An Evaluation Methodology and Experimental Comparison of Graph Databases

Matteo Lissandrini
University of Trento
ml@disi.unitn.eu

Martin Brugnara
University of Trento
mb@disi.unitn.eu

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

We are witnessing an increasing interest in graph data. The need for efficient and effective storage and querying of such data has led the development of graph databases. Graph databases represent a relatively new technology, and their requirements and specifications are not yet fully understood by everyone. As such, high heterogeneity can be observed in the functionalities and performances of these systems. In this work we provide a comprehensive study of the existing systems in order to understand their capabilities and limitations. Previous similar efforts have fallen short in providing a complete evaluation of graph databases, and drawing a clear picture on how they compare to each other. We introduce a micro-benchmarking framework for the assessment of the functionalities of the existing systems and provide detailed insights on their performance. We support the broader spectrum of test queries and conduct the evaluations on both synthetic and real data at scales much higher than what has been done so far. We offer a systematic evaluation framework that we have materialized into an evaluation suite. The framework is extensible, allowing the easy inclusion in the evaluation of other datasets, systems or queries.

1. INTRODUCTION

Graph data [61] has become increasingly important nowadays since it can model a wide range of applications, including transportation networks, knowledge graphs [44, 58], and social networks [35]. As the graph datasets are becoming larger and larger, so does the need for their efficient and effective management, analysis, and exploitation. This has led to the development of graph data management systems.

There are two kinds of graph data management systems (Figure 1). One is the graph processing systems [27, 33, 37, 45, 46, 47]. They are systems that analyze graphs with the goal of discovering characteristic properties in their structures, e.g., average degree of connectivity, density, or modularity. They also perform batch analytics at large-scale that implement a number of computationally expensive graph algorithms like PageRank [54], SVD [30], strongly connected component identification [59], core identification [28], and others. Examples in this category include systems like Giraph, GraphLab, Graph Engine, and GraphX [60]. The second kind of graph management systems comprises the so-called graph databases, or GDB for short [20]. Graph Databases focus on storage and querying tasks where the priority is the high-throughput interrogations of the data, and the execution of transactional operations. Originally, they were implemented by exploiting specialized schemas on relational systems. As the sizes of the graphs was becoming larger and more complex, it became apparent that more dedicated systems were needed. This gave rise to a whole new wave of graph databases, that include Neo4j [11], OrientDB [13], Sparksee [14] (formerly known as DEX), Titan [16], and the more recent, ArangoDB [6] and BlazeGraph [15]. The focus of this work is on this second kind of graph management systems, i.e., the graph databases.

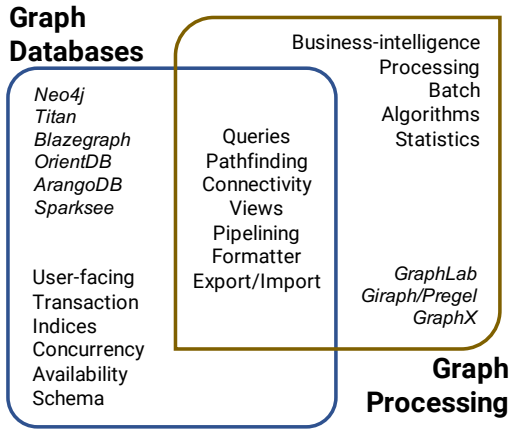

Figure 1: Overview of the distinction between Graph Databases and Graph Processing systems

Given the increased popularity that graph databases are enjoying, there is a need for comparative evaluations of their available options. Such evaluations are critically important for practitioners in order to better understand both the capabilities and limitations of each system, as well as the conditions under which perform well, so that they can choose the system that better fits the task at hand. A comparative study is also important for researchers, since they can find where they should invest their future studies. Last but not least, it is of great value for the developers, since it gives them an idea of how graph data management systems com-

Technical Report, Department of Information Engineering and Computer Science - DISI
University of Trento.

pare to the competitors and what parts of their systems need improvement. There is already a number of experimental comparisons on graph databases [29,40,41], but they do not provide the kind of complete and exhaustive study needed. They test a limited number of features (i.e., queries), which provide only a partial understanding of each system. Furthermore, existing studies do not perform experiments at large scale, but make prediction on how the systems will perform based on tests on smaller sizes. Apart from the fact that they tend to provide contradictory conclusions, when we performed the experiments at larger scale, results were highly different from those they had predicted. Finally, many of the tests performed are either too simplistic or too generic, to a point that it is not easy to interpret the results and identify the exact limitations of each system.

Given the above motivations, in this work we provide a complete and systematic evaluation of the state-of-the-art graph database systems. Our approach is based not only on the previous works of this kind, but also on the principles that are followed when designing benchmarks [17, 29, 39, 40, 41]. Based on an extensive study of the literature, we have made an effort to cover all the scenarios that have so far been identified. As result, we scrupulously test all the types of insert-select-update-delete queries considered so far, with special attention to the various use-cases, and extend such tests to cover the whole spectrum of tasks, data-types and scale. As an indication of the extent of our work, we test 35 classes of operations with single queries and batch workloads as well (for a total of about 70 different tests) as opposed to 4-13 that existing studies have done, and we scale our experiments up to 28M nodes/ 31M edges, as opposed to the 250K nodes/2.2M edges of existing works. Finally, in the design of the tests, we follow a microbenchmark model [25]. Instead of considering elaborate situations, we have identified primitive operators and we designed tests that provide a clear understanding of each such elementary operator. Complex tasks can be typically decomposed into combinations of basic steps, thus, the performance of more involved tests can be inferred by that of the primitive components. In addition, basic operators are often implemented by opaque components in the system, therefore, by identifying the underperformed operators it is easy to determine system components with limited performance.

The specific contributions of this work are the following: **(i)** We explain the limitations of existing graph database evaluations, and clarify the motives of the current evaluation study (Section 2); **(ii)** We describe the model of a graph database and present the most well-known such systems, both old and new, the features that each one provides, and highlight the implementation choices that characterize them (Section 3); **(iii)** Based on a systematic study of the existing literature, we provide an extensive list of fundamental primitive operations (queries) that graph databases should support (Section 4); **(iv)** We introduce an exhaustive experimental evaluation methodology for graph databases, driven by the micro-benchmarking model. The methodology consists of queries identified previously and a number of synthetic and real-world datasets at different scales, complexity, distribution and other characteristics (Section 5). For fairness across systems, the methodology adopts a standard application interface, the Gremlin query language, which allows the testing of each system using the same set of operations; **(v)** We materialize the methodology into a testing suite based on software containers, which is able to automate the installation and investigation of different graph databases. The suite allows for future extensions with additional tests, and is available online as open source, alongside a number of datasets; **(vi)** We apply this methodology on the state-of-the-art graph databases that are available today, and study the effect that different real and synthetic datasets, from co-citation, biological, knowledge base, and social network domains has on different queries (Section 6), along with a report on our experience with the set-up, loading, and testing of each system. Note that we focus on single machine installations, even though some systems may support clusters, since single-machine installations are still highly popular [36]. Our goal was, as a first step, to understand how the graph databases perform in a single-machine installation. The question about which system is able to scale-out better, may only come after the understanding of its inherent performance [51, 57]. Multi-machine exploitation is our next step that would naturally complement the current work.

2. EXISTING EVALUATIONS

Since we focus on the existing evaluation of graph databases and not of graph processing systems [27, 37, 46], we do not elaborate further on the latter. For graph databases there are studies, however most of them are incomplete or have become out-dated. In particular, one of the earliest works [20] surveys graph databases in terms of their internal representation and modeling choices. It compares their different data-structures, formats and query languages, but provides no empirical evidence of their effectiveness. Another work [18] compares 9 different systems and distinguishes them into graph databases and graph stores based on their general features, data modeling capabilities and support for specific graph query constructs. Unfortunately, not even this work provides any experimental comparison, and like the previous one, it includes systems that have either evolved considerably since then, or have been discontinued.

A different group of studies [29, 40, 41] has focused on the empirical comparison of the performance of the systems, but even these studies are limited in terms of completeness, consistency, and currency of the results. The first of such works [29] analyzes only 4 systems, two of which are no longer supported. Its experiments are limited both in dataset size as well as in number and type of operations performed. The systems were tested on graphs with at most 1 million nodes, and the operations supported were limited to edge and node insertion, edge-set search based on weights, subgraph search based on 3-hops BFS, and the computation of betweenness centrality. Update operations, graph pattern and path search queries are missing, alongside many scalability tests. A few years later, two empirical works [40, 41] compared almost the same set of graph databases over datasets of comparable small sizes, but agree only partially on the concluded results. In particular, the systems analyzed in the first study [40] were DEX[1], Neo4j, Titan, and OrientDB, while the second study [41] considered also Infinite Graph. The results have shown that for batch insertion DEX[1] is generally the most efficient system, unless properties are attached to elements, in which case

[1]DEX is the old name for the Sparksee system

Neo4j is the fastest one [41]. For traversal with breadth-first search, both works agree that Neo4j is the most efficient. Nonetheless, the second work claims, but without proving it, that DEX[1] would outperform Neo4j on bigger and denser graphs [41]. In the case of computing unweighted shortest paths between two nodes, Neo4j performs best in both studies, but while Titan ends up being the slowest in the first study [40], it is one of the fastest in the second [41]. For node-search queries, the first work [40] shows that both DEX[1] and OrientDB are the best systems when the selection is based on node identifiers, while the other [41], which implements the search based on a generic attribute, shows Neo4j as the winner. Finally, on update operations, the two experimental comparisons present contradicting results, showing in one study favorable results for DEX[1] and OrientDB, while in the other for Neo4j. Due to these differences, these studies have failed to deliver a consistent picture of the available systems, and also provide no easy way of extending them with additional tests and systems.

The benchmarks proposed in the literature to test the performance of graph databases are also of high significance [19, 21, 32]. Benchmarks typically come with tools to automatically generate synthetic data, sampling techniques to be used on real data, and query workloads that are designed to pinpoint bottlenecks and shortcomings in the various implementations. These existing benchmarks are domain specific, i.e., RDF focused [19, 21] or social network focused [32], but despite the fact that we do not use any of them directly, the design principles and the datasets upon which they have been built have highly influenced our work.

3. GRAPH DATABASES

3.1 Data Model

Graph data are data consisting of nodes (also called vertices) and connections between them called edges. There are various types of graphs depending on the kind of annotations one assumes. In this work we consider generic graphs where every edge has a label and every node or edge has a set of attributes that describes its characteristic properties.

Formally, we axiomatically assume the existence of an infinite set of names $\mathcal{N}$ and an infinite set of values $\mathcal{A}$. A *property* is an element in the set $\mathcal{N}\times\mathcal{A}$ of name-value pairs.

A *graph* is then a tuple $G=\langle V, E, l, p\rangle$ where V is a set of nodes, E is a set of edges between them, i.e., $E\subseteq V\times V$, $l{:}E\to\mathcal{N}$ is an edge labeling function, and $p{:}\{V\cup E\}\to 2^{\mathcal{N}\times\mathcal{A}}$ is a property assignment function on edges and nodes.

Note that the above model allows different nodes to have exactly the same properties, and different edges to have exactly the same label and set of properties. To be able to distinguish the different nodes or edges, systems extend the implementation of the above model by means of unique identifiers. In particular, they consider a countable set $\mathcal{O}$ of unique values and a function $id{:}\{N\cup E\}\to\mathcal{O}$ that assigns to each node and edge a unique value as its identifier. Furthermore, the nodes and edges, as fundamental building blocks of graph data, are typically implemented as atomic *objects* in the systems and are referred to as such.

Figure 2 illustrates a portion of graph data. The annotations containing the colon symbol " : " are the properties, while the others are the labels. The number on each node indicates its identifier. For presentation reasons we have omitted the ids on the edges.

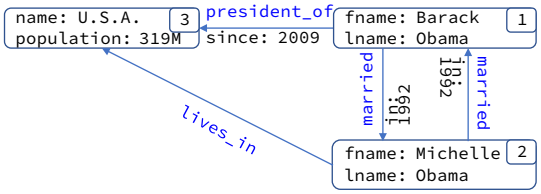

Figure 2: A portion of graph data

3.2 Systems

For a fair comparison we need all systems to support a common access method. Tinkerpop [5], an open source, vendor-agnostic, graph computing framework, is currently the only common interface in most graph databases. TinkerPop-enabled system are able to process a common query language: the Gremlin language. Thus, we chose systems that support some version of it through officially recognized implementations. Furthermore, we consider systems with a licence that permits the publication of experimental comparisons, and also those that were made available to us to run on our server without any fee. Table 1 summarizes the characteristics of the systems we consider in our study. Among others, we show the query languages that these systems support (other than Gremlin). We would've also included GraphDB [12] and InfiniteGraph [10], but licensing issues of the first did not allow us to publish any performance verification results, while the second has been discontinued.

3.2.1 ArangoDB.

ArangoDB [6] is a multi-model database. This means that it can work as a document store, a key/value store and a graph database, all at the same time. With this model, objects like nodes, edges or documents, are treated the same and stored into special structures called collections. Apart from Gremlin, it supports its own query language, called AQL, *ArangoDB Query Language*, which is an SQL like dialect that supports multiple data models with single document operations, graph traversals, joins, and transactions. The core, which is open-source (Apache License 2.0), is written in C++, and is integrated with the V8 JavaScript Engine (`github.com/v8/v8`). That means that it can run user-defined JavaScript code, which will be compiled to native code by V8 on the fly, while AQL primitives are written in C++ and will be executed as such. Nonetheless, the supported way of interacting with the database system is via REST API and HTTP calls, meaning that there is no direct way to embed the server within an application, and that every query will go through a TCP connection.

