Towards a Benchmark for Graph Data Management and Processing

Michael Grossniklaus
University of Konstanz
Germany
michael.grossniklaus
@uni-konstanz.de

Stefania Leone
University of
Southern California
United States of America
stefania.leone@usc.edu

Tilmann Zäschke
ETH Zurich
Switzerland
zaeschke@inf.ethz.ch

ABSTRACT

Graph data is used in an increasing number of analytical data processing applications, ranging from social network analysis, to monitoring of network traffic, and to ontologies in the semantic web. Both application-specific and general graph data management systems have been proposed to address this trend. Nevertheless, it would be foolish to dismiss relational and object databases as possible solutions for graph data management and processing, due to the vast amount of experience that they encompass. As a consequence, graph data analysts are faced with a broad variety of choices and approaches. However, regardless of the approach taken, users need to be able to evaluate and assess which of the many possible solutions works best in their use case. We propose a benchmark in terms of a data model, query workload, sample data sets, and usage scenarios. We also report performance figures measured based on an open-source graph database as well as commercial relational and object databases. Finally, we discuss how our benchmark can be used to characterize databases with respect to particular application scenarios.

1. INTRODUCTION

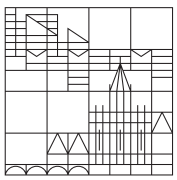
A large and growing number of database applications require graph data management and processing. Use cases include monitoring network data traffic, analyzing email conversations, querying RDF graphs, recommender systems, computational biology and social network analysis [31][1]. In particular, data management for social networks is gaining importance as evidenced by events such as the DBSocial [6] workshop at SIGMOD 2011. Apart from this trend, we believe there are other reasons why graph data will become even more important as an application domain for databases in the near future. First, the graph data model [2] is relatively simple and lends itself to formal specification, not unlike the relational model. Second, there is a wide range of precisely defined operations to process and transform graphs. Furthermore, in-memory algorithms implementing these operations are well known in terms of time and space complexity, opening up opportunities for the development of out-of-core counterparts. Finally, general graph databases such as Neo4j[2], OrientDB[3], or HyperGraphDB[4] as well as domain-specific solutions, e.g. Jena[5] or Sesame[6] for RDF graphs, have recently been proposed.

While these recent proposals pursue the approach of developing novel database systems that provide data models and operators specific to the domain of graphs, an alternative approach is to develop a specialized system by building a layer on top of an existing database to provide the additional functionality. Examples of this approach from the domain of astronomy data include the Sloan Digital Sky Survey (SSDS) SkyServer [25] that builds on Microsoft SQL Server[7], the Herschel telescope [29] of the European Space Agency (ESA) that uses Versant[8] in their data storage and processing system, and the US Air Force's Space Situational Awareness Foundational Enterprise (SSAFE) program [21] which is based on Objectivity/DB[9]. All of these systems have demonstrated that this second approach can be equally, if not more, successful than building a novel type of database system from scratch, due to the vast body of experience and knowledge that is encompassed by existing databases. At this point in time, we therefore believe that it would be unwise to dismiss time-tested relational and object databases as possible solutions for graph data applications.

As a consequence, a professional who faces graph data processing tasks has a plethora of possible choices to make. First, they need to decide which of the two approaches they follow, i.e. whether they use one of these new graph databases or whether they are better served by building a layer on a traditional database. In a next step, they need to select a specific database from a large and constantly growing number of systems. Then, they need to decide how to store graph data in that system at the logical level. While the graph data model is simple and well-defined at the concep-

[1]It is important to note that the discipline of social network analysis predates "Web 2.0" applications such as Facebook and Twitter and, therefore, the term "social network" has a far broader meaning than its common use today.

[2]http://www.neo4j.org
[3]http://www.orientechnologies.com
[4]http://www.kobrix.com/hgdb.jsp
[5]http://jena.sourceforge.net/
[6]http://www.openrdf.org/
[7]http://www.microsoft.com/sqlserver/
[8]http://www.versant.com
[9]http://www.objectivity.com



University of Konstanz
Department of Computer and Information Science
Technical Report **KN-2013-DBIS-01**

tual level, it is characteristic of graph data that the best logical representation can vary considerably depending on the set of queries required by the application. As we will see, this is even true if a graph database is selected in the first step. Finally, they need to decide whether to implement the desired graph operations as queries within the database or in a client-side application. We are aware that many of these decisions also occur in traditional database applications, but in the case of graph data the performance implications of these choices are more difficult to understand. For example, the performance of certain graph queries is dominated not by the size of a graph, but by other factors such as the number of connected components or its diameter.

We address this problem and propose a benchmark for graph data management and processing that we have designed to meet the following criteria. First, the benchmark needs to include all types of graphs that are relevant in practice. This criterion implies that the benchmark works with a wide variety of publicly available data sets that consist in directed or undirected and weighted or unweighted graphs alike. Second, the core query workload should be a minimal set of general and representative graph operations that can be specified unambiguously. By *general* we mean that the selected operations should be common to as many graph data applications as possible, while by *representative* we mean that these operations are capable of precisely gauging data set and application-specific performance implications of a database. Finally, the benchmark should also meet the criterion of being customizable and extensible to cater for future or unforeseen requirements as argued by Gray [16]. We concur with the argument that there is a need for application-specific benchmarking [23] and propose our benchmark as a solution for applications that involve graph data, for which existing benchmarks such as TPC-C or TPC-H [27] are not suitable. The main contributions of this paper are as follows.

- **Specification** of a standard benchmark for graph data management and processing, which consists of different parts that each addresses one of the above criteria.
- **Validation** of the benchmark by means of three different implementations.
- **Experiences** and **best practices** to guide the selection of a database with respect to the characteristics of both the data set and the graph application.

We begin in Section 2 by defining the proposed benchmark. It consists of an abstract data model, a query workload, a collection of sample data sets, and use scenarios. We discuss how the benchmark was implemented in three different commercial database systems in Section 3. The chosen systems include a relational, an object and a graph database. Performance results obtained from these implementations are reported and analyzed in Section 4. In Section 5, we discuss experiences and best practices from a system-level and user-level point of view. Related work is discussed in Section 6 and we conclude in Section 7, where we also provide directions of future work.

2. GRAPH DATA BENCHMARK

In this section, we define the proposed benchmark for graph data management and processing. Generic desiderata and guidelines for specifying a concrete graph benchmark have been identified by Domínguez-Sal *et al.* [12] and serve as a starting point for our work. We begin by identifying the required properties of the conceptual graph data structure that each experimentation platform needs to map into a physical representation. Then, we continue by specifying a workload of nine graph data queries that forms the basis for performance measurements in our benchmark. Finally, we discuss possible data sets for experimentation and establish how applications can be characterized by the query workload of our benchmark.

2.1 Data Model, Terminology and Notation

The abstract graph data structure used in the proposed benchmark follows the commonly accepted definition. A *unweighted graph* $G = (V, E)$ is a set of *nodes* V and a set of *edges* E. Edges are pairs (u, v) of distinct nodes u and v that can either be ordered or unordered. In the former case, the graph is said to be *directed* and *undirected* in the latter. In addition, a *weighted graph* $G = (V, E, \omega)$ specifies a total function $\omega : E \to \mathbb{R}$ that assigns a weight w to each edge $(u, v) \in E$. In the following, we sometimes refer to directed edges as *arcs* and to undirected edges simply as *edges*. We denote the existence of an arc $(u, v) \in E$ by $u \to v$ and the existence of an edge $(u, v) \in E$ by $u - v$.

