap-graph.sourceforge.net`, 2010.

[2] O. A. R. Board. OpenMP application program interface. `http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf`, 2013.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1959.

[4] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. In *VLDB*, 2012.

[5] J. Gray and A. Reuter. *Transactional Processing: Concepts and Techniques.* 1992.

[6] T. Hoff. Neo4j - a graph database that kicks buttox. `http://highscalability.com/neo4j-graph-database-kicks-buttox`, 2009.

[7] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[8] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[9] M. Luby and P. Ragde. A bidirectional shortest-path algorithm with good average-case behavior. Algorithmica, 1989.

[10] Neo4j. Neo4j customers. `http://www.neotechnology.com/customers/`, 2013.

[11] Neo4j. Neo4j, the world's leading graph database. http://www.neo4j.org/, 2013.

[12] P. Neubauer. Graph databases, NOSQL and Neo4j. `http://www.infoq.com/articles/graph-nosql-neo4j`, 2010.

[13] M. Rajendran. Oracle database 12c: New features - pluggable databases. `http://www.orafaq.com/node/2756`, 2012.

[14] S. Sakr, S. Elnikety, and Y. He. Gsparql: A hybrid engine for querying large attributed graphs. In *CIKM*, 2012.

[15] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In *STACS*, 2007.