It supports ACID transactions by storing data modification operations in a write-ahead log, which is a sequence of append-only files containing every write operations executed on the server. While ArangoDB automatically indexes some system attributes (i.e., internal node identifiers), users can also create additional custom indexes. As a consequence, every collection (documents, nodes or edges) has a default primary index, which is an unsorted hash index on object identifiers, and, as such, it can be used neither for non-equality range queries nor for sorting. Furthermore, there exists a default edge index providing for every edge quick access to its source and destination. ArangoDB can serve

Table 1: Features and Characteristics of the tested systems

System	Type	Storage	Edge Traversal	Gremlin	Query Execution	Access	Languages
ArangoDB (2.8)	Hybrid (Document)	Serialized JSON	Hash Index	v2.6	AQL, Non-optimized	REST (V8 Server)	AQL, Javascript
BlazeGraph (2.1.4)	Hybrid (RDF)	RDF statements	B+Tree	v3.2	Programming API, Non-optimized	embedded, REST	Java, SPARQL
Neo4J (1.9, 3.0)	Native	Linked Fixed-Size records	Direct Pointer	v2.6 / v3.2	Programming API, Non-optimized	embedded, WebSocket, REST	Java, Cypher,
OrientDB (2.2)	Native	Linked Records	2-hop Pointer	v2.6	Mixed, Mixed	embedded, WebSocket,REST	Java, SQL-like
Sparksee (5.1)	Native	Indexed Bitmaps	B+Tree/Bitmap	v2.6	Programming API, Non-optimized	embedded	Java, C++,Python, .NET
SQLG (1.2) / Postgres (9.6)	Hybrid (Relational)	Tables	Table Join	v3.2	SQL, Optimized(*)	embedded (JDBC)	Java
Titan (0.5, 1.0)	Hybrid (Columnar)	Vertex-Indexed Adjacency List	Row-Key Index	v2.6 / v3.0	Programming API, Optimized	embedded, REST	Java

multiple requests in parallel and supports horizontal scale-out with a cluster deployment using Apache Mesos [4].

3.2.2 *BlazeGraph.*

Blazegraph [15] is open-source and available under GPLv2 or under a commercial licence. It is an RDF-oriented graph database entirely written in Java. Other than Gremlin, it supports SPARQL 1.1, storage and querying of reified statements, and graph analytics.

Storage is provided through a journal file with support for index management against a single backing store, which scales up to $50B$ triples or quads on a single machine. Full text indexing and search facility are built using a key-range partitioned distributed B+Tree architecture. The database can also be deployed in different modes of replication or distribution. One of them is the federated option that implements a scale-out architecture, using dynamically partitioned indexes to distribute the data across a cluster. While updates on the journal and the replication cluster are ACID, updates on the federation are *shard-wise ACID*. Blazegraph uses Multi-Version Concurrency Control (MVCC) for transactions. Transactions are validated upon commit using a unique timestamp for each commit point and transaction. If there is a write-write conflict the transaction aborts. It can operate as an embedded database, or in a client-server architecture using a REST API and a SPARQL end-point.

3.2.3 *Neo4J.*

Neo4j [11] is a database system implemented in Java and distributed under both an open source and commercial licence. It provides its own unique language called Cypher, and supports also Gremlin, and native Java API. It employs a custom disk-based native storage engine where nodes, relationships, and properties are stored separately on disk. Dynamic pointer compression expands the available address space as needed, allowing the storage of graphs of any size in its latest version. Full ACID transactions are supported through a write-ahead log. A lock manager applies locks on the database objects that are altered during the transaction.

Neo4j has in place a mechanism for fast access to nodes and edges that is based on IDs. The IDs are basically offsets in one of the store files. Hence, upon the deletion of nodes, the IDs can be reclaimed for new objects. It also supports *schema indexes* on nodes, labels and property values. Finally, it supports full text indexes that are enabled by an external indexing engine (Apache Lucene [3]), which also allows nodes and edges to be viewed and indexed as "key:value" pairs. Other Neo4J features include replication modes and federation for high-availability scenarios, causal cluster, block device support, and compiled runtime.

3.2.4 *OrientDB.*

OrientDB [13] is a multi-model database, supporting graph, document, key/value, and object data models. It is written in Java and is available under the Apache licence or a Commercial licence. Its multi-model features Object-Oriented concepts with a distinction for classes, documents, document fields, and links. For graph data, a node is a document, and an edge is mapped to a link. Various approaches are provided for interacting with OrientDB, from the native Java API (both document-oriented and graph-oriented), to Gremlin, and extended SQL, which is a SQL-like query language.

OrientDB features 3 storage types: (i) *plocal*, which is a persistent disk-based storage accessed by the same JVM process that runs the queries; (ii) *remote*, which is a network access to a remote storage; and (iii) *memory-based*, where all data is stored into main memory. The disk based storage (also called Paginated Local Storage) uses a page model and a disk cache. The main components on disk are files called *clusters*. A cluster is a logical portion of disk space where OrientDB stores record data, and each cluster is split into pages, so that each operation is atomic at page level. As we will discuss later (Section 6), the peculiar implementation of this system provides a good performance boost but poses an important limitation to the storing of edge labels.

OrientDB supports ACID transactions through a write ahead log and a *Multiversion Concurrency Control* system where the system keeps the transactions on the client RAM. This means that the size of a transaction is bounded by the JVM available memory. OrientDB also implements SB−Tree indexes (based on B-Trees), hash indexes, and Lucene full text indexes. The system can be deployed with a client-server architecture in a multi-master distributed cluster.

Sqlg/Postgresql. Sqlg [48] is an implementation of Apache TinkerPop on a relational DBMS. Postgresql [55] is one among those RDBMS supported, and the one we chose for our experiments. Sqlg provides Java API to the gremlin language, and the underlying implementation maps graph semantics to that of the RDBMS. It is possible to also send standard SQL queries directly to the back-end relational database, although this is not often convenient. For graph data, a vertex label is modeled by a table, containing all vertices with that label, and all the vertex's properties. An edge label is modeled as a many-to-many join-table between the vertices, Similarly to vertices, edge labels are mapped to tables. containing the vertex ID of the two edge endpoints alongside the edge properties. Indexes, transactions and parallelization are inherited from the underlying database system.

3.2.5 *Sparksee.*

Sparksee [14, 49], formerly known as DEX [50], is a commercial system written in C++ optimized for out-of-core operations. It provides a native API for Java, C++, Python and .NET platforms, but it does not implement any other query language apart from Gremlin.

It is specifically designed for labeled and attributed multigraphs. Each vertex and each edge are distinguished by permanent object identifiers. The graph is then split into multiple lists of pairs and the storage of both the structure

and the data is partitioned into different clusters of bitmaps for a compact representation. This data organization allows for more than 100 billion vertices and edges to be handled by a single machine. Bitmap clusters are stored in sorted tree structures that are paired with binary logic operations to speedup insertion, removal, and search operations.

Sparksee supports ACID transaction with a *N-readers* and *1−writer* model, enabling multiple read transactions with each write transaction being executed exclusively. Both search and unique indexes are supported for node and edge attributes. In addition a specific neighbor index can also be defined to improve certain traversal operations. Finally, Sparksee provides horizontal scaling, enabling several slave databases to work as replicas of a single master instance.

3.2.6 Titan.

Titan [16] is available under the Apache 2 license. The main part of the system handles data modeling, and query execution, while the data-persistence is handled by a third-party storage and indexing engine to be plugged into it. For storage, it can support an in-memory storage engine (not intended for production use), Cassandra [1], HBase [2], and BerkeleyDB [7]. To store the graph data, Titan adopts the adjacency list format, where each vertex is stored alongside the list of incident edges. In addition, each vertex property is an entry in the vertex record. Titan adopts Gremlin as its only query language, and Java as the only compatible programming interface. No ACID transactions are supported in general, but are left to the storage layer that is used. Among the three available storage backends only Berkeley DB supports them. Cassandra and HBase provide no serializable isolation, and no multi-row atomic writes.

Titan supports two types of indexes: *graph centric* and *vertex centric*. A graph index is a global structure over the entire graph that facilitates efficient retrieval of vertices or edges by their properties. It supports equality, range, and full-text search. A Vertex-centric index, on the other hand, is local to each specific vertex, and is created based on the label of the adjacent edges and on the properties of the vertex. It is used to speed up edge traversal and filtering, and supports only equality and range search. For more complex indexing external engines like Apache Lucene or ElasticSearch [9] can be used. Due to the ability of Cassandra and HBase to work on a cluster, Titan can also support the same level of parallelization in storage and processing.

3.3 Architectures and Query Processing

There are two ways to implement a graph database. One is to build it from scratch (*native* databases) and the other to achieve the required functionalities through other existing systems (*hybrid* databases). In both cases the two challenges to solve are how to store the data and how to traverse these stored structures.

3.3.1 Native System Architectures:

For data storage, a common design principle is to separate information about the graph structure (nodes and edges) from other they may have, e.g., attribute values, to speed-up traversal operations.

Neo4J has one file for node records, one file for edge records, one file for labels and types, and one file for attributes. **OrientDB** stores information about nodes, edges and attributes similarly, in distinct records. In both systems, node and edge records contain pointers to other edges and nodes, and also to types and attributes, but the organization is different in the two systems. In Neo4J nodes and edges are stored as records of fixed size and have unique IDs that correspond to the offset of their position within the corresponding file. In this way, given the id of an edge, it is retrieved by multiplying the record size by its id, and reading bytes at that offset in the corresponding file. Moreover, being records of fixed size, each node record points only to the first edge in a doubly-linked list, and the other edges are retrieved by following such links. A similar approach is used for attributes. In OrientDB, on the other hand, record IDs values are not linked directly to a physical position, but point to an append-only data structure, where the logical identifier is mapped to a physical position. This allows for changing the physical position of an object without changing its identifier. In both cases, given an edge, to obtain its source and destination requires constant time operations, and inspecting all edges incident on a node, hence visiting the neighbors of a node, has a cost that depends on the node degree and not on the graph size.

Sparksee decomposes data into separate data-structures: one structure for objects, which refers to both nodes and edges, two for relationships which describe which nodes and edges are linked to each other, and a data-structure for each attribute name. Each of these data-structures is in turn composed by a map from keys to values, and a bitmap for each value [49]. In each data-structure objects are identified by IDs generate sequentially, and each ID is linked as key trough the map to one single value. Also, each value links to a bitmap, where each bit corresponds to an object ID, and the bit is set if that object has that value. Given a label, one can scan the corresponding bitmap to identify which edges share the same label. Furthermore, each bitmap identifies all edges incident to a node. For the attributes a similar mechanism is used. The main advantage of this organization is that many operations become bitwise operations on bitmaps, although operations like edge traversals have no constant time guarantees.

3.3.2 Hybrid System Architectures:

ArangoDB is based on a document store. Each document is represented as a self contained JSON object (serialized in a compressed binary format). To implement the graph model, ArangoDB materialize JSON objects for each node and edge. Each object contains links to the other objects to which it is connected, e.g., a node lists all the IDs of incident edges. A specialized hash index is in place, in order to retrieve the source and destination nodes for an edge, this speed-ups many traversals. **BlazeGraph** is an RDF database and stores all information into Subject-Predicate-Object (SPO) triples. Each statement is indexed three times by changing the order of the values in each triple, i.e., a B+Tree is built for each one of SPO, POS, OSP. BlazeGraph stores attributes for edges as reified statements, i.e., each edge can assume the role of a subject in a statement. Hence, traversing the structure of the graph may require more than one accesses to the corresponding B+Tree.

In **Sqlg** the graph structure consists of one table for each edge type, and one table for each node type. Each node and edge is identified by a unique ID, and connections between nodes and edges are retrieved through joins. The limitation

of this approach is that unions and joins are required even for retrieving the incident edges of a node.

Finally, **Titan** represents the graph as a collection of adjacency lists. With this model the system generates a row for each node, and then one column for each node attribute and each edge. Hence, for each edge traversal, it needs to access the node (row) ID index first.

3.3.3 *Query Processing and Evaluation:*

All the systems we considered support Gremlin. A Gremlin query is a series of operations. Consider, for instance, query 28 in Table 2, which selects nodes with at least k incoming edges. It first filters nodes (`g.V.filter{...}`) and then the incoming edges are counted (`it.inE.count()`) for every node in the output of the filter.

In **ArangoDB** each step is converted into an AQL query and sent to the server for execution, so the above Gremlin query will be executed as a series of two independent AQL queries implementing its two parts. ArangoDB does not provide any overall optimization of these parts. Note that Gremlin is a touring-complete language and can describe complex operations that declarative languages, like AQL or Cypher, may not be able to express in one query. The only other query system that translates all operations to a declarative query language is **Sqlg**. Where possible, the system tries to conflate operators in a single query, which is some form of query optimization. All the other systems translate Gremlin queries directly into a sequence of low-level operators with direct access to their programming API, evaluate every operator, and pass the result to the next in the sequence. In **OrientDB**, in particular, some consequent operators may get translated into queries and then the processed with the programming API, resulting in some form of optimization for a part of the query. **Titan**, which has Gremlin as the only supported query language, features also some optimization during query processing.

4. QUERIES

To generate the set of queries to run on the systems we follow a micro-benchmark approach [25]. The list is the results of an extensive study of the literature and of many practical scenarios. Of the many complex situations we found, we identified the very basic operations of which they were composed. We eliminated repetitions and ended up with a set of common operations that are independent from the schema and the semantics of the underlying data, hence, they enjoy a generic applicability.