A *path* from v_1 to v_n, denoted by $p : v_1 \overset{*}{\to} v_n$, is a sequence of nodes $\langle v_1, v_2, \ldots, v_n \rangle$ such that $\forall\ 1 \leq i < n : \exists\ (v_i, v_{i+1}) \in E$. We say an edge $(v_i, v_{i+1}) \in p \Leftrightarrow \exists\ i : \langle v_i, v_{i+1} \rangle \sqsubseteq p$, where $\sqsubseteq$ denotes a consecutive subsequence. The set of all possible paths from node u to node v is denoted by $P(u, v)$. Similarly, $P(u, v)[w]$ denotes all possible paths from u to v that pass through node w, where $u \neq v \neq w$. A graph is *connected* if there is a path between every pair of nodes.

2.2 Query Workload

The workload of nine queries has been assembled by selecting common and well-known graph algorithms as well as often-used metrics from the domain of social network analysis [31]. As Domínguez-Sal *et al.* [12], we concur that social network metrics form a viable core for a graph benchmark.

2.2.1 Q1: Transitive Closure

We adopt the definition of transitive closure that is commonly used in computer science, e.g. [24], as a data structure to efficiently solve reachability queries in a graph. More formally, the transitive closure of a graph $G = (V, E)$ is a graph $G' = (V, E')$, where $E' = \{(u, v) \mid \exists\ p : u \overset{*}{\to} v\}$. Query Q1 computes the transitive closure of the entire graph.

2.2.2 Q2: Node Degrees

The degree of a node $v \in V$ in a graph $G = (V, E)$, denoted as $deg(v)$, is the number of edges incident to that node. Formally, $deg(v) = |\{(x, y) \mid (x, y) \in E, x = v \wedge y = v\}|$, where $|\cdot|$ denotes set cardinality. Query Q2 computes the node degrees of all nodes in the graph.

2.2.3 Q3: Connectedness

In a graph $G = (V, E)$, two nodes u and v are connected, if $\exists\ p : u \overset{*}{\to} v$. For all possible pairs of nodes in the graph, query Q3 checks whether they are connected.

2.2.4 Q4: Shortest Paths

In a weighted graph, a shortest path $\hat{p} : u \overset{*}{\to} v$ minimizes the sum of the weights of the edges that are incident to

the nodes on $\hat{p}$, i.e. $\sum_{e \in \hat{p}} \omega(e)$. Note that this definition can easily be extended to unweighted graphs by using $\omega' : E \rightarrow const$ as a weight function. For all possible pairs of nodes in the graph, query Q4 computes the shortest path and retrieves the edges of each path.

2.2.5 Q5: Degree Centrality

The degree centrality of a node v in a graph $G = (V, E)$, denoted by $C^D(v)$, is defined as the number of edges incident to v, normalized by the number of other nodes in G. Formally, degree centrality is defined as

$$C^D(v) = \frac{deg(v)}{|V \backslash v|}.$$

For each node of the graph, query Q5 computes its degree centrality.

2.2.6 Q6: Closeness Centrality

The closeness centrality of a node v in a graph $G = (V, E)$, denoted by $C^C(v)$, is given by the inverse of the sum of its geodesic distances to all other nodes in G. The geodesic distance $\delta(u, v)$ between nodes u and v is the total weight of the shortest path $\hat{p} : u \stackrel{*}{\rightarrow} v$. Formally, closeness centrality is defined as

$$C^C(v) = \frac{1}{\sum_{u \in V \backslash v} \delta(v, u)}.$$

For each node in the graph, query Q6 computes its closeness centrality.

2.2.7 Q7: Betweenness Centrality

The betweenness centrality of a node v in a graph $G = (V, E)$, denoted by $C^B(v)$, is defined as

$$C^B(v) = \sum_{u,w \in V \wedge u \neq v \neq w} \frac{|\hat{P}(u, w)[v]|}{|\hat{P}(u, w)|}.$$

In other words, for all possible pairs of nodes u and w, betweenness centrality sums the ratio of all shortest paths that pass through v and the total of all shortest paths between these nodes. Query Q7 computes the betweenness centrality of all nodes in the graph.

2.2.8 Q8: Bridges

A bridge in a graph $G = (V, E)$ is an edge $e \in E$ that, if deleted, increases the number of connected components. Query Q8 identifies and retrieves all edges that are bridges in the graph.

2.2.9 Q9: Diameter

The diameter d of a graph $G = (V, E)$ is defined as the longest shortest path between any of its nodes $u, v \in V$, i.e.

$$d = \max_{\forall u,v \in V} \delta(u, v),$$

where $\delta(u, v)$ is the geodesic distance in terms of the total weight of the shortest path $\hat{p} : u \stackrel{*}{\rightarrow} v$.

2.2.10 Query Classification

Domínguez-Sal *et al.* [12] presents a comprehensive categorization of possible graph queries. While we have based our work on theirs, we use an alternative and more practical classification of graph queries in this paper.

Scope The scope describes the result of a query in terms of the complexity of its input. A query that returns information about one node has scope *node*. A query with scope *path* relates two nodes to each other. A query that starts from a subset of nodes has scope *subgraph*. Finally, a query that does not limit its input to any particular set of nodes has (implicitly) scope *graph*.

Radius The radius of a query captures the pattern in which data is accessed by a query starting from the input scope. A query that only accesses the nodes in its input scope has radius *local*. If a query accesses a node and its neighbors up to paths of a fixed length, it has radius *neighbors*. A query that accesses the graph by traversing paths (of arbitrary length) starting from its input nodes has radius *path*. Finally, a query that accesses arbitrary parts of the graph and is therefore not bound by an access pattern has radius *global*.

Result Graph queries may return sets of *values*, *nodes*, or *edges*. In many cases, the set might only contain a single value. Finally, if both nodes and edges are returned, the result of the query is of type *subgraph*.

Dependencies Knowledge of dependencies between graph operations enables a system to pre-compute or cache query results for reuse by other queries. This "optimization" is particularly interesting in the analytical setting that is typical of many graph processing applications, due to the small number of expected updates.

Complexity As initially mentioned, most graph operations are well researched in terms of possible algorithms to implement them. As a consequence, the space and time complexity of each of those algorithms is known and can contribute to characterize our queries further.

Table 1 summarizes the proposed query workload according to this classification. Note that we have omitted the *complexity* category since this information is well-known for the algorithms discussed in this paper. As can be seen from the table, there is no query with scope *subgraph* or *radius* neighbors. An example of a query with a subgraph scope is one that performs graph or subgraph matching. Since this is a central part of the SPARQL query language for which specialized benchmarks have been proposed [17], we chose not to include it. A query that could be included in the benchmark to represent *radius* neighbors is retrieving the friends of friends of a given node. Our decision not to include this query at this point is motivated by the fact that this operation is more typical of an operational setting such as hosting a Web 2.0 site, rather than an analytical task.

2.3 Data Sets

Graph and network data sets are publicly available in many formats. In the context of the proposed benchmark, we have decided to support the Pajek [10] data format, since it is well-documented and widely used. A detailed discussion of the software that we provide to parse and load Pajek data sets can be found in Appendix A. For the purpose of the experiments presented in this paper, we have selected three data sets from an online collection of Pajek data sets [7].

Table 2 gives an overview of the characteristics of the chosen data sets. The *NetScience* data set [20] is a fairly small

Query	Scope	Radius	Result	Dependencies
Q1	graph	global	edges	–
Q2	node	local	value	–
Q3	path	path	value	Q1, Q4
Q4	path	path	edges	Q1
Q5	node	local	value	Q2
Q6	node	path	value	Q1, Q4
Q7	node	global	value	Q1, Q4
Q8	graph	global	edges	–
Q9	graph	global	value	Q1, Q4

Table 1: Workload query classification

undirected weighted graph with 396 connected components that captures co-authorships in the domain of network theory and experimentation. Nodes represent authors and two authors are connected by an edge if they have co-authored a publication. The weight of such an edge is a score defined by Newman [20]. Similarly, the *Geom* data set represents collaboration in the field of computational geometry and has 2060 connected components. The weight of an edge connecting two authors who have co-authored a paper is the number of joint works. Finally, the *Erdos* data set represents co-authorship as an undirected unweighted graph as its primary purpose is the computation of the so-called Erdős Number [15]. Since it only contains one component, it is fully connected.

	NetScience	Geom	Erdos
Nodes	1589	7343	6927
Edges	2742	11898	11850
Directed	–	–	–
Weighted	✓	✓	–
Components	396	2060	1
Diameter	17	14	4
Average Path Length	5.763	5.310	3.776

Table 2: Data set characteristics

We realize that all of these data sets are small in comparison to other examples of the graphs and networks that recently have been cited in the literature. Nevertheless, we argue that there are valid reasons to use these data sets for the purpose of this work. First, we do not propose novel graph processing technique and therefore absolute performance and scalability is not important. However, what is important is that the proposed benchmark can be used to reliably measure relative performance. Second, intermediate and final results of some the queries included in the proposed workload can be much larger than the original graph itself. For example, the transitive closure of the *Erdos* data set contains close to 48 million edges, since the graph has only one component. Finally, many of the newly proposed graph databases put strong emphasis on graphs that can be entirely managed in main memory. Therefore, one of our goals is to investigate the performance of these systems in conditions under which the data either do or do not fit into the main memory of a commodity personal computer, e.g. 4 or 8GB. The *NetScience* data set has been chosen to quickly validate implementations of the benchmark in terms of correctness. Data set *Geom* has been selected as it requires less than 4GB, both for disk storage and for processing in main memory. Most of the databases we have tested, store the *Erdos* data set using less than 4GB on disk. However, keeping the entire data set and all intermediate results in main memory during query processing is typically not possible.

2.4 Characterizing Applications

Due to the many applications using graph data, there are several possible scenarios of how a graph data is processed. Therefore, a benchmark for graph data management and processing has to be a reliable predictor of a given database system for a given class of applications. Below, we define example application classes and show how each of them could be characterized in terms of a subset of queries from the workload. We note that this list is not exhaustive and other classes could be added analogously.

Network traffic analysis and routing In this usage scenario, weighted undirected graphs are used to represent computer networks in terms of hosts (nodes) and network links (edges). In order to analyze and route network traffic, the topology of a graph is of importance. As a consequence, queries of interest for this scenario are Q2, Q3, Q4, Q8, and Q9.

Social network analysis Graphs are used to represent actors (nodes) and their relationships (edges). A graph can contain different types of actors and various kinds of relationships, e.g. friendships, co-authorship, collaboration, etc. Centrality measures that aim to capture the importance of an actor in a social network play a central role in the analysis of these graphs. Therefore, queries of interest are Q5, Q6, and Q7. Most centrality measures capture *relative* importance of a node and it is therefore typically necessary to compute the centrality of several nodes at at time. As a consequence, it makes sense to pre-compute the transitive closure (Q1) as a basis for the computation of Q6 and Q7.

Data provenance Some database systems use directed graphs (or trees) to represent the provenance (edges) of a data item (nodes). A typical query in such a system is expanding the ancestor tree of a single node (data item). Queries of interest for this traversal-based usage scenario are Q3 and Q4.

3. IMPLEMENTATION

To validate the proposed benchmark, we realized it in three database systems. As motivated in Section 1, there are several possible approaches to managing and processing graph data. While emerging graph databases are a good fit in terms of data model and operations, we have to assume that they are not yet as mature as relational databases. Finally, we also consider object databases to be candidates for graph applications, due to the similarity of the object to the graph data model. In all three implementations, we assume the usage scenario of social network analysis as described in Section 2.4. In line with the arguments presented there, we choose to persist the result of Q1 and reuse it in other queries, whenever this is possible and yields a performance benefit over the direct approach.

3.1 Relational Database

To represent graphs in our RDBMS implementation of the benchmark, we have chosen the straightforward approach of

using two tables to store node data and (directed) edges, respectively. The former is defined as NODES(Id: *int*, Label: *varchar*) and the latter is defined as ARCS(SourceId: *int*, TargetId: *int*, Distance: *decimal*), where SourceId and TargetId are foreign keys referencing NODES.Id and the primary key of the relation is clustered over SourceId and TargetId, sorted in ascending order. Unweighted graphs are represented by setting the Distance column to *null*, while undirected graphs are modelled based on a view EDGES(SourceId, TargetId), defined as the union of ARCS and its "inverse".

To compute the *transitive closure* of a graph, we have implemented two alternative algorithms in SQL. The first follows the well-known Floyd-Warshall algorithm [14, 30], with its three nested loops realized as cursors over the NODES table. The second algorithm follows the so-called "naive" approach [4] of recursively joining the ARCS (or EDGES) table to itself until no new result tuples are generated. We are aware of the vast body of research, e.g. [18, 1], investigating the computation of the transitive closure in relational systems. However, since it is not our goal to repeat previous results or design new algorithms, we have chosen to implement basic variants of these algorithms.

To store the result, both of our implementations populate the table TRANSITIVECLOSURE(SourceId: *int*, TargetId: *int*, PathSize: *int*, Distance: *decimal*, ParentId: *int*, PathCount: *int*). Apart from source and target nodes, this relation also records the size (in terms of steps) or the length (in terms of distance) of the shortest paths for unweighted and weighted graphs, respectively. We also keep track of the parent or predecessor of a node on the shortest path, allowing a shortest path to be reconstructed later. Finally, both algorithms also compute and store the number of shortest paths between each source and target node pair. The primary key of the TRANSITIVECLOSURE relation is clustered over SourceId and TargetId in ascending sort order.

In this setting, the *degree of a node* is easily computed based on the EDGES view (not the ARCS table) by selecting and grouping over the SourceId column and counting the TargetId column. We check whether two nodes are *connected* with a simple selection on the TRANSITIVECLOSURE table. The stored procedure to reconstruct a *shortest path* between two nodes recursively selects the parent or predecessor node from the TRANSITIVECLOSURE relation and adds the constructed edges to a result table. A node's *degree centrality* is computed analogously to its degree. The *closeness centrality* of a node is found by summing all path sizes (or lengths) for one source node and returning its reciprocal.

Due to its imperative nature, we did not use Brandes' algorithm [8] to compute the *betweenness centrality* of a node in the SQL implementation of the benchmark. Rather, we use the declarative approach shown in Listing 1 that follows the definition more directly. For every possible pair of a source and a target node, the function uses the TRANSITIVECLOSURE relation to determine both the total number of shortest paths between those nodes and the number of paths that go through the node identified by `@id`.

```
create function BetweennessCentrality(@id int)
    returns float
begin
  declare @result float
  select @result = sum(cast(aux.PathCount as float)
      / aux.TotalPathCount)
  from (
    select l.SourceId as SourceId,
           r.TargetId as TargetId,
           (l.PathCount * r.PathCount) as PathCount,
           pvt.PathCount as TotalPathCount
    from   dbo.TransitiveClosure l,
           dbo.TransitiveClosure r,
           dbo.TransitiveClosure pvt
    where  l.TargetId = @id and r.SourceId = @id and
           l.SourceId = pvt.SourceId and
           r.TargetId = pvt.TargetId and
           (l.PathSize + r.PathSize) = pvt.PathSize
  ) as aux
  where aux.TotalPathCount <> 0 and
        aux.PathCount <> 0
  return @result
end
```

Listing 1: SQL implementation of (unweighted) betweenness centrality

The SQL function to find *bridges* in a graph closely follows Tarjan's algorithm [26]. In a first step, we therefore create a spanning tree starting from a table containing a root node and repeatedly expanding it by joining it with the EDGES relation until we have processed all nodes. The second step of Tarjan's algorithm calls for a post-order traversal of the spanning tree which we accomplish by iteratively descending as far as possible (and required) into the tree from the root and updating the selected node. We realize the third step of the algorithm using a cursor to iterate over the nodes in ascending order while computing $ND(v)$, $L(v)$, and $H(v)$ as described by Tarjan [26]. Doing so, we build a new relation TARJAN(Id: *int*, Order: *int*, ND: *int*, L: *int*, H: *int*), where Order is the node's post-order position assigned in the previous step. Finally, we join tables TARJAN and EDGES on Id and TargetId, and select all (SourceId, TargetId) pairs that meet the condition $H \leq Order \wedge L > (Order - ND)$. In order to process graphs that have multiple components all at the same time, the relations used in the implementation of Tarjan's algorithm keep track to which tree a node belongs. For the sake of simplicity, we have chosen to omit this implementation detail from this discussion.