In the query list we consider different types of operations. We consider all the "CRUD" kinds, i.e., **C**reations, **R**eads, **U**pdates, **D**eletions, for nodes, edges, their labels, and for their properties. Specifically for the creation, we treat separately the case of the initial loading of the dataset from the individual object creations. The reason is because the first happens in bulk mode on an empty instance, while the second at run time with data already in the database. We also consider *traversal* operations across nodes and edges, which is characteristic in graph databases. Recall that operations like finding the centrality, or computing strongly connected components are for graph analytic systems and not typical in a graph database. The categorization we follow is aligned to the one found in other similar works [18, 40, 41] and benchmarks [32]. The complete list of queries can be found in Table 2 and is analytically presented next. The syntax is for Gremlin 2.6, but the syntax for gremlin version 3 is quite similar.

4.1 Load Operations

Data loading is a fundamental operation. Given the size of modern datasets, understanding the speed of this operation is crucial for the evaluation of a system. The specific operator (Query 1) reads the graph data from GraphSON[2] file. In general it's bound to the speed with which objects are inserted, which will be affected by any index in place and any other consistency check. In some cases GDBs have in place special methods or configurations to allow bulk loading, e.g., to deactivate indexing, but in general they are vendor specific, i.e., not found in the Gremlin specifications. Some of them will be described later (Section 7).

4.2 Create Operations

The first category of operations (**C**) includes operators that create new structures in the database. In this group we consider anything that generates new data-entries. Creation operators may be for nodes, edges, or even properties on existing nodes or edges. Often, to create a complex object, e.g., a node with a number of connections to existing nodes, many different such operators may have to be called. Among the others, we also consider a special composite workload where we first insert a node and then also a set of edges connecting it to other existing nodes in the graph.

Insert Node (Query 2) The operator creates a new node in the database with the set of properties that are provided as argument, but without any connection (edges) to other nodes.

Insert Edge (Queries 3, and 4) This operator creates a new edge in the graph between the two nodes specified as arguments, and with the provided label. In a second version, the operator can also take a set of properties as additional argument. In the experiments performed we randomly select nodes among those in the graph, we choose a fresh value as label, and a custom property name and value pair.

Insert Property (Queries 5, and 6) These two operators test the addition of a new property to a specific node and to a specific edge, respectively. The node (or the edge) is explicitly stated, i.e., referred, through its unique id, and, there is no search task involved since the lookup for the object with the specific identifier is performed before the time is measured. In this case the operation are applied directly to the node and edge (`v` and `e`).

Insert Node with Edges (Query 7) This operation requires the insertion of a new node, alongside a number of edges that connect it to other nodes already existing in the database.

4.3 Read Operations

The category of read operations comprises queries that locate and access some part of the graph data stored in the system that satisfy certain conditions. Sometimes, such part may end up being the entire graph.

Graph Statistics (Queries 8, 9, and 10) Many operations often require a scan over the entire graph datasets. Among

[2]A JSON-based format `tinkerpop.apache.org/docs/current/reference/#graphson-io-format`

Table 2: Test Queries by Category (in Gremlin 2.6)

#	Query	Description	Cat
1.	g.loadGraphSON("/path")	Load dataset into the graph 'g'	**L**
2.	g.addVertex(p[])	Create new node with properties p	**C**
3.	g.addEdge(v1 , v2 , l)	Add edge l from $v1$ to $v2$	
4.	g.addEdge(v1 , v2 , l , p[])	*Same as q.3*, but with properties p	
5.	v.setProperty(Name, Value)	Add property *Name*=*Value* to node v	
6.	e.setProperty(Name, Value)	Add property *Name*=*Value* to edge e	
7.	g.addVertex(...); g.addEdge(...)	Add a new node, and then edges to it	
8.	g.V.count()	Total number of nodes	**R**
9.	g.E.count()	Total number of edges	
10.	g.E.label.dedup()	Existing edge labels (no duplicates)	
11.	g.V.has(Name, Value)	Nodes with property *Name*=*Value*	
12.	g.E.has(Name, Value)	Edges with property *Name*=*Value*	
13.	g.E.has('label',l)	Edges with label l	
14.	g.V(id)	The node with identifier *id*	
15.	g.E(id)	The edge with identifier *id*	
16.	v.setProperty(Name, Value)	Update property *Name* for vertex v	**U**
17.	e.setProperty(Name, Value)	Update property *Name* for edge e	
18.	g.removeVertex(id)	Delete node identified by *id*	**D**
19.	g.removeEdge(id)	Delete edge identified by *id*	
20.	v.removeProperty(Name)	Remove node property *Name* from v	
21.	e.removeProperty(Name)	Remove edge property *Name* from e	
22.	v.in()	Nodes adjacent to v via incoming edges	**T**
23.	v.out()	Nodes adjacent to v via outgoing edges	
24.	v.both('l')	Nodes adjacent to v via edges labeled l	
25.	v.inE.label.dedup()	Labels of in coming edges of v (no dupl.)	
26.	v.outE.label.dedup()	Labels of outgoing edges of v (no dupl.)	
27.	v.bothE.label.dedup()	Labels of edges of v (no dupl.)	
28.	g.V.filter{it.inE.count()>=k}	Nodes of at least k-incoming-degree	
29.	g.V.filter{it.outE.count()>=k}	Nodes of at least k-outgoing-degree	
30.	g.V.filter{it.bothE.count()>=k}	Nodes of at least k-degree	
31.	g.V.out.dedup()	Nodes having an incoming edge	
32.	v.as('i').both().except(vs).store(j).loop('i')	Nodes reached via breadth-First traversal from v	
33.	v.as('i').both(*ls).except(j).store(vs).loop('i')	Nodes reached via breadth-First traversal from v on labels ls	
34.	v1.as('i').both().except(j).store(j).loop('i') {!it.object.equals(v2).retain([v2]).path()}	Unweighted Shortest Path from $v1$ to $v2$	
35.	v1.as('i').both('l').except(j).store(j).loop('i') {!it.object.equals(v2).retain([v2]).path()}	*Same as q.34*, but only following label l	

* The symbol [] denotes a Hash Map structure

the queries of this type, three operators were included in the query evaluation set. One that scans and counts all the nodes, one that does the same for all edges, and one that counts the unique labels of the edges. The goal of the last operation is also to stress-test the ability of the system to maintain intermediate information in memory, since it requires to eliminate duplicated before reporting the results.

Search by Property (Queries 11, and 12) These two queries are typical selections. They identify nodes (or edges) that have a specific property. The name and the value of the property are provided as arguments. There may be a unique object satisfying the condition of having the specific property, or there may be more than one.

Search by Label (Query 13) The search by label task is similar to the search by property, but has only one operator since labels are only on edges. Labels are fundamental components of almost every graph dataset, and this is probably the reason why the syntax in Gremlin 3.x distinguishes between labels and properties with a special provision, while in 2.6, they were treated equally. In a graph database edge labels have a primary role, also usually, labels are not optional and are immutable, hence searching edges based on a specific label should receive special attention.

Search by Id (Queries 14, and 15) As it happens in almost

any other kind of database, a fundamental search operation is the one done by reference to a key, i.e., ID. Those are *system defined*, and in some cases based on the internal data organization of the system. These two queries have been included, to retrieve a node and an edge via their unique identifier.

4.4 Update Operations

Data update operators are typical of dynamic data, and graph data is no exception. Since edges are first class citizens of the system, an update of the structure of the graph, i.e., on the connectivity of two or more nodes, requires either the creation of new edges or deletion of existing. In contrast, updates on the properties of the objects are possible without deletion/insertion, as it happens in other similar databases. Thus, we have included Queries 16, and 17 to test the ability of a system to change the value of a property of a specific node or an edge. In this case, as above, we do not consider the time required to first retrieve the object to be updated.

4.5 Delete Operations

To test how easily and efficiently data can be removed from a graph database, we included three types of deletions: one for a node, one for an edge and one for a property.

Delete Node (Query 18) Deleting a specific node requires the elimination of all its properties, all its edges, as well as the node itself. It may result to a very costly operation when many different data-structures are involved.

Delete Edge (Query 19) Similarly to the node case, deleting an edge requires the prior removal of its properties. This operation is probably one of the most common delete operations in continuously evolving graphs.

Delete Property (Queries 20, and 21) The last two queries eliminate a property from an node or an edge, respectively. As the structure of a node or edge is not fixed, it may happen that either element lose a property.

4.6 Traversals

The ability to conveniently perform traversal operations is one of the main reason why graph models are preferred to others. A traversal means moving across different nodes that are connected in a consecutive fashion through edges.

Direct Neighbors (Queries 22, 23) A popular operation is the one that, given a node, retrieves those directly reachable from it (1-hop), i.e., those that can be found by following either an incoming or an outgoing edge.

Filter Direct Neighbors (Query 24) The specific query performs a traversal of only one hop, and for edges having a specific label. The reason why it is considered separately from other traversals is because it is very frequent and involves no recursion, and as such, it is often subject to separate efficient implementation by the various systems.

Node Edge-Labels (Queries 25, 26, and 27) Given a node, there is often the need to know the labels of the incoming, outgoing, or both edges. These three kinds of retrieval is exactly what this set of three queries perform, respectively.

K-Degree Search (Queries 28, 29, 30, and 31) For many real application scenarios there is a need to identify nodes with many connections, i.e., edges, since this is an indicator of the importance of a node. The number of edges a node has is called the degree of the node, and nodes with high degree are usually hubs in the network. The first three queries identify and retrieve nodes with at least k edges. They differ from each other in considering only incoming edges, only outgoing, or both. The fourth query identifies nodes with at least one incoming edge and is often used when a hierarchy needs to be retrieved.

Breadth-First Search (Queries 32, and 33) A number of search operations give preference to nodes found in close proximity, and they are better implemented with a breadth-first search from a given node. This ability is tested with these two queries, with the second being a special case of the first that considers only edges with a specific label.

Shortest Path (Queries 34, and 35) Another traditional operation on graph data is the identification of the path between two nodes that contain the smallest number of edges. For this we included these two queries, with the second query being a special case of the first that considers only edges with a specific label. In particular, given an unweighted graph, they determine the shortest path between two nodes via a BFS-like traversal.

4.7 Complex Query Set

In order to test the ability of the systems to optimize complex queries, i.e., collectively optimize multiple primitive operators, we also created a workload of 12 queries based on queries provided by the LDBC Social Network benchmark [32]. The queries mimic the tasks carried out for a new user in the system, from the point of creating an account (creating a new node with attributes) and filling up her profile (connecting to nodes representing the school, the place of birth and the workplace), to the point of making recommendations of topics and other users. For these operations there are queries in the workload with composition of multiple primitive operators, multiple joins predicates, group by, sorting, max finding, and top-k. Since these queries are heavily dependent on the schema of the dataset, they can be run solely on the *ldbc* dataset presented below (Section 5).

Max-search To add a new user we should assign an unique identifier to it. In the *ldbc* dataset objects have two different properties, one is '*oid*' and the other is the '*iid*', the first is a string the second is an integer. Although in real applications this is handled different, here we search for the maximum value for both (queries '*max-iid*' and '*max-oid*'), and the we will increment these values and use them when creating a new node for a user.

User Creation We create a new node for a user, we take all the attributes that a user has and attach those attribute to the node created. We will also assign to it the two values for '*iid*' and '*oid*' obtained previously.

User Profile Once the node for the user is created, we connect it to other nodes signifying some personal details. In particular we issue a query for finding places, companies and universities with name starting with some combination of letters. For each of those we take the first in alphabetical order and add an edge between the node of the user and the node of the place. Those represent the city where she lives, the company where she works at, and the university

in which she studied.

User Friends We also search and add as friends other existing users. Users to be added as friends (i.e., connected via an edge labeled 'knows') are selected based on a selection on their first name ('*friend1*'), and on a selection based on both first and last name ('*friend2*'). All results are sorted in ascending order based on the name and only the top-10 results are returned.

Recommending Tags Tags are similar to labeled topics, and users are connected to the nodes representing the topics they like. We first select among the tags liked by the user friend, the top 10 tags with highest number of likes ('*friend-tags*'), and then connect those tags to the user ('*add-tags*').

Friend Search and Recommendation The last part of the workload explored the neighborhood of a user node searching for friends. In the first query ('*friend-of-friend*') we find up to 10 people with a given first name that the user is connected to by at most 3 steps via 'knows' relationships. For the retrieved persons we return their personal information, including workplaces and places of study. The retrieved persons are sorted by their distance from the user.

In the second query we recommend instead friend of friends, that are not already friend of the user, simulating in this way a first iteration of triangle closure.

5. EVALUATION METHODOLOGY

Fairness, reproducibility, and extensibility have been three fundamental principles in our evaluation of the different systems. In particular, a common query language and input format for the data has been adopted for all the systems. For the query executions, it has been ensured that they have been performed in isolation so that they have not been affected by external factors. Any random selection made in one system (e.g., a random selection of a node in order to query it) has been maintained the same across the other systems. Furthermore, all experiments have been performed on the same machine to avoid any effect caused by hardware variations. The goal is to perform a comparative evaluation and not an evaluation in absolute terms. Both real and synthetic datasets have been used, especially on large volumes in order for the experiments to be able to highlight the differences across the systems. Finally, our evaluation methodology has been materialized in a test suite and is available on-line [43] It contains scripts, data and queries, and is extensible to new systems and queries.

Common Query Language. We have opted for a common query language across all the systems to ensure that the semantics of the queries we run are interpreted in the same way by the different systems. In particular, we selected as application layer the Apache TinkerPop [5] framework and the expressive query language Gremlin [56], which is the most supported language across graph databases. In the context of graph databases, TinkerPop acts as a database-independent connectivity layer, while Gremlin is the analogous to SQL in relational databases [38]. All the graph databases we tested have adapters for Gremlin already implemented and supported by the various database vendors.