The query to compute the *diameter* of a graph again relies on the TRANSITIVECLOSURE relation and simply selects the maximum value of column PathSize (in unweighted graphs) or of column Distance (in weighted graphs).

3.2 Object Database

The ODBMS implementation is based on a commercial product that is accessed via a Java interface, which was used to implement the algorithms on the client side.

To represent graph data in the object database, we have defined two persistent classes, one for nodes and one for edges, as illustrated in Figure 1. An EDGE instance defines an `edgeID`, references to its `source` and `target` nodes as well as a possible weight. In unweighted graphs, the weight is set to 1. A NODE specifies a `nodeID`, a `label` and a list of `edges` that connect a node to its neighbours. To compute and represent the transitive closure, we further introduce the class EDGEPROPERTY. A node defines an attribute `rowIndex` which is a map from `nodeId` to EDGEPROPERTY, and a `rowIndex` contains one EDGEPROPERTY object for each node in the graph. Each EDGEPROPERTY object defines the distance between the two nodes (including the weight for weighted graphs), the number of shortest paths between the two nodes as well as a reference to the predecessor node of that node. The predecessor node is needed to reconstruct the shortest path. Note that since we never access EDGEPROPERTY objects directly, but always through

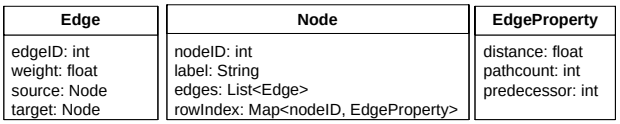


Figure 1: UML class diagram

the `rowIndex`, we do not treat EdgeProperty as a persistent class. We store it as a serialized object (BLOB) for reasons of performance.

The computation of the *transitive closure* is based on a slightly modified version of the Floyd-Warshall algorithm. Our modified algorithm produces three matrices: a distance matrix, a predecessor matrix and a matrix recording the number of shortest paths between two nodes. We make use of these three matrices to initialize the `rowIndex` map for each node in the graph, which includes the instantiation of the EdgeProperty objects. The computation of the transitive closure based on a straightforward implementation of Floyd-Warshall is rather memory intensive, since we generate three matrices, where the size corresponds to the number of nodes in the network. As an alternative, we have implemented a persistent implementation for computing the transitive closure. In contrast to the first implementation, the persistent algorithm first initializes the `rowIndex` including all the EdgeProperty objects for each node and than gradually populates them while computing Floyd-Warshall. By doing so, the computed data is constantly written to the database. Note that in case of undirected graphs, the three matrices produced when computing Floyd-Warshall are symmetric. That means that we currently store redundant information: the values of an EdgeProperty for node m in node n's `rowIndex` corresponds to the values of the EdgeProperty for node n in the node m's `rowIndex`.

The *degree of a node* is computed by counting the number of edges of a node managed as `edges` attribute of the Node class. To check whether two nodes are *connected*, we access the `rowIndex` of one node and retrieve the EdgeProperty object for the other node, to check whether that node is reachable, i.e. whether there exists a predecessor. The *shortest path* between two nodes is computed with an algorithm that, for a given source node, recursively selects the predecessor attribute from the EdgeProperty of the target node in the `rowIndex` map and adds it to the result set. The *degree centrality* of a node is computed analogously to a node's degree, and the degree is then divided by the number of nodes in the network. A node's *closeness centrality* is computed by iterating over a node's EdgeProperties, summing up the distances and returning its reciprocal. The *betweenness centrality* of a node is computed based on the transitive closure, where we can find all shortest paths that pass through a given node by iterating through all nodes and their `rowIndex`s. Alternatively, we have an implementation of betweenness centrality based on Brandes' algorithm. Note that by using Brandes' algorithm, the whole graph has to be loaded into memory.

To find bridges in the graph, we rely on a straightforward implementation of Tarjan's algorithm. Here, we also need to load the whole graph into memory. For the *diameter* computation, we again rely on the transitive closure and find the longest shortest path by iterating over all nodes and their `rowIndex`s.

Access to persistent nodes and edges is currently realized through persistent lists and maps that load objects in batches of a fixed size which is configured according to the scope and radius of a query. Alternatively, objects could also be retrieved using queries, which would also load the objects in blocks of configurable size. In small graphs, the overhead of queries outweighs their benefits and navigational access is favorable. However, the navigational approach does not scale for large graphs, since the persistent collection data structures do not scale. Therefore, queries are favorable for large graphs, due to their constant overhead and scalability.

3.3 Neo4j

As a graph database, Neo4j provides constructs to manage node and edges. When nodes are created, Neo4j automatically assigns a system id to them. As is typical for so-called "NoSQL" databases, nodes do not have a schema and properties can freely be added to them. Since we do not have control over Neo4j's assignment of system ids, we have decided to also use application-specific ids that are represented as a node property. Unfortunately, this is necessary as our Pajek input format also uses its own ids to reference nodes from edge definitions. The only other node property we use stores the label of a node. Note that internally Neo4j stores all properties as strings, regardless of their Java type.

Edges in Neo4j are always directed in the sense that they have a distinct start and end node. Representing directed graphs in Neo4j is therefore straightforward. To represent an undirected graph, there are two possibilities. First, each undirected edge could be represented by two directed edges of opposite direction. Second, undirected edges could be represented as directed edges and treated as undirected at the level of the queries. The advantage of the former case are simpler queries, whereas storing the graph uses less space in the latter case. Our tests have shown that any possible performance gain of simple queries is dominated by the penalty of managing twice as many edges as necessary. As a consequence, we have implemented the second approach. Neo4j edges can have a type and properties. In the case of a weighted graph, we use an edge property to store its weight. The distinction between directed and undirected graphs is realized as a global property of the graph that is queried whenever edges are accessed. If a graph is directed only outgoing edges are accessed, otherwise both incoming and outgoing edges are returned. Without user-defined indexes, Neo4j only supports node and edge lookups by system id. To support lookups by application id, our implementation creates an index for nodes over the corresponding property. For edges we have defined an index that supports lookups by start node and/or end node.

We have implemented two variants of the Floyd-Warshall algorithms to compute the *transitive closure* of a graph in Neo4j. Since Neo4j only supports values of basic types or arrays of basic types as node and edge properties, both algorithms are forced to store the transitive closure by inserting additional edges of a new type between existing nodes. As properties, these edges store the predecessor and the size or length of the shortest path for unweighted or weighted graphs, respectively. The first variant, dubbed *persistent variant*, works directly on the graph by retrieving and inserting edges as it executes the Floyd-Warshall algorithm. The other variant, called *in-memory variant*, first reads all edges and allocates an adjacency matrix on the heap. Af-

ter running Floyd-Warshall on that matrix, it then serializes the new edges in one batch. All newly created edges of the transitive closure are indexed in the same way as the edges of the graph, but in a separate index.

A major issue in both variants is the fact that Neo4j manages all data structures of transactions in main memory (as opposed to a write-ahead logging). Therefore, the number of operations that can be performed in a single transaction is bound by main memory. Since the transitive closure adds as many as n^2 nodes, where n is the number of nodes in the graph, it is generally not possible to compute it in one Neo4j transaction. We have addressed this problem by intermediate commits after a fixed number of operations (persistent variant) and by using a batch update that circumvents transactions (in-memory variant). It should be noted that both solutions are inferior to the RDBMS implementation and therefore neither satisfactory nor equivalent.

There are two ways to compute the *degree* of a node in Neo4j. First, one can retrieve the node from the database and iterate over its outgoing (and incoming) edges. Second, one can look up the node in the edges index, without specifying a corresponding start or end node, and count the result set. Confirming our intuition, the second approach has performed consistently better in all our tests. A similar choice needs to be made when implementing the check whether two nodes are transitively *connected*. Again, we can retrieve the node and navigate to all other nodes that are connected to it by edges of the transitive closure, or we can look up the two nodes in the index for these edges. Interestingly, there are combinations of the total number of edges and the average node degree where the first approach performs better.[10]

To find *shortest paths* in a Neo4j graph, we have tested two possible implementations. The first is based on Neo4j's own graph algorithm library that provides an implementation of Dijkstra's algorithm [11]. The second is based on reconstructing a shortest path from the predecessor information stored along the edges of the transitive closure.

Our implementation of *degree centrality* reuses the query for a node's degree. However, we note that in Neo4j it is not easily possible to retrieve the total number of nodes (or edges) in the database. Therefore, we use a lookup on the node index for all nodes that have the id property set. The calculation of a node's *closeness centrality* is based on the path size (or length) information stored along the edges of the transitive closure incident to the node in question. The Neo4j implementation for *betweenness centrality* follows Brandes' algorithm.

To find the *bridges* of a graph we again use Tarjan's algorithm. The algorithm's implementation is straightforward, but we note that we experimented with two alternative versions. The first relies on Neo4j's own traversal framework to construct the initial spanning tree and label its nodes in post-order. In the second version, we implemented our own traversal routine in order to leverage the presence of a user-defined edge index. We use node properties to label nodes in post-order and another user-defined index to keep track of the edges of the spanning tree. Finally, the *diameter* of a graph is found using the index over all transitive closure edges by sorting it in descending order of weight (weighted graph) or path length (unweighted graph) and retrieving the first hit. In an alternative implementation, we have used Neo4j's query language Cypher that is available as an experimental feature.

[10]At the time of writing this, we are still investigating the precise nature of this trade-off. We note, however, that the *NetScience* data set performs better with the first approach, whereas both *Geom* and *Erdos* favor the second approach.

4. EXPERIMENTS AND RESULTS

All experiments were run on an Intel Core2 Quad CPU at 2.66GHz with 4GB of main memory and a 250GB SATA hard disk at 7200 RPM. Measurements were taken running Microsoft Windows 7 Enterprise (64bit). For both RDBMS and ODBMS, the 64bit editions of the latest product version were used. RDBMS is one of the major commercial products on the relational database market, as is ODBMS on the object database market. Neo4j (Version 1.4.1) is an open-source, purely Java-based library that was used in embedded mode. In the case of RDBMS, experiments were run from the Windows command-line using server-side traces to measure performance. In the case of ODBMS and Neo4j, experiments were run against their respective Java interfaces using the 64bit edition of the Oracle Java virtual machine (Version 1.7.0) in server mode. In all experiments, the main memory available to the database system was configured to 3GB, leaving 1GB for the operating system.

4.1 Data Loading

Table 3 reports total wall clock times for loading the NetScience, Geom, and Erdos data sets into RDBMS, ODBMS, and Neo4j. We measured five runs that consisted of creating a new database and loading the data. The numbers reported in the table are an average of three runs, after removing the slowest and fastest run. The time it takes to create the new database is not included in the measurement. It is noticeable that the loading times of RDBMS are one (or even two) orders of magnitude larger than the ones of ODBMS and Neo4j. We attribute this mostly to to constraint checking and index building, and to a lesser extent to the fact that the SQL scripts are more verbose and therefore much larger than the compact Pajek files used for ODBMS and Neo4j.

Time (ms)	NetScience	Geom	Erdos
RDBMS	54556	292200	411655
ODBMS	983	6687	5928
Neo4j	7717	9400	19245

Table 3: Total loading times

In Table 4 we report databases size for all systems and data sets combinations. The first line of each system gives the database size after loading, whereas the second line reports the size after the transitive closure (Q1) has been computed and persisted. The numbers for RDBMS are as expected and we will not discuss them further. In tuning the ODBMS implementation, we have increased the size of the database file that is pre-allocated, which explains the large sizes after loading. Since our ODBMS implementation stores a full `rowIndex` instance for each node, the database for the Geom data set is larger than the one for the Erdos data set, despite the sparsity of the Geom graph itself. The numbers of Neo4j are again as expected, but storing the transitive closure as graph edges of a special type together with their properties requires a comparatively large amount of storage space. Finally, we note that in our experimental setup the storage size of Erdos with transitive closure is large enough

to preclude the possibility of processing the graph (and intermediary data structures) in memory.

Size (MB)	NetScience	Geom	Erdos
RDBMS	3	4	3
	11	794	2783
ODBMS	490	494	494
	569	2325	2222
Neo4j	< 1	3	3
	17	1450	5295

Table 4: Database size after data loading and computing transitive closure

4.2 Query Workload

The results reported in this section have been measured as follows. For every database system and data set combination, we have executed five sequential runs that load the data and execute the queries in ascending order. A new database was created before each batch of five runs. At the end of each run, the database server was restarted to reset server-side caches. Each run was executed in its own process to reset client-side caches. Each task within a run, opens the database before executing and closes it at the end. Figure 4.2 presents an overview of all measured results in milliseconds. The bar graphs report average run times, after discarding the slowest and fastest run. The error bars report the standard error over *all* five runs and therefore give an indication of the variability of a system. We now discuss the results in detail.

The run times of Q1 reported in Figure 2(a) are based on the "naive" approach for RDBMS and the *in-memory* algorithm for ODBMS and Neo4j as described in Section 3. We also measured the *persistent* algorithms for ODBMS and Neo4j on NetScience which resulted in average run times of 4 minutes and several hours, respectively. Note that the faster run times of ODBMS and Neo4j depend on the fact that the intermediate data structures of Floyd-Warshall fit into main memory. Also, both implementations had to break up the query into several transactions in order not to run out of memory. In contrast, the RDMBS implementation runs in one transaction and scales beyond main memory. Nevertheless, we have chose to report these "best case" figures for ODBMS and Neo4j as they can be interpreted as a lower bound in case of optimal memory management.

Figure 2(b) reports the numbers for computing all node degrees, normalized by the number of nodes. The results indicate that ODBMS and Neo4j scale more gracefully than RDBMS. The reason for this is that computing the node degree is a very local operation that only needs to retrieve one node, whereas in the case of RDBMS it is a select and group by operation over the entire Edges view.

Run times for Q3 are shown in Figure 2(c), measured by testing 10,000 node pairs and normalizing by that number. All implementations answer this query based on the transitive closure. We note that ODBMS takes a big performance hit on the Geom data set since our `rowIndex` implementation cannot profit from the sparsity of that graph. On the fully connected Erdos data set, the run times of all three implementations are within the same order of magnitude.