Software Containers. To ensure full control over the environment in which each system runs, and to facilitate reproducibility, we opted for installing and running each graph database within a dedicated software container [24]. A popular solution is Docker [8], an open source software that creates a level of "soft" virtualization of the operating system, which allows an application within the environment to access machine resources directly without the overhead of interacting with an actual virtual machine. Furthermore, thanks to the so called *overlay file-system* (AUFS [53]), it is possible to create a "snapshot" of a system and it's files, and then share the entire computational environment. This allows the sharing of our one-click installation scripts for all the databases and our testing environment, so that the experiments can be repeated elsewhere.

Hardware. For the experiments we used a machine with a 24-core CPU, an Intel Xeon E5-2420, 1.90GHz processor, 128 GB of RAM, 2TB hard disk with 20000 rpm, Ubuntu 14.04.4 operating system, and with Docker 1.13, configured to use AUFS on *ext4*. Each graph database was configured to be free to use all the available machine resources, e.g., for the JVM we used the option -Xmx120GB. For other parameters we used the settings recommended by the vendor. The latter applies also to Apache Cassandra that was serving as the back-end for Titan.

Evaluation Approach. The Gremlin queries are called for execution via Groovy[3] scripts. For the systems supporting different major versions of Gremlin, we tested both. The reason is that since the latest version came out recently, we would like to illustrate the difference in performance that has been achieved and help stakeholders having an old system in in operation to decide whether it is worth the extra step of upgrading them. Furthermore, the difference between the versions illustrates the space for improvement that exists, an area that our current work can help significantly.

All system were tested using the *embedded mode*, where direct Java calls are sent to the system, and the application runs within the JVM of the engine. The only exception was ArangoDB and Sqlg. They may receive Gremlin commands through the Java API, but in the back-end perform REST and/or JDBC calls to the underlying storage engine. Nonetheless, since all storage systems operate locally, there is no delay due to network routing or latency.

Note that Gremlin has no specification for indexes. Some systems create indexes automatically in a way agnostic to the user while others require explicit commands in their own language. We considered both the default behavior of not taking any action and letting the system go with its default indexing policy, and the case of explicitly creating the needed indexes.

In the case of queries with parameters, for fair comparisons, the parameter values are decided in advance and kept the same for all the systems. For instance, query 14 needs the ID of the node to retrieve. If a different node is retrieved in every system, then it wont be possible to compare them. For this reason, when we need to evaluate a query, we first decide the parameters to use and then start the executions on the different systems. The same applies on queries that need to start from a node or an edge, e.g. query 22 needs to know the node v. For these queries, the node is decided first and then the query is run for that same node in all the systems. Naturally, the time needed to identify the node (or edge) that will be used in the query and retrieve its id, is

[3]A superset of Java: `groovy-lang.org`

not part of the time reported as execution time for the respective queries. A similar approach is followed also for the multi-fold evaluation. When we perform k runs of the same query on the same system and dataset (to select the average response time as the indicative performance), we first select k parameter values, nodes, or edges to query (usually through random sampling), and then perform each of the k runs with one of these k parameters (or nodes, or edges). Here each query is executed k =10 times.

In the scalability studies of queries 11 and 12 that are performing selection based on a property value, it is important that the performance variation observed when running the same query on datasets of different sizes is due to the size of the data and not due to the cardinality variation of the answer set. To achieve this goal, we select to use properties that not only exist in all the datasets of different sizes, but also have the same cardinality in all of them. In case such properties do not exist, we create and assign at loading time two different properties with predefined names to 10 random edges and 10 random nodes in each dataset and then use these property names for queries 11 and 12. (The different case of the same type of query run on the same dataset producing results of different cardinalities is covered by the different parameter values that are decided in the k-fold experiments.)

Unfortunately, almost all the databases, when loading the data, create and assign their own internal identifiers. This creates a problem when we later need to run across all the systems the same query that is using the identifier as a parameter. For this reason, when loading the data, we add to every node a property $\langle$"$objectID$", $id\rangle$ where the id is the node identifier in the original dataset. As a result, even if the system decides to replace the identifier of the node with an internal one, we still have a way to find the node using the $objectID$ property. So before starting the evaluation of query $g.V(id)$, for instance, on the graph database system S, where id is the node identifier in the original dataset, we first search in the system S and retrieve the internal identifier iid of the node with the attribute $\langle$"$objectID$", $id\rangle$. Then, we execute the query $g.V(iid)$ instead of the $g.V(id)$, and report its execution time as the time for the evaluation of Q.14.

The k times that a query execution is repeated are performed first in isolation and then in batch mode. For the isolation, we turn the system on, run the single query with one of the parameters, then shut the system off, and reset the file-system. Then repeat again with the next parameter. In this way, each run is unaffected by what has run before. In batch mode, we turn the system on, run the query with the first parameter, then with the second, then the third, and so forth. At the end we shut down the system. The isolation mode makes no sense to be repeated for the queries 8, 9, 10, 28, 29, 30 and 31 since they have no graph-dependent parameters, thus, every isolation mode repetition will be identical to the others. Thus, these queries are evaluated only once in isolation and not in batch. In queries 28, 29 and 30, the k has been considered a threshold and not a parameter, and the fixed value k=50 has been considered throughout the experiments. In total, for every evaluation of a specific system with a specific dataset, 337 query executions are taking place. To these we add the 120 queries from the LDBC benchmark.

Test Suite. We have materialized the evaluation procedure into a software package (a test suite) and have made it available on-line [43], enabling repeatability and extensibility. The suite contains the scripts for installing and configuring each database in the Docker environment and for loading the datasets. The queries themselves are also contained in the suite. There is also a python script that instantiates the Docker container and provides the parameters required by each query. To test a new query it suffices to write it into a dedicated script, while in order to perform the tests on a new dataset one only needs to place the dataset in GraphSON format in a JSON file in the directory from where the data are loaded.

Datasets. We have tested our system on both real and synthetic datasets. One dataset (*MiCo*) describes co-authorship information crawled from the CS Microsoft Academic portal [31]. Nodes represent authors while edges represent co-authorship between two authors and have as a label the number of co-authored papers. Another dataset (*Yeast*) is a protein interaction network [23]. Nodes represent budding yeast proteins (*S.cerevisiae*) [26] and have as labels the short name, a long name, a description, and a label based on its putative function class. Edges represent protein-to-protein interactions and have as label the two classes of the proteins involved. A third real dataset is Freebase [34], which is one of the largest knowledge bases nowadays. Nodes represent entities or events, and edges model relationships between them. We have taken the latest snapshot [42, 52] and have considered four subgraphs of it of different sizes.

Despite the fact that the raw data dump contains 1.9B triples [34], those comprise duplicate, technical, or experimental meta-data and links to other sources that are commonly removed [22, 52], leaving a clean dataset of 300M facts. The size of the samples were chosen to ensure that all engines had a fair chance to process them in reasonable times, and on the other hand to show the system scalability at levels higher than those of previous works.

One subgraph (*Frb-O*) was created by considering only the nodes related to the topics of organization, business, government, finance, geography and military, alongside their respective edges. Furthermore, we randomly selected 0.1%, 1%, and 10% of the edges from the complete graph, which alongside the nodes at their endpoints created 3 graph datasets, the *Frb-S*, *Frb-M*, and *Frb-L*, respectively.

For a synthetic dataset we used the data generator[4] provided by the Linked Data Benchmark Council[5] (LDBC) [32] to produce a graph that mimics the characteristics of a real social network with power-law structure, and real-word characteristics like assortativity based on interests or preferences (*ldbc*). The generator was instructed to produce a dataset simulating the activity of 1000 users over a period of 3 years. The *ldbc* is the only dataset with properties on both edges and nodes. The others have properties only on the nodes.

Table 3 provides the characteristics of the aforementioned datasets. It reports the number of nodes ($|V|$), edges ($|E|$), labels ($|L|$), connected components (#), the size of the maximum connected component (Maxim), the graph density (Density), the network modularity (Modularity), the average degree of connectivity (Avg), the max degree of connectivity (Max), and the diameter (Δ).

As shown in the table, the *MiCo* and the *Frb* are sparse, while the *ldbc* and *Yeast* are one order of magnitude denser,

[4] `github.com/ldbc/ldbc_snb_datagen`

[5] `ldbcouncil.org`

which reflects their nature. The *ldbc* is the only dataset with a single component, while the *Frb* datasets are the most fragmented. The average and maximum degree are reported because large hubs become bottleneck in traversals.

Evaluation Metrics. For the evaluation we consider the disk space, the data loading time, the query execution time, but we also comment on the experience with installing and running each system.

6. EXPERIMENTAL RESULTS

In the tests we run we noticed that *MiCo* and *ldbc* were giving results similar to the *Frb-M* and *Frb-O*. The *Yeast* was so small that didn't highlight any particular issue, especially when compared to the results of *Frb-S*. We also tried to load the full freebase graph *Frb-F* (with 314M edges and 76M nodes), but only Neo4J, Sparksee, and Sqlg managed to do so without errors, and only Neo4J (v.3.0) to successfully complete all the queries, making it the most scalable. Moreover, the running times recorded on the full dataset respected the general trends witnessed with the other subsamples. Thus, in what follows, we will talk mainly about the results of the *Frb-S*, *Frb-O*, *Frb-M*, and *Frb-L* and only when there is some different behavior from the others we will be mentioning it. Additional details about the experimental results on the other datasets can be found below (Section 6.9).

Regarding the documentation, Neo4J, Sqlg, and OrientDB provide in-depth information. Sparksee, Titan and ArangoDB are limited in some aspects, yet clear for basic installation, configurations and operational needs. The Titan documentation is the less self-contained, especially on how to be configured with Cassandra. Finally, the BlazeGraph documentation is largely outdated.

In terms of system configuration Neo4J doesn't require any specific set-up. OrientDB instead supports a default maximum number of edge labels equal to 32676 divided by the number of CPU cores, and requires disabling a special feature for supporting more. Sqlg has limits on the maximum length of nodes and edge labels (inherited from Postgresql). ArangoDB requires two configurations, one for the engine, and one for the V8 javascript server for logging. With only default values this system generated 40 GB of log files in about 24 hours of activity, with a single active client and it is not able to allocate more than 4GB of memory. For Titan instead the most important configurations are for the JVM Garbage Collection and for the Cassandra backend. Moreover, for large datasets, it is necessary to disable automatic schema creation, and create it manually before data loading.

Finally, the systems based on Java, namely, BlazeGraph, Neo4J, OrientDB and Titan, are sensitive to the JVM garbage collection, especially for very large datasets that require large amount of main-memory. As a general rule, the option `-XX:+UseG1GC` for the *Garbage First* (G1) garbage collector is strongly recommended.

6.1 Data Loading

The Task. For many systems, loading the data simply by executing the Gremlin query 1 was causing system failures or was taking days. For OrientDB and ArangoDB we are forced to load the data using their native routines. With Gremlin, ArangoDB sends each node and edge insertion instruction separately to the server via a HTTP call making it prohibitively slow even for small datasets. For OrientDB, limited edge-label storing features and long loading times required us to pass through some server-side implementation-specific commands in order to load the datasets. BlazeGraph required the explicit activation of the *bulk loading* feature otherwise we were facing loading times in the order of days. Titan, for any medium to large size dataset requires disabling the automatic schema creation during loading, otherwise its storage back-end (Cassandra) would get swamped with extra consistency check operations. This means that the complete schema of the graph should be known to the system prior to the insertion of the data and is immutable unless implementation specific directives are issued to update it. Sqlg, instead, has a limit on the maximum length of labels (due to Postgresql). In the Gremlin implementation in all other systems those operations are transparent to the user. As a result, only Neo4J and Sparksee managed the loading through the gremlin API with no issues, and they did so in times comparable to those achieved by the built-in scripts provided by ArangoDB. Consequently, since (for the loading alone) the different systems did not go through exactly the same procedures, discussions regarding the loading times need to be taken with this information in mind.

The time. For the *Yeast*, which is the smallest dataset, loading times vary from a couple of seconds (with ArangoDB) to a minute (with Titan (v.1.0)). With the *Frb-S* dataset, loading times range from 16 seconds (with ArangoDB), 5 minutes (Titan and OrientDB), 16 minutes (BlazeGraph), up to 42 minutes (Sqlg). Titan and OrientDB are the second slowest, requiring around 5 minutes. Neo4J is usually the second fastest in all loadings tasks being only ten seconds slower than ArangoDB. This ranking stays similar with *MiCo* and *ldbc*, with the only exception of Sqlg being much more faster.

Using the *Frb-O*, *Frb-M*, *Frb-L*, we observed that loading time increases proportionally to the number of elements (nodes and edges) within each dataset. With the largest dataset (*Frb-L*) ArangoDB has the fastest loading time (∼19 min) and only Neo4J (v.3.0) is just few minutes slower, followed by Neo4J (v.1.9) (∼38 min), and Sparksee (∼48 min). OrientDB, instead, took almost 3 hours, while both versions of Titan approximately 4.5 hours. BlazeGraph instead took almost 4.45 hours to load *Frb-M* and around 4 days to load *Frb-L*. These tests showed that BlazeGraph, Sqlg and OrientDB, given they internal storage structure, are very sensitive to the edge label cardinality.

The Space. We exploited the docker utilities to measure the disk size occupied by the data in each system. The results are illustrated in Figures 3(a) and 3(b). For each system, we obtained the size of the docker image with the system installed and its required libraries, then we measured again the size of said image after the data loading step. The difference gives a precise account of all files that the loading phase has generated.

Loading *Yeast*, not reported in figure, leaves the image size almost unchanged for both Neo4J (v.1.9) and Titan (v.0.5), and only 10, 20, 30, 60 and 70MB are added for Neo4J (v.3.0), Sparksee, Titan (v.1.0), Sqlg, and OrientDB, respectively. Instead, ArangoDB generates almost 150MB of

Table 3: Dataset Characteristics

	|V|	|E|	|L|	Connected Component #	Connected Component Maxim	Density	Modularity	Degree Avg	Degree Max	Δ
Yeast	$2.3K$	7.1K	167	101	$2.2K$	$1.34{*}10^{-3}$	$3.66{*}10^{-2}$	6.1	66	11
MiCo	$100K$	1.1M	106	$1.3K$	$93K$	$1.10{*}10^{-6}$	$5.45{*}10^{-3}$	21.6	$1.3K$	23
Frb-O	$1.9M$	4.3M	424	$133K$	$1.6M$	$1.19{*}10^{-6}$	$9.82{*}10^{-1}$	4.3	$92K$	48
Frb-S	$0.5M$	0.3M	1814	$0.16M$	$20K$	$1.20{*}10^{-6}$	$9.91{*}10^{-1}$	1.3	$13K$	4
Frb-M	$4M$	3.1M	2912	$1.1M$	$1.4M$	$1.94{*}10^{-7}$	$7.97{*}10^{-1}$	1.5	$139K$	37
Frb-L	$28.4M$	31.2M	3821	$2M$	$23M$	$3.87{*}10^{-8}$	$2.12{*}10^{-1}$	2.2	$1.4M$	33
ldbc	$184K$	1.5M	15	1	$184K$	$4.43{*}10^{-5}$	0	16.6	$48K$	10

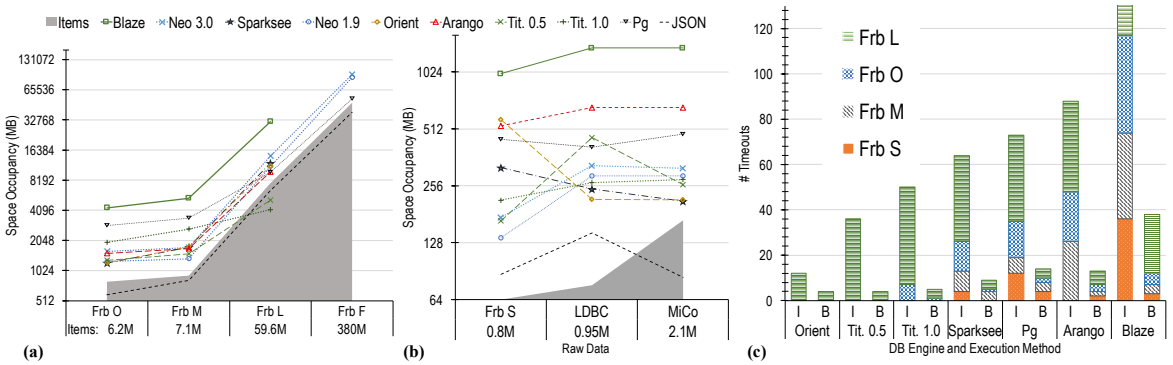

Figure 3: Space occupancy on disk required by the systems on the various datasets compared to the size of the original Json file and number of elements in the dataset ((left) and (center)) and number of Time-Outs for Isolation (I) and Batch (B) modes (right)

Table 4: Evaluation Summary

Task	★★★	★★	★
Load	$N_1 \cdot N_3 \cdot S$	$A \cdot O \cdot P$	$T_0 \cdot T_1 \cdot B$
Insertions	$S \cdot O \cdot N_1 \cdot A$	$P \cdot N_3 \cdot T_0 \cdot T_1$	B
Graph Statistics	$S \cdot N_3$	$N_1 \cdot O \cdot P$	$T_0 \cdot T_1 \cdot A \cdot B$
Search by Property	$P \cdot N_3 \cdot N_1$	O	$S \cdot T_0 \cdot T_1 \cdot A \cdot B$
Search by Label	$P \cdot N_3 \cdot N_1$	$O \cdot S$	$T_0 \cdot T_1 \cdot A \cdot B$
Search by Id	$S \cdot N_1 \cdot O$	$A \cdot P \cdot N_3 \cdot T_0 \cdot T_1$	B
Updates	$S \cdot A \cdot O \cdot N_1 \cdot P$	$N_3 \cdot T_0 \cdot T_1$	B
Delete Node	$A \cdot N_1$	$S \cdot O \cdot T_0 \cdot T_1 \cdot N_3 \cdot P$	B
Other Deletions	$S \cdot A \cdot O \cdot N_1 \cdot P$	$T_0 \cdot T_1 \cdot N_3$	B
Direct Neighbors	$N_1 \cdot O \cdot N_1$	$A \cdot S \cdot T_0 \cdot T_1$	$B \cdot P$
Node Edge-Labels	$N_1 \cdot O \cdot N_3$	$A \cdot T_0 \cdot T_1$,S	$B \cdot P$
K-Degree Search	$N_3 \cdot N_1$	$O \cdot T_0 \cdot T_1$	$A \cdot B \cdot P$
Breadth-First Search	$N_3 \cdot O \cdot N_1$	$T_0 \cdot T_1$	$A \cdot S \cdot B \cdot P$
Shortest Path	$N_3 \cdot N_1$	$O \cdot T_0$	$A \cdot T_1 \cdot S \cdot B \cdot P$
SN Queries	$P \cdot N_3$	$O \cdot N_1 \cdot S \cdot T_1$	$A \cdot T_0 \cdot B$

A=ArangoDB; B=BlazeGraph; N1=Neo4J (v.1.9); N3=Neo4J (v.3.0); O=OrientDB; S = Sparksee; T0 =Titan (v.0.5); T1 =Titan (v.1.0); P=Sqlg

additional disk space, and BlazeGraph more than 830MB, the latter due to the size of journal and automatic-indexing that, when generated, are multiples of a fixed size.

With the *Frb-O* dataset, as Figure 3 illustrates, Sparksee, OrientDB, Neo4J (v.1.9), and Titan (v.0.5) are all equally *compact*, with a delta on the disk image size of about 1.2GB. For *Frb-M*, though, Neo4J (v.1.9) and Titan (v.0.5) are equally effective in disk size and a little better than the others, requiring respectively 1.3GB and 1.5GB to store a dataset of 816MB and 7.1 million elements.

Titan (v.1.0) has, on both medium size datasets (*Frb-O* and *Frb-M*), the third worst performance (the worst being BlazeGraph and the second worst Sqlg), with three to four times the space consumption of the original dataset in plain text. Instead, for the *Frb-L*, Titan (v.1.0) scales the best, compressing 6.4GB of raw data into 4.1GB, followed by Titan (v.0.5) taking 5.1GB. The remaining databases are almost equivalent, taking from 10 to 14GB. Exception is the BlazeGraph, on all the datasets, requiring on average three times the size of any other system. This shows that the compression strategy of Titan is the most compact at larger scales.

The comparison between the disk space required by the systems to store *Frb-S*, *MiCo* and *ldbc* (Figure 3(b)) reveals a peculiar behavior for Sparksee and OrientDB, where the space occupied on disk is smaller for the two larger datasets. As a matter of fact, the *ldbc* dataset stored as plain text file occupies twice more space on disk than the *Frb-S* file, and contains 2 hundred thousands more elements. Nonetheless Sparksee requires for *ldbc* about 25% less space, and OrientDB less than half the space occupied on disk by the corresponding image with *Frb-S*. *MiCo* has comparable size, in plain text, to *Frb-S*, and contains twice the objects of *Frb-S*, but still the respective docker images of OrientDB and Sparksee for *MiCo* are almost half the size of their images containing the *Frb-S*. These disproportions can be explained by the fact that *Frb-S* contains almost ∼2K different edge labels, while *MiCo* 106, and *ldbc* only 15. Apparently these systems store different data-structure for different edge la-

bels, causing a certain amount of overhead for datasets with many such labels.

It is important to note here that we have tried also much larger datasets, but we were not able to load them on a large number of systems so we could not have comparison across all the systems and we have not reported them.

6.2 Completion Rate

Since graph databases are often used for on-line applications, ensuring that queries terminate in a reasonable time is important. We count the queries that could not complete within 2 hours, in isolation or in batch mode, and illustrate the results in Figure 3(c). Note that, if one instantiation of one query fails to run within the alloted amount of time in isolation, when executed in a batch it will cause the failure of the entire batch as well.

Neo4J, in both version, is the only system which completed successfully all tests with all parameters on all datasets (omitted in the figure). OrientDB is the second best, with just few timeouts on the large *Frb-L*. BlazeGraph is at the other end of the spectrum, collecting the highest number of timeouts. It reaches the time limit even in some batch executions on *Yeast*, and almost on all queries on *Frb-L*. In general the most problematic queries are those that have to scan or filter the entire graph, i.e., queries Q.9 and Q.10. Some shortest-path searches, and some bread first traversal with depth 3 or more in most databases reach the timeout on *Frb-O*, *Frb-M* and *Frb-L*. Filtering of nodes based on their degree (Q.28, Q.29, and Q.30) and the search for nodes with at least one incoming edge (Q.31) are proved to be extremely problematic almost for all databases apart from Neo4J and Titan (v.1.0). In particular for Sparksee these queries cause the system to exhaust the entire available RAM and swap space on all Freebase subsamples (this has been linked to a known problem in the gremlin implementation). BlazeGraph fails also these last queries on all the Freebase datasets, while ArangoDB fails it only on *Frb-M* and *Frb-L*, and OrientDB instead only on *Frb-L*.

6.3 Insertions, Updates and Deletions

For operations that add new objects (nodes, edges, or properties), tests show extremely fast performances for Sparksee, Neo4J (v.1.9), and ArangoDB, with times below 100ms, with Sparksee being generally the fastest (Figure 4(a)). Moreover, with the only exception of BlazeGraph, all databases are almost unaffected by the size of the dataset. We attribute this to the use of write-ahead logs, and the internal configuration of the adopted data-structures. BlazeGraph is instead the slowest with times between 10 seconds and more than a minute. Both versions of Titan are the second slowest with times around 7 seconds for insertion of nodes, and 3 seconds for insertion of edges or properties, while for the insertion of a node with all the edges (Q.7) it takes more than 30 seconds. Sparksee, ArangoDB, OrientDB, Sqlg, and Neo4J (v.1.9) complete the task in less than a second. OrientDB is among the fastest for insertions of nodes (Q.2) and properties on both nodes and edges (Q.5 and Q.6), but is much slower, with inconsistent behavior, for insertion of edges. Neo4J (v.3.0), is more than an order of magnitude slower than its previous version, with a fluctuating behavior that does not depend on the size of the dataset. Sqlg is among the fastest for insertions of nodes, and nodes alongside edges, while is much slower for all other queries. Similar results are obtained for the update of properties on both nodes and edges (Q.16, and Q.17), and for the deletion of properties on edges (Q.21).

The performance of node removal (Q.18) for OrientDB, Sqlg, and Sparksee seems highly affected by the structure and size of the graphs (Figure 4(b)). On the other hand, ArangoDB and Neo4J (v.1.9) remain almost constantly below the 100ms threshold, while Neo4J (v.3.0) completes all the deletions between 0.5 and 2 seconds. Finally, for the removal of nodes, edges, and node properties, Titan shows almost one order of magnitude improvement.

For creations, updates and deletions, as a whole, the fastest are Neo4J (v.1.9), with constant times below 100ms, and then Sparksee, but with quite a scale-sensitive behavior for edge-deletion, that is shared with OrientDB. ArangoDB is also consistently among the fastest, but its interaction through REST calls, and the fact that it does not support transactions, constitutes a bias on those results in its favor since the time is measured on the client side.

6.4 General Selections

With *read* queries, some heterogeneous behaviors start to show up. The search by ID (Figure 5(b)) differs significantly from all other queries in this class. BlazeGraph is again the slowest, with performances around 500ms for the search of nodes, and instead 4 seconds or more for the search of edges. All other systems take less than 400ms to satisfy both queries, with Titan the slowest among them. Here Sparksee, OrientDB and Neo4J (v.1.9) return in about 10ms, hinting to the fact that, given the ID, they are able to jump immediately to the right position on disk where to find it.

In counting nodes and edges (Q.8, and Q.9), Sparksee has the best performance followed by Neo4J (v.3.0). As a matter of fact Sparksee and Neo4J (v.3.0) complete the two tasks in less than 10 seconds on all sizes of Freebase, while Neo4J (v.1.9) take more than an minute on the *Frb-L*. For BlazeGraph and ArangoDB, node counting is one of the few queries in this category that complete before timeout. In particular in Q.8 BlazeGraph is faster than ArangoDB, but then it hits the time limit for Q.9 on all Freebase subsamples, while ArangoDB, at least for *Frb-S* it's able to get the answer in time also on the other queries. Edge iteration, on the other hand, seems hard for ArangoDB that rarely completes within 2 hours for the Freebase datasets.

Computing the set of unique labels (Q.10) changes a little the ranking. Here, the two versions of Neo4J are the fastest databases, while Sparksee gets a little slower. The search for nodes (Q.11) and edges (Q.12) based on property values performs similar to the search for edges based on labels (Q.13), for almost all databases. These 3 are some of the few queries where the RDBMS-backed Sqlg works best, with results an order of magnitude faster than the competition. Among the others, Neo4J (v.3.0) gives the shortest time, with the Q.13 performing slightly faster than the others, getting an answer in a little more than 10 seconds on the larger dataset, while Neo4J (v.1.9), Sparksee, and OrientDB are at least one order of magnitude slower. Only for Sparksee we notice relevant differences between Q.12 and Q.13. Hence, equality search on edge labels is not really optimized in the various systems.