Figure 2(d) summarizes the run times for Q4, measured by recursively reconstructing the shortest paths for 10,000 node pairs and normalizing by this number. For ODBMS, we again note the previously mentioned issue with our `rowIndex` design for sparse graphs. This query also demonstrates the potential performance hits that Neo4j takes if the processed data does not fit into memory. For every recursively accessed node, it maps all 6,927 edges of the transitive closure attached to that node into memory, even though the algorithm will only access one of these edges.

We measured the run times of Q5 by computing the degree centrality for all nodes and normalizing by the number of nodes. Since Q5 builds on Q2, we expected the results to be similar to Figure 2(b). As can be seen in Figure 2(e), this is only partially the case. While the Q5 run times of RDBMS and ODBMS are comparable to those of Q2, Neo4j's performance degrades due to the fact that the total number of nodes cannot be easily accessed, but must be queried as the size of the node index.

The run times of Q6 reported in Figure 2(f) were measured by computing the closeness centrality for 100 nodes and normalizing by that number. The good performance of RDBMS is due to the fact that the query result can be computed in a single aggregation over the global TransitiveClosure relation, while we need to load each node and iterate over its `rowIndex` or its edges of the transitive closure in ODBMS and Neo4j, respectively. As in Q4, we note the issue with the `rowIndex` design for ODBMS on Geom as well as the issue with memory management for Neo4j on Erdos.

The run times of Q7 in Figure 2(g) were measured by computing the betweenness centrality for 10 nodes and normalizing by that number. RDBMS implements our own algorithm based on the TransitiveClosure relation as described in Section 3.1, whereas ODBMS and Neo4j implement Brandes' algorithm. We can see that the scaling of RDBMS and Neo4j is governed by the sparsity of the graph and is roughly the same in terms of orders of magnitude. Since our ODBMS implementation does not profit from sparse graphs, it already takes a performance hit for Geom, but remains closer to RDBMS on Erdos due to better memory management.

The run times measured for Q8 are shown in Figure 2(h). Two results require closer examination. First, Neo4j takes a performance hit on graphs consisting of many components (NetScience and Geom). When constructing the spanning tree (or rather spanning forest), we use an ad-hoc node property to keep track of the nodes that are already included in the spanning forest. Since this property is not indexed by the node index, Neo4j requires a sequential scan to search for the root of the next spanning tree, i.e. a node without the ad-hoc property. Second, the performance of RDBMS can be explained by the fact that we had to use a cursor to implement the last phase of Tarjan's algorithm, which, as expected, is not very efficient.

Figure 2(i) reports the run times for Q9. RDBMS answers this query with a scan of the TransitiveClosure relation, whereas Neo4j uses an index over all edge weights of the transitive closure that is populated in Q1. The scaling of both implementations depends on the number of edges in the transitive closure. Our ODBMS implementation is less efficient and its scaling depends on the number of nodes, as it iterates over all nodes and `rowIndex`s to find the diameter.

5. DISCUSSION AND EXPERIENCES

In this section, we report experiences and lessons learned from implementing our benchmark with three different da-

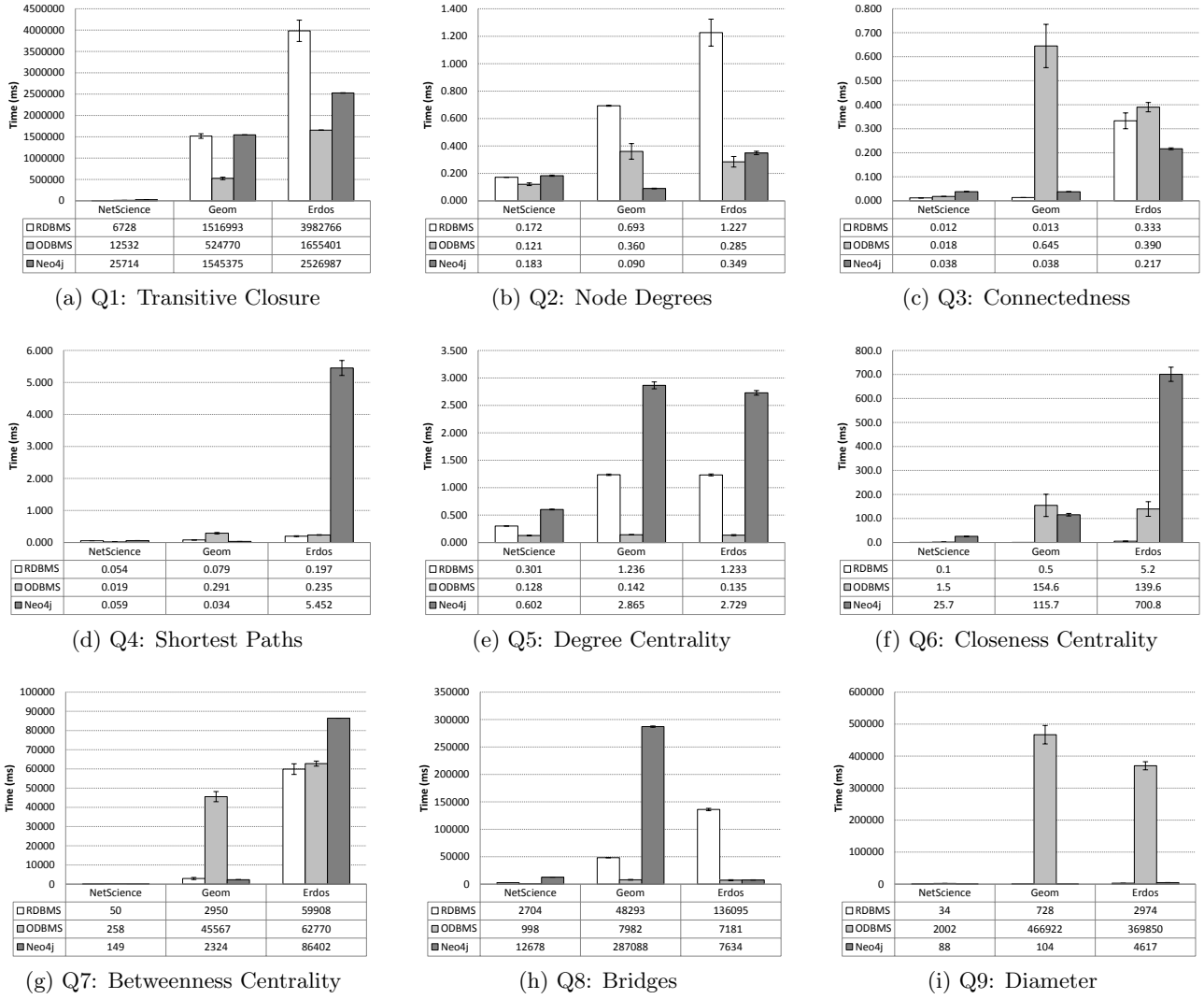


Figure 2: Experimental results for the RDBMS, ODBMS, and Neo4j implementations on the NetScience, Geom, and Erdos data sets.

tabases. We also offer best practices designed to inform the choice of a database system for a specific graph data application.

5.1 System-level Aspects

We begin by describing how the architecture and functionality of a database at the system-level impacts its performance with respect to a specific graph data application.

5.1.1 Memory Management

To process large graphs, efficient memory management is a key requirement and one of the major advantages of choosing a relational database system for graph data management. While we have faced several memory management challenges in our ODBMS and Neo4j implementations, this has never been an issue with RDBMS. Whereas ODBMS provides fine-grained control over how data is loaded into main memory, Neo4j's support in this respect is limited to setting the amount of memory that it uses for mapping database files into memory and to configuring the type of cache (hard, soft, or weak) for node and edge objects. In our experience, Neo4j tends to load more data than required which is especially problematic when traversing edges in a fully connected graph. In this case, all edges incident to a node (regardless of their edge type) seem to be brought into memory. Therefore, Neo4j's traversal framework was rendered useless in the presence of transitive closure edges and we encountered queries (Q4 and Q6) where thrashing could not be avoided. However, since Neo4j claims to be (mostly) an in-memory graph database, we also note that it achieves very good performance figures if this condition is met.