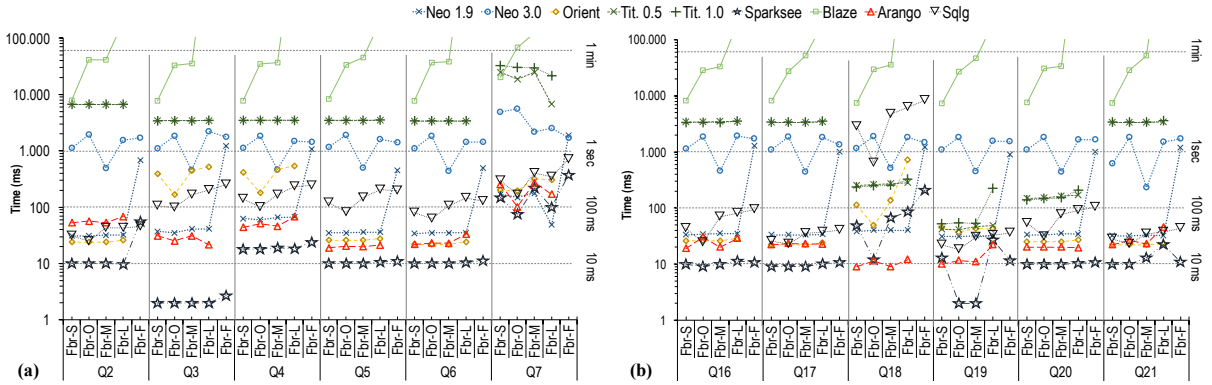

Figure 4: Time required for (a) insertions and (b) updates and deletions.

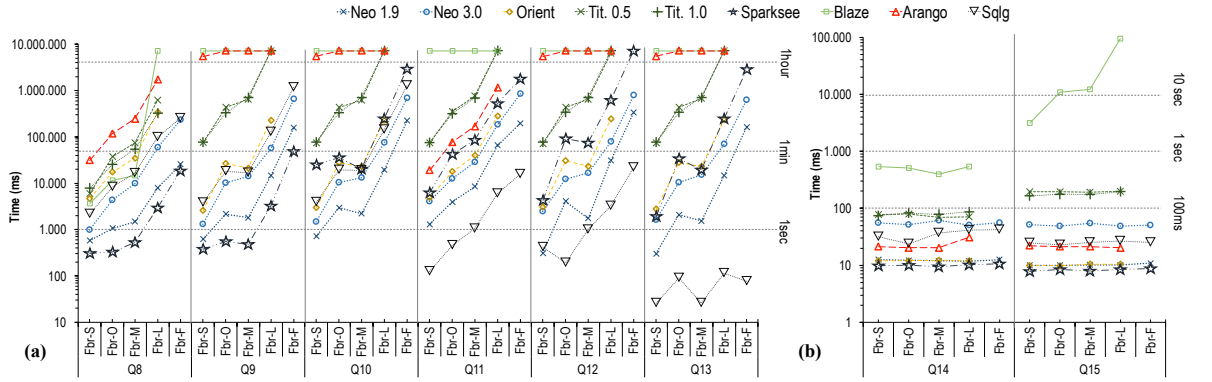

Figure 5: Selection Queries: The Id-based (right) perform orders of magnitude better than the rest (left)

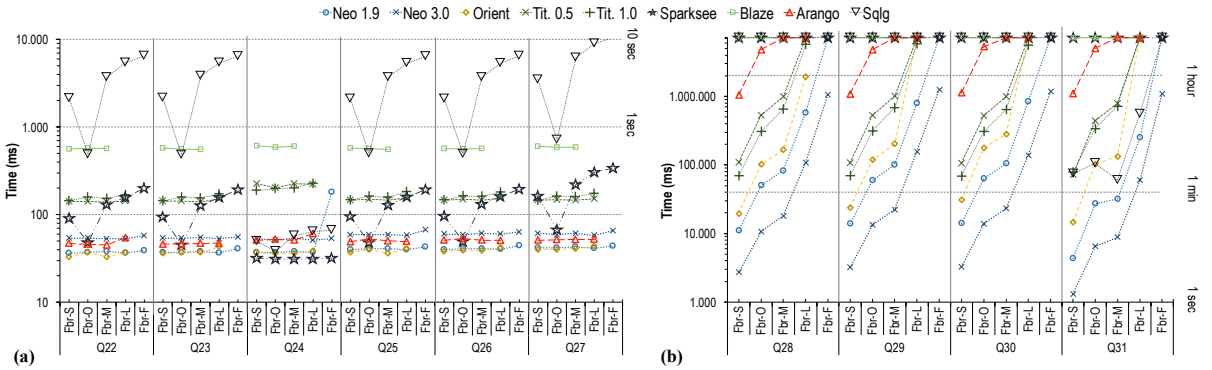

Figure 6: Time required for traversal operations: (a) local access to node edges, and (b) filtering on all nodes

6.5 Traversals

As mentioned above, the most important class of queries that GDBs are called to satisfy regards the *traversal* of graph structures. In the performance of traversal queries that access the direct neighborhood of a specific node (Q.22 to Q.27), we observe (Figure 6(a)) that OrientDB, Neo4J (v.1.9), ArangoDB, and then Neo4J (v.3.0) are the fastest, with response times below the 60ms, and being robust to the size and structure of the dataset. Sparksee seems to be more sensitive to the structure and size of the graph, requiring around 150ms on *Frb-L*. The only exception for Sparksee is when performing a visit of the direct neighborhood of a node filtered by the edge labels, in which case it is on par with the former systems. BlazeGraph is again an order of magnitude slower (∼600ms) preceded by Titan (∼160ms). We notice also that Sqlg is the slowest engine for these queries, unless a filter is posed on the label to traverse, in which case Sqlg becomes much faster.

When comparing the performance of queries Q.28 to Q.31 that traverse the entire graph filtering nodes based on the edges around them, as shown in Figure 6(b), the clear winner is Neo4J (v.3.0), with its older version being the second fastest. Those two are also the only two engines that complete the query on all datasets. In particular Neo4J (v.3.0) completed each query on *Frb-L* in less than two minutes on average, while Neo4J (v.1.9) took at lest 10 minutes for the

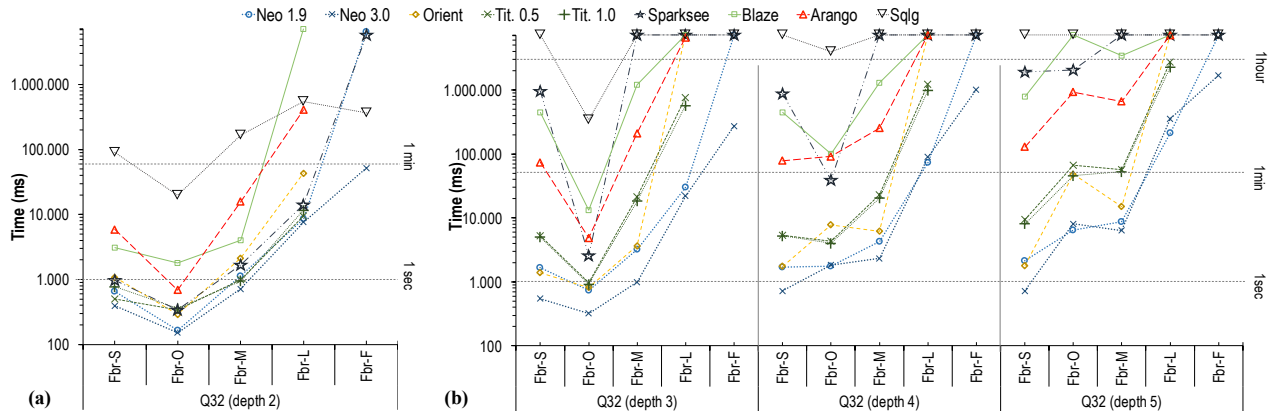

Figure 7: Time required for breadth-first traversal (a) at depth= 2, and (b) at depth>= 3

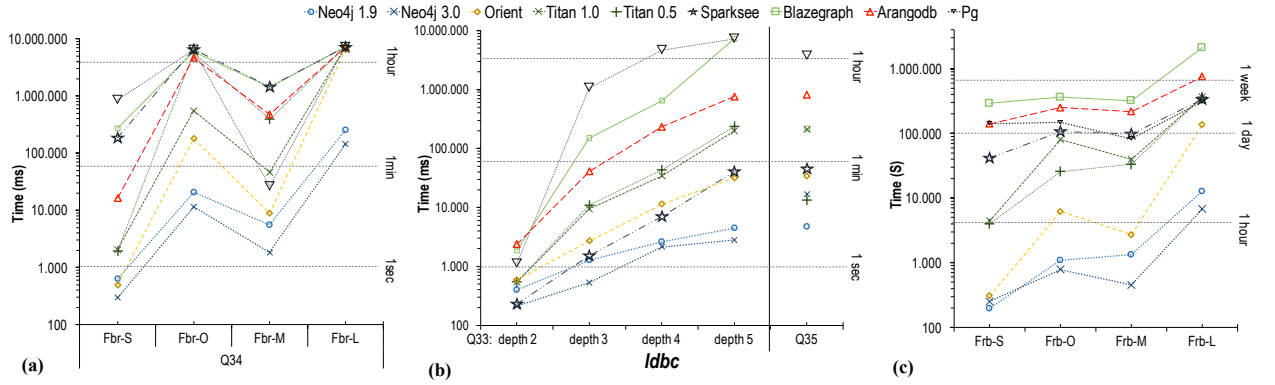

Figure 8: Performance of (a) Shortest Path, (b) label-constrained BFS and Shortest Path, and (c) Overall

same dataset. All tested systems are obviously affected by the number of nodes and edges to inspect. Sparksee is unable to complete any of these queries on Freebase due to the exhaustion of the available memory, indicating probably a problem in the implementation, as this never happens in any other case. BlazeGraph as well hits the timeout limit on all samples, while ArangoDB is able to complete only on *Frb-S* and *Frb-O*. Finally Sqlg is able to complete only Q.31, although with time comparable to Neo4J (v.1.9). Nevertheless, all systems complete the task on *Yeast*, *ldbc* and *MiCo*.

We study breadth-first searches (Q.32 and Q.33) and shortest path searches (Q.34 and Q.35) separately from the other traversal operations. The performance of the unlabeled version of breadth-first-search, shown in Figure 7, highlights once more the good scalability of both versions of Neo4J at all depths. Although Neo4J (v.3.0) is the only system to complete the task before timeout even on the *Frb-F*. OrientDB and Titan give the second fastest times for depth 2, with times 50% slower than those of Neo4J. For depth 3 and higher, as Figure 7(b) illustrates, OrientDB is a little faster than Titan. On the other hand, in these queries we observe that Sqlg and Sparksee are actually the slowest engines, even slower than BlazeGraph. For query Q.34 in Figure 8(a), which is the shortest path with no label constraint, the performance of the system is similar to the above, BlazeGraph and Sparksee are in this case very similar, and Sqlg still the slowest.

The label-filtered version of both the breadth first search and the shortest path query on the Freebase samples (not shown in a figure) were extremely fast for all datasets because the filter on edge labels cause the exploration to stop almost immediately. Running the same queries on *ldbc* we still observe (Figure 8(b)) that Neo4J is the fastest engine, but also Sparksee is the second fastest in par with OrientDB for the breadth-first search, while on the shortest path search filtered on labels, Titan (v.1.0) gets the second place.

6.6 Effect of Indexing.

We built node-attribute indexes on the graphs in the various systems to evaluate the effect of indexing on the system performance (Figure 9, and 10). BlazeGraph has been excluded since it does not allow any custom index (and the system already builds its own). ArangoDB showed no difference in running times, so we suspect some defect in the gremlin implementation. For insertions, updates, and deletions, we noticed longer running times, as expected since the indexes had to be updated, but was no more than 10% in most cases. The only exception to this trend are Neo4J (v.3.0) and OrientDB, with delays of about 30% and 100% respectively. Despite the increase in time, Neo4J (v.1.9), Sparksee, and OrientDB remained the fastest systems for *CUD* operation. For search queries (Q.11), the presence of indexes gave to Neo4J (v.1.9), OrientDB, Titan (v.0.5), and Titan (v.1.0) an improvement of 2 to 5 orders of magnitude (depending on the dataset size), while Sqlg witnessed up to a 600x speed up. Sparksee and Neo4J (v.3.0) instead were not able to take advantage of the indexes. As a matter of fact, both system support labels not only for edges but also

for nodes. For this reason, for both systems, indexes are tied to a specific node label, and hence only queries specifying a selection on a node label can then exploit the index for the attribute. This means that indexes play a significant role in most of the systems, and are taken seriously into consideration in query execution.

6.7 Complex Queries.

The complex queries were executed in the presence of indexes in the various systems. Figure 11 illustrates their performance. ArangoDB and Titan (v.0.5) were the slowest, while BlazeGraph didn't manage to complete the workload since the initial queries timed out. Most likely all these three systems have a focus only on primitive operators. Titan (v.1.0), despite being among the slowest for most queries, resulted by far the fastest for the portion of operations that required a short-distance traversal restricted on a single specific label. Suggesting this newer version has some optimization functionality indicating that its developers are heading towards that direction. Neo4J (v.3.0) is very fast (possibly as a result of being fast in primitive operations execution). Yet, it is not as good as Sqlg. Sqlg seems to be taking advantage of the relational optimizer in pushing down selection predicates or exploiting indexes, which in combination with the fact that the queries can be easily translated to conditional join queries, with no recursion and short join chains, make it the fastest system.