5.1.2 Consistency and Transactions

Since both ODBMS and Neo4j rely on the Java garbage collector to evict objects from main memory, we were forced to split memory-intensive computations (e.g. Q1 or Q7) into multiple transactions to release the handles to no longer required objects. This approach not only introduces the maximum transaction size as another parameter that is potentially difficult to configure, it also means the partial re-

sults of a computation become accessible to other concurrent transactions. In the case of Q1, this implies that even intermediate results, which are most likely wrong due to the iterative nature of Floyd-Warshall, could be read by other transactions that use the transitive closure. The use of such a database is therefore only advisable in single-user scenarios where concurrency is easily managed outside the database.

5.1.3 Execution Strategies

An architectural difference between RDBMS and the other two systems is that RDBMS has total knowledge of both the data and the queries since they are managed as SQL stored procedures in the database. In contrast, ODBMS and Neo4j have total knowledge of the data, but no knowledge of the algorithm used to evaluate the query since it is executed on the client. As a consequence, RDBMS can optimize the entire query, while ODBMS and Neo4j operate on a "one library call at the time" basis. In our experience, this architectural difference has worked to the advantage of RDBMS for complex queries that process a lot of data, i.e. queries with scope graph or radius global. Then again, simple queries with scope node and radius local or path seem to be more efficient on ODBMS and Neo4j. The lack of a declarative query language and optimizer also implies that it is up to the developer to implement the best possible query plan. As we have shown, it is unfortunately often the case that an implementation strategy that performs good on one graph is suboptimal on another. In ODBMS and Neo4j, we were forced to address this issue by implementing alternative strategies and selecting the best for each data set. Normally, it is precisely this type of access path selection that one would expect from a database system.

5.1.4 Tuning

First, we note that the numbers reported for RDBMS in the previous section are all "out of the box" performance numbers as no tuning was applied. The same cannot be said for ODBMS and Neo4j that both involved extensive tuning. In the case of ODBMS, we adjusted the default database size to speed up data loading. We further tuned the batch size for persistent collections (cf. Section 3.2) according to the scope and radius of the query. For queries with scope node or path and radius local or path, we chose a batch size of one object, whereas all other queries had the batch size that was maximally possible with the available memory. Finally, we experimented with the maximum transaction size for memory-intensive operations. Small transaction sizes free up memory quickly, but imply overhead. Given that the node size is correlated to the number of nodes (since we create one EdgeProperty object per node in the network), the ideal transaction granularity depends on a network's node size and has to be configured accordingly. In Neo4j, we tuned the sizes of memory-mapped database file regions and object cache types. Again, our decisions were guided by the scope and radius of queries. For graph/global and node/global queries we mapped as much of the graph as possible into memory. For queries with scope path or radius path, we prioritized the mapping of the database file storing the relationships over the one storing nodes to favor traversals. The cache type for queries with radius global was set to "weak", all other queries were executed with cache type "soft". The former favors high throughput applications, while the latter is suitable for traversals.

5.2 User-Level Aspects

We continue by discussing aspects at the user-level that factor into the application-specific performance of a system.

5.2.1 Data Modeling

Since none of the database systems evaluated in this work supports the general graph data model defined in this paper, a mapping had to be defined in all three cases. Arguably, the closest match is Neo4j as it is a dedicated graph database. However, its inability to manage undirected edges and its schema-less approach open up the possibility of alternative physical graph representations. These options and limitations need to be carefully evaluated and there might not be one single approach that supports all use cases equally well. The same is true for the even richer data model supported by ODBMS that provides countless opportunities for fine-grained optimizations or ample room for error, depending on the skills of the developer. Again, a data model that works well for one task might not work at all for another, as evidenced by the `rowIndex` that was designed to optimize Q1 but backfired on Q9. The simplicity of the relational model guarantees a straightforward mapping of graphs and therefore the risk of inefficient designs is minimized.

5.2.2 Programmability

Being able to efficiently express graph algorithms in the programming language of the database system is another important user-level aspect. In ODBMS and Neo4j, algorithms are primarily expressed in Java with some functionality implemented as queries. As a consequence, the expressiveness of these systems is guaranteed to be sufficient and algorithms are easily developed, by starting from a text book in-memory algorithm and augmenting it with calls to the database. In RDBMS, all algorithms had to be expressed in SQL which is more challenging. With the exception of Tarjan's algorithm, it was however possible to do so in a fully relational way, i.e. without resorting to control structures such as *if-then-else* and explicit *loop* statements. In our experience, a naive algorithm using only relational operators will often outperform a more clever implementation using control structures. Resorting to control structures often means that the database is no longer able to fully optimize and automatically parallelize queries over multiple cores. In our experiments, this is evidenced by the performance drop of Q8.

5.3 Summary

Generally, our experiments and experiences indicate that RDBMS provides solid average performance and leaves little margin for error to inexperienced users. Automatic memory management and query optimization are valuable assets for graph/global type queries. ODBMS and Neo4j are more difficult to master, but can outperform RDBMS when expertly tailored to a specific use case or under the assumption of special circumstances.

6. RELATED WORK

Our work is motivated by the increasing number of applications that require graph data management. The same trend is identified by Domníguez-Sal *et al.* [12], who identify criteria for the design of a graph database benchmark. Our proposal follows their guidelines and we agree with their observation that social network metrics constitute a viable basis for a representative graph database benchmark.

However, our proposal differs from theirs in several aspects. First, we do not limit ourselves to dedicated graph databases, but also include existing database system as possible solutions for graph data management and processing. Second, our proposed query workload is smaller and therefore more easily implemented, yet still representative. Finally, we have validated our proposal by implementing it using three very different databases. We also report performance figures, experiences and best practices.

The HPC Scalable Graph Analysis Benchmark [3] is a general benchmark to evaluate graph analysis systems. In contrast to our proposal, the benchmark only considers directed weighted graphs. Further, it defines a workload of only four tasks (loading, aggregation, traversal, and betweenness centrality). We believe this limited set of tasks is not sufficient to reliably characterize applications and predict their performance with respect to a specific candidate database. Ruflin *et al.* [22] have evaluated several NoSQL databases and MySQL in terms of their suitability as storage and processing systems for social data. In contrast to our work, they do not consider analytical applications, but evaluate these systems as platforms for the operation of Web 2.0 sites.

The LUBM [17] benchmark defines a SPARQL query workload to evaluate the performance of RDF data stores. While LUBM constitutes a benchmark for graph data management and processing, it is geared towards one specific application of graphs and therefore not general. Furthermore, the specification of queries at the level of SPARQL precludes the possibility of evaluating different types of databases that might not support this language. Due to their related data model, benchmarks proposed for XML and object databases could also be considered in this context. Not unlike graph data applications, XML data applications can vary considerably in terms of requirements and characteristics. XBench [33] defines a family of benchmarks designed to address this fact by targeting both data-centric and text-centric as well as single and multiple document applications. OO7 [9] is a well-know benchmark for object databases. It was recently ported to the Java programming language in the wake of the "second coming" of object databases [28]. We note that these benchmarks have been proposed with automated business application in mind. Therefore, they do not adequately represent the analytical workload typical of today's graph data management and processing applications.

7. CONCLUSIONS

In this paper, we acknowledge the increasing demand for graph data management and processing systems by proposing a benchmark to evaluate candidate solutions with respect to specific applications. We define and classify a workload of nine query that together cover a wide variety of graph data use cases. We characterize applications in terms of a subset of queries that can be used to gauge the application-specific performance of a system. In contrast to other proposals, we do not limit ourselves to one specific type of database, but rather put the emphasis on the capabilities and functionality required by graph data applications.