6.8 Single vs Batch Execution.

We looked at the times differences between single executions (run in isolation) and batch. We report times for each batch execution for *Frb-S*, *Frb-O*, *Frb-M*, and *Frb-L* in Figures 12, 13, 14, and 15. Running the queries in batch mode does not create any major changes in the way the systems compare to each other. For the retrieval queries, the batch requests of the 10 queries were taking exactly 10 times the time of one iteration, i.e., no benefit obtained from the batch execution. Exception is for queries 14 and 15 (Figure 13 b), here times to retrieve 10 nodes by their internal IDs are almost exactly the same as for retrieving one single node (see Figure 5 above). Such behavior suggests that the systems load the data into main memory at the first call, and then retrieves everything from there.

Instead, for the create, update and delete operations, the batch is less than 10 times the time needed for one iteration, meaning that in single mode most of the time we measure is some initiation set-up time for the operation. For traversal queries the batch executions only stressed the differences between faster and slower databases.

6.9 *Yeast*, *MiCo*, and *ldbc*

In the following we report on the results of the tests performed on the *Yeast*, *MiCo*, and *ldbc* datasets, which are generally smaller than the Freebase samples, and also have a much smaller number of edge labels. Results for queries in isolation mode are reported in Figure 16, 18, 20, and 22, while results for the batch mode execution are in Figure 17, 19, 21, and 23. Experiments on these datasets, as noted above, show again similar relative performances compared to the results on the Freebase samples described earlier. In general we see Sparksee performing among the fastest databases more often. ArangoDB's performance as well is much more similar to the other systems. BlazeGraph instead is usually the slowest also on those datasets. As a matter of fact, even in tests with *Yeast*, BlazeGraph is not always able to terminate queries within the timeout limit, which indicates some serious implementation problems for some of the selection queries (Figure 18).

6.10 Overall Evaluation and Insights

To sum up the evaluation we can compare the cumulative time taken by each system to complete the entire set of queries in both single and batch executions (Figure 8(c)). Overall Neo4J is the fastest engine. An important change that took place between the old implementation and the new for the gremlin API, is due to a different licensing for Tinkerpop. In particular Neo4j is distributed under the GPL V3 license[6], while Tinkerpop is now distributed under the Apache licence[7]. This forced the developers of Neo4j to add an additional indirection layer on top of their library, the effect of which is evident for very fast operations (e.g., **C**, **U**, **D**). Nonetheless, on the most time-consuming queries for class **R** and **T** Neo4J (v.3.0) is usually the fastest, and Neo4J (v.1.9) the runner-up. Pretty good running times have also been recorded for OrientDB, which is often on par with Neo4J, and in cases is better than one of its two versions. It does not, however, do well in cases where large portions of the graph have to be processed and kept in memory, e.g., with *Frb-L*. Titan results quite often one order of magnitude slower than the best engine. It shows difficulties in create and update operations, however, it is much better in deletions, most likely due to the *tombstone* mechanism, where it marks an item as removed instead of actually removing it. Nonetheless, this method seems to result slower than the write ahead log (WAL) adopted by the others. Sparksee gives almost consistently the best times in the operations for creating, updating, and deleting objects. Although it is not very fast with deletions of nodes having a lot of edges, it is still better than others. It also performs best in edge and node counts, as well as in retrieval of nodes and edges by ID, thanks to its internal compressed datastructures. Nevertheless, it performs worse than others for the other retrieval queries and is the worst in most traversals, showing good performances only when a filter on edge labels was applied. Finally, it gives a lot of timeouts on the degree-based node search queries. ArangoDB excels only in few queries. For creation, updates and deletes, it ranks among the best. For retrievals, its performance is in general poor, except when searching by ID, while for traversals it has a narrow lead over Sparksee and BlazeGraph. This is due to the way Gremlin primitives are translated into the engine, where it has to materialize all objects in order to iterate through them.

Sqlg shows the expected low performance for all the traversal operations due to the need to traverse the graph via relational joins instead of direct links to node/edges. For queries containing 1 or 2 hop traversal restricted to a single edge-label, however, it performs extremely well, since joins are limited to a single table containing only the necessary edges. BlazeGraph results also in a generally poor performance. The indexes it builds automatically do not seem to help much, and most likely is optimized for SPARQL queries only and not for a generic graph management. This is probably due to its internal structure, since in RDF attributes

[6] `www.gnu.org/licenses/quick-guide-gplv3.html`
[7] `www.apache.org/licenses/GPL-compatibility.html`

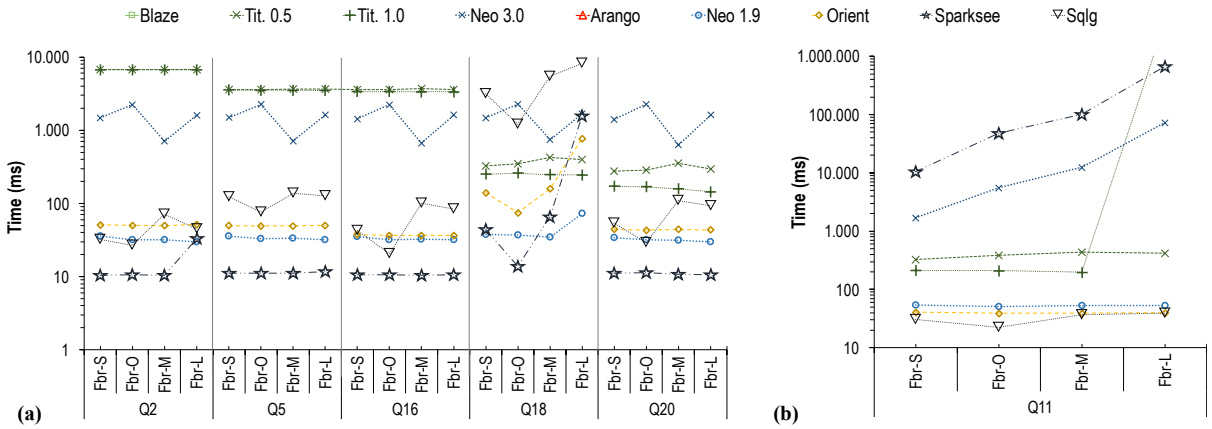

Figure 9: Effect of indexing on the Time required for Q.2, Q.5, Q.16, Q.18, Q.20, and Q.11

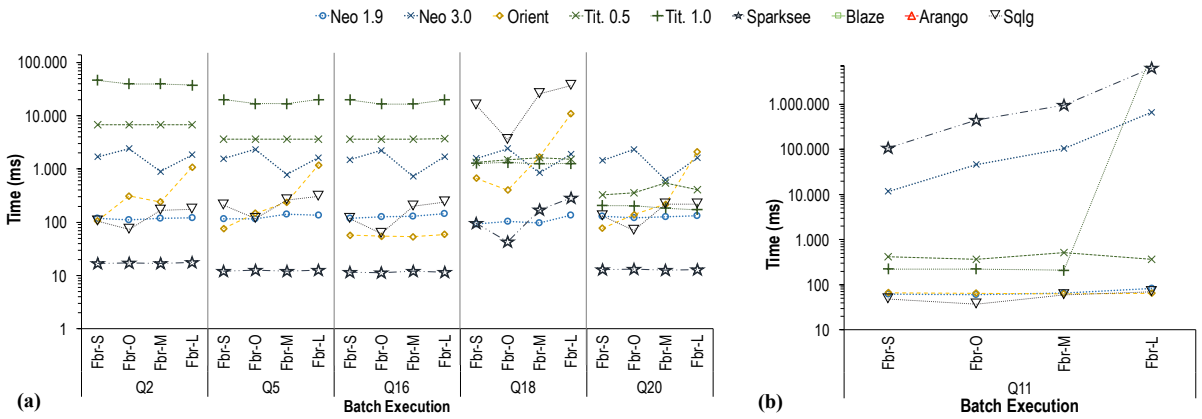

Figure 10: Effect of indexing on the Time required in Batch Mode for Q.2, Q.5, Q.16, Q.18, Q.20, and Q.11

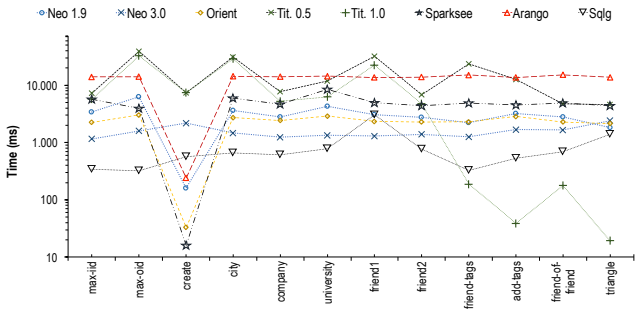

Figure 11: Complex Query Performance

are generic triples, similar to predicates, so it is possible that it is iterating over the whole dataset. A summary of the ranking for each query type is presented in Table 4.

Although we considered all the queries equally important, for different applications some may be playing a more important role. We have identified three main uses of graph databases: interactive, exploratory and business analytics. The weight of each query for each such use, as produced in collaboration with some system developers, can be found in Table 5 and can be used to determine which weight each query performance should have for specific workloads.

The experiments have also shown that hybrid and native systems perform differently. Native are the systems built from top to bottom for graph data, e.g, the Neo4J, OrientDB, and Sparksee. Hybrid are those based on other type of systems to offer graph data functionality. Examples of the latter include ArangoDB, which is based on a document-store, Sqlg, which is backed by a RDBMS, Titan, which relies on a column-store, and BlazeGraph which is built on top of an RDF engine. For a limited set of use cases hybrid systems perform equally well with the native, but for more advanced queries, like finding connectivity between two nodes, unbounded traversal and enumeration of structures, the hybrid systems under-perform significantly.

Regarding the query language, Gremlin is the one supported by all the systems. Nevertheless, each system offers its own native query language and performs all the optimizations on it. Gremlin queries have to be translated to the native primitives, loosing this way many of the possible optimizations. Gremlin may be used as a standard, but is not the first priority of the systems. The fact that data loading was not possible through gremlin but through native calls, is another indication of this. Therefore, this gap needs to be bridged, for instance, by means of a syntactic translation from gremlin syntax to the various declarative languages. Otherwise, an effort to standardize a declarative language for graph traversal could be more effective.

By comparing the methodologies of the micro and macro-benchmarking, it became clear the importance of studying individual operators in a context-agnostic way. The micro-benchmark evaluation has pointed out many specific problems that could successfully be communicated to the ven-

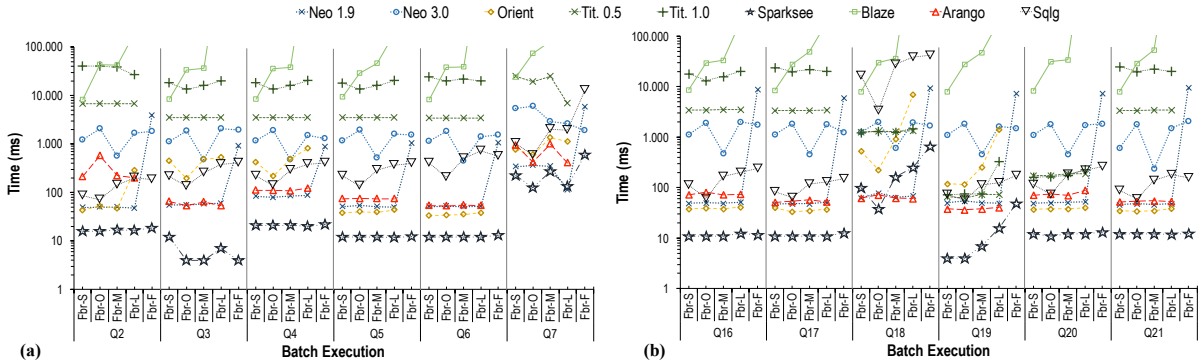

Figure 12: Time required in Batch Mode for (a) insertions and (b) updates and deletions.

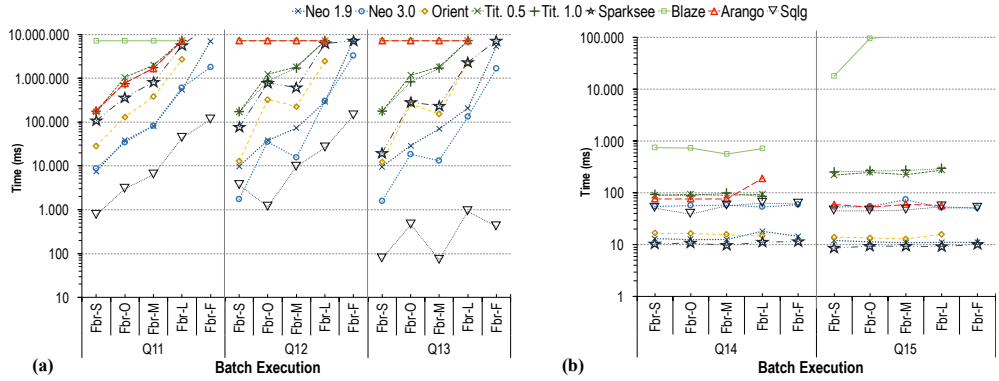

Figure 13: Selection Queries in Batch Mode: The Id-based (right) perform orders of magnitude better than the rest (left), and compared to the isolation mode they take the same amount of time

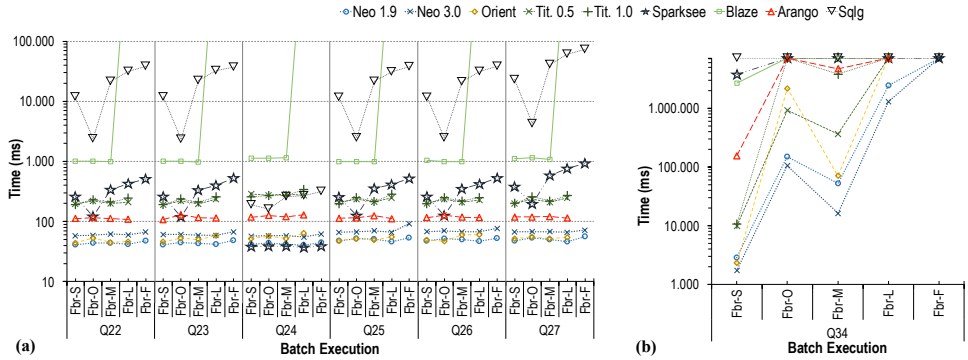

Figure 14: Time required for traversal operations in Batch Mode: (a) local access to node edges, and (b) for shortest path search

dors, while with the macro-benchmarking a deeper analysis had to be performed.