We demonstrated the feasibility of our approach by implementing the benchmark in three very different database systems. Apart from discussing performance figures that we measured based on these implementations, we also reported experiences and best practices from a system-level and user-level perspective. In particular, we showed how our query classification in terms of scope and radius can drive the tuning of a database system. We believe that our experiences further motivate the need for graph data benchmarks such as this as they document the different behavior of different database systems for different graph applications.

8. REFERENCES

[1] R. Agrawal and H. V. Jadadish. Direct Algorithms for Computing the Transitive Closure of Database Relations. In *Proc. VLDB*, pages 255–266, 1987.

[2] R. Angles and C. Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40:1:1–1:39, 2008.

[3] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, T. Meuse, and E. Robinson. HPC Scalable Graph Analysis Benchmark. `http://www.graphanalysis.org/benchmark/GraphAnalysisBenchmark-v1.0.pdf`, 2009.

[4] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, pages 165–178. Springer Verlag, 1986.

[5] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[6] D. Barbosa, G. Miklau, and C. Yu, editors. *Proc. Workshop on Databases and Social Networks (DBSocial)*, 2011.

[7] V. Batagelj and A. Mrvar. Pajek Data Sets. `http://pajek.imfm.si/doku.php?id=data:index`, 2003.

[8] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[9] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. SIGMOD*, pages 12–21, 1994.

[10] W. de Nooy, A. Mrvar, and V. Batagelj. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2nd edition, 2011.

[11] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[12] D. Domínguez-Sal, N. Martínez-Bazan, V. Muntés-Mulero, P. Baleta, and J.-L. Larriba-Pey. A Discussion on the Design of Graph Database Benchmarks. In *TPC Technology Conference on Performance Evaluation and Benchmarking*, 2010.

[13] P. Erdős and A. Rényi. On Random Graphs, I. *Publicationes Mathematicae*, 6:290–297, 1959.

[14] R. W. Floyd. Algorithm 97: Shortest Path. *CACM*, 5(6):345, 1962.

[15] C. Goffman. And What Is Your Erdős Number? *The American Mathematical Monthly*, 76(7):791, 1969.

[16] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 2nd edition, 1993.

[17] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics*, 3(2):158–182, 2005.

[18] Y. E. Ioannidis. On the Computation of the Transitive Closure of Relational Operators. In *Proc. VLDB*,

pages 403–411, 1986.
[19] J. H. Kim and V. H. Vu. Generating Random Regular Graphs. In *Proc. STOC*, pages 213–222, 2003.
[20] M. E. Newman. Finding Community Structure in Networks Using the Eigenvectors of Matrices. *Phys. Rev. E*, 74(3):036104, 2006.
[21] Objectivity, Inc. Building a Comprehensive Space Object Catalog for Real Time Tracking. `http://www.objectivity.com/pages/downloads/whitepaper/pdf/mitre.pdf`, 2008.
[22] N. Ruflin, H. Burkhart, and S. Rizzotti. Social-Data Storage-Systems. In *Proc. DBSocial*, pages 7–12, 2011.
[23] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The Case for Application-Specific Benchmarking. In *Proc. HotOS*, pages 102–107, 1999.
[24] S. S. Skiena. *The Algorithm Design Manual*. Springer Verlag, 2nd edition, 2008.
[25] A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The SDSS SkyServer: Public Access to the Sloan Digital Sky Server Data. In *Proc. SIGMOD*, pages 570–581, 2002.
[26] R. E. Tarjan. A Note on Finding the Bridges of a Graph. *Information Processing Letters*, 2(6):160–161, 1974.
[27] Transaction Processing Performance Council. `http://www.tpc.org/`, 2011.
[28] P. van Zyl. *Performance Investigation into Selected Object Persistence Stores*. PhD thesis, University of Pretoria, 2010.
[29] Versant Corp. Data from Outer Space: European Space Agency Case Study. `http://www.versant.com/pdf/cs-herschel-telescope.pdf`, 2011.
[30] S. Warshall. A Theorem on Boolean Matrices. *JACM*, 9(1):11–12, 1962.
[31] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.
[32] D. J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 2003.
[33] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. ICDE*, pages 621–632, 2004.

APPENDIX

A. SOFTWARE AND TOOLS

As part of the proposed benchmark, we have developed a series of tools designed to facilitate the implementation of our benchmark in databases other than the one presented in this paper. We plan to make this software freely available to any interested parties.

A.1 Pajek Parser and Serializer

As presented in Section 2.3, many graphs from the domain of (social) network analysis are available in the Pajek [10] data format. Pajek is a simple text-based format that first lists all nodes with their properties, followed by the definition of directed or undirected edges, respectively referred to as arcs and edges. Each weighted arc or edge is defined on a single line using the format "`source target weight`". Unweighted arcs and edges are defined using arc or edge lists, where every line has the format "`source { target }`". Finally, Pajek also supports the definition of arcs and edges using adjacency matrices.

Our parser is compliant with the Pajek format specification that defines the structure of a graph. It does, however, not support the parsing of Pajek layout metadata, e.g. node positions, colors, etc. The parser follows an event-based model and provides three call-back methods that are implemented by a so-called *graph builder* to handle the creation of nodes, arcs, and edges. The goal of the Pajek serializer is to write Pajek files from graphs that are managed in memory or in a database. The serializer offers methods to write out entire graphs or single nodes, arcs, and edges. Apart from transferring graphs between databases, we also make use of the Pajek serializer in the context of the graph data generator framework discussed below.

A.2 Graph Data Set Generators

When implementing our proposed benchmark for a new database, it is often useful to work with synthetic data sets that comply with certain cardinalities or graph characteristics. Apart from testing the correctness of the implementation, experiments with these data sets can also provide valuable information as to how different graph characteristics affect the performance of certain tasks in a specific database. To better support new implementations of our proposed benchmark, we therefore provide a number of synthetic graph data set generators.

The *average node degree* graph generator creates directed or undirected graphs of a specific average node degree. Given the number of nodes n and the average node degree $\bar{d}$, the generator creates a random graph with $\lceil (n \cdot \bar{d})/2 \rceil$ edges. The *random regular graph* generator creates a uniform graph where all nodes have a degree of exactly d. To generate this graph, we use the algorithm presented by Kim and Vu [19].

The *linear* graph generator creates a path graph, i.e. a tree with two or more vertices that is not branched at all. All nodes in the graph are either of degree 1 or 2. Linear graphs can be directed or undirected and weighted or unweighted. The *ring* graph generator creates a fully connected graph where each node has a degree of 2. It is a special case of the ring lattice generator. The *ring lattice* graph generator augments a ring graph with additional edges in order for all nodes to comply to a given node degree d. Edges are inserted by connecting a node to its $d-2$ "successors" on the ring. The *star* graph generator creates graphs with a central node that is connected to all other nodes.

The *Barabási-Albert* [5] graph generator creates a directed or undirected graph by looping a defined number of steps and adding a new node in every step, starting from an unconnected graph of "seed" nodes. The new node created in one step of the loop is connected to an existing node v with a probability of $p = (deg(v) + 1)/(|E| + |V|)$. The *Erdős-Rényi* [13] graph generator creates a random graph that can either be directed or undirected and weighted or unweighted, where any two nodes are connected with a given probability p. The *Watts-Strogatz* [32] graph generator creates a graph by starting with a one-dimensional ring lattice in which each node has k neighbors. With probability p, it then randomly rewires the edges in order to create small world networks for certain values of p and k that exhibit low characteristic path lengths and a high clustering coefficient.