7. EXPERIENCES

In general our experiences cover a large spectrum of issues, technical challenges that we faced, and areas of improvement that are related to the usability of the various systems.

Installation, Configuration, Documentation and Support. First we stress that the only 2 systems that we were able to install and run as expected were Neo4J and Sparksee. For those, after downloading the relevant binaries and following the instructions provided on the respective websites, we were almost immediately able to load our datasets, at all sizes, and run some queries. For the others, as mentioned earlier (and below) we had to overcome some difficulties in importing the datasets, configuring the systems properly, and understanding the errors raised when running some of the queries. As a result, for those system that are open-source and hosted on a public repository, we reported those problems and bugs found as issues. In total we issued 8 support request (comprising bug issues) for ArangoDB, 4 for

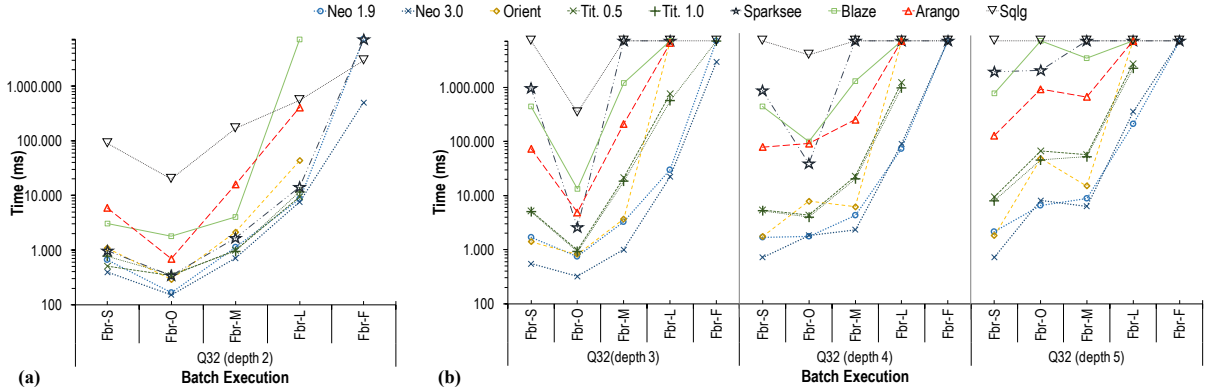

Figure 15: Time required for breadth-first traversal in batch mode (a) at depth= 2, and (b) at depth>= 3

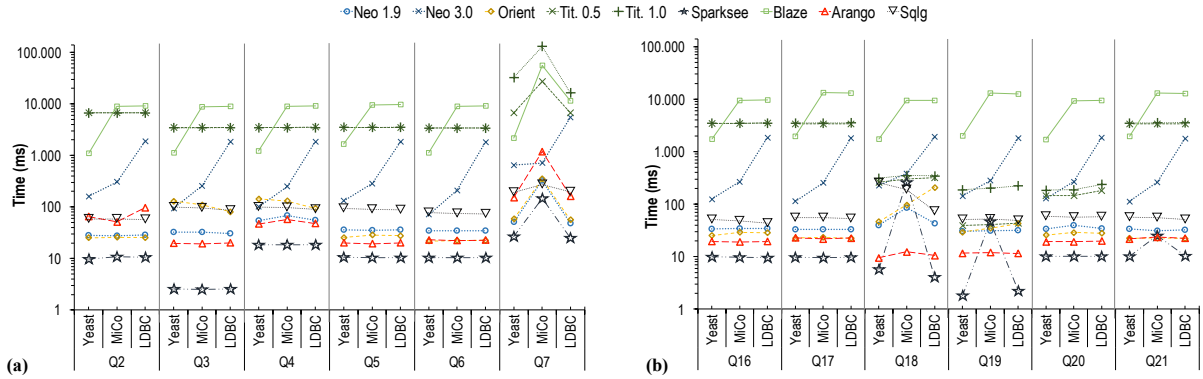

Figure 16: Time required on *Yeast*, *ldbc*, and *MiCo* for (a) insertions and (b) updates and deletions in isolation mode.

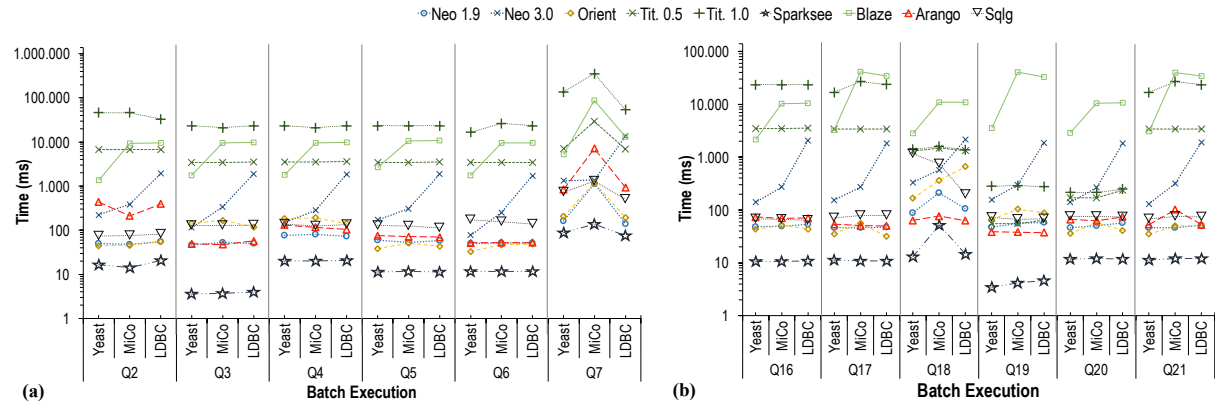

Figure 17: Time required for (a) insertions and (b) updates and deletions in batch mode for *Yeast*, *ldbc*, and *MiCo*.

OrientDB, 2 for Titan, 2 for Sqlg, and 1 for BlazeGraph.

For ArangoDB and OrientDB some of those bugs have been fixed in official releases of the software or have been included in the development road-map. Instead those regarding Titan and BlazeGraph didn't receive any reply from the developers (in many months) and, where possible, were either fixed or circumvented in our local installs. This also describes the level of support received by the respective development teams.

Regarding the documentation, we note that Neo4J, Sqlg and OrientDB are provided with pretty in-depth informations for developers. Sparksee, Titan and ArangoDB have some documentation, limited in some aspects, but still clear for basic installation, configurations and operational needs. Among those Titan manual contains a lot of confusion among the various existing software versions, and in some cases, the provided instructions and example-code are not actually self-contained. Also, given the reliance on Cassandra for the storage, it is to note the reduced amount of information on how to properly configure this system and how to tackle the

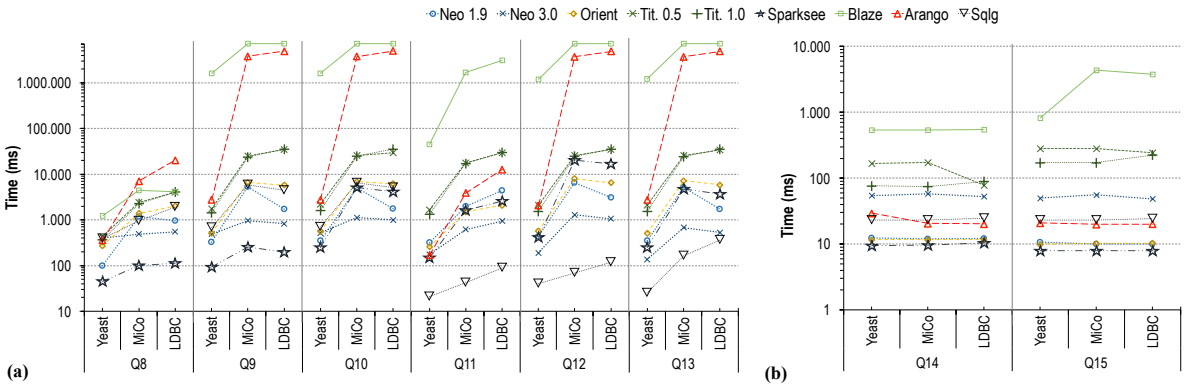

Figure 18: Selection Queries in isolation mode for *Yeast*, *ldbc*, and *MiCo*: The Id-based (right) perform orders of magnitude better than the rest (left)

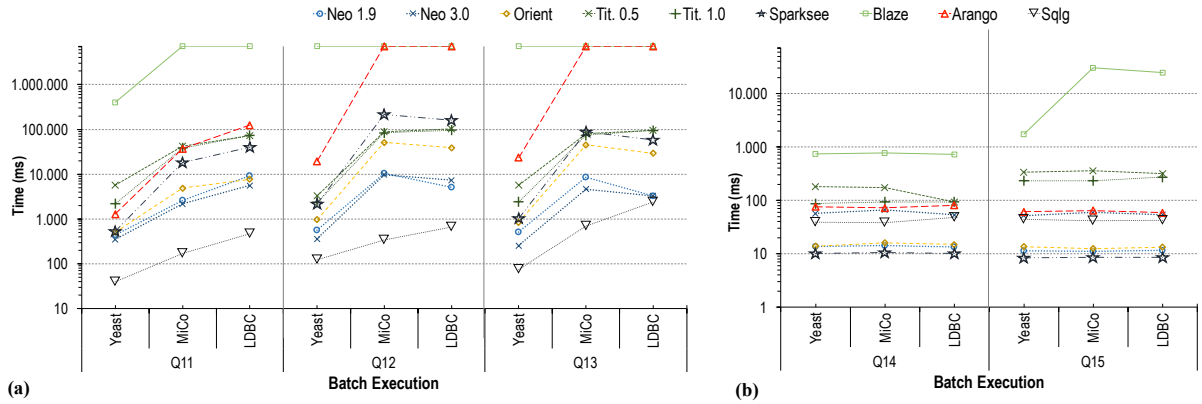

Figure 19: Selection Queries in Batch Mode for *Yeast*, *ldbc*, and *MiCo*: The Id-based (right) perform orders of magnitude better than the rest (left), and compared to the isolation mode they take the same amount of time

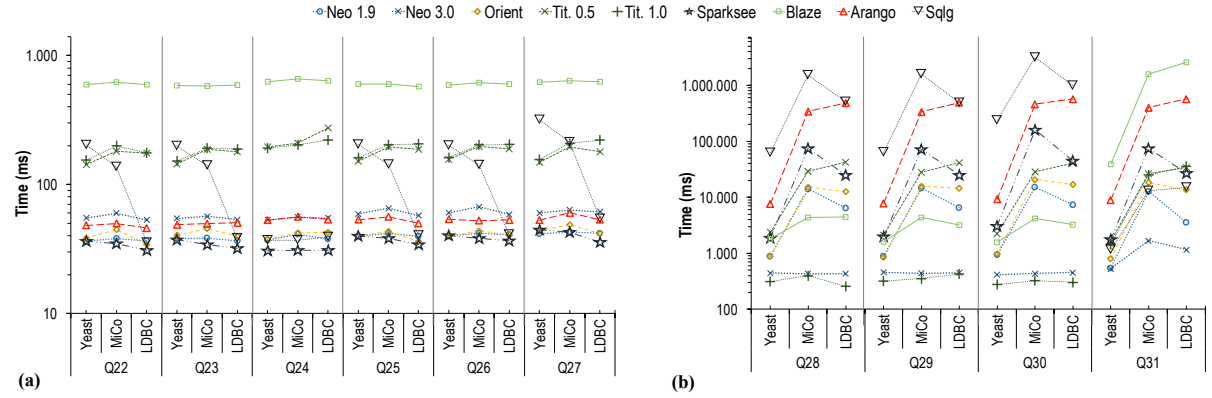

Figure 20: Time required for traversal operation for *Yeast*, *ldbc*, and *MiCo*: (a) local access to node edges, and (b) global filtering of nodes based on degree.

various problems arising with it. BlazeGraph's documentation, instead, is largely outdated. Also, even though the system relies a lot on the user for proper configuration, the information provided is generally cryptic.

Regarding the configuration of the other systems, we report that Neo4J doesn't require any specific configuration. OrientDB instead supports by default a number of edge labels at most equal to 32676 divided by the number of cores in the machine (e.g., 4084 edge labels on a 8 cores machine), for supporting more labels, it requires a special feature to be disabled. ArangoDB requires two configurations, one for the engine, and one for the V8 javascript server, the second regards the level of logging of the system. Without proper configuration (with only default values) this system gener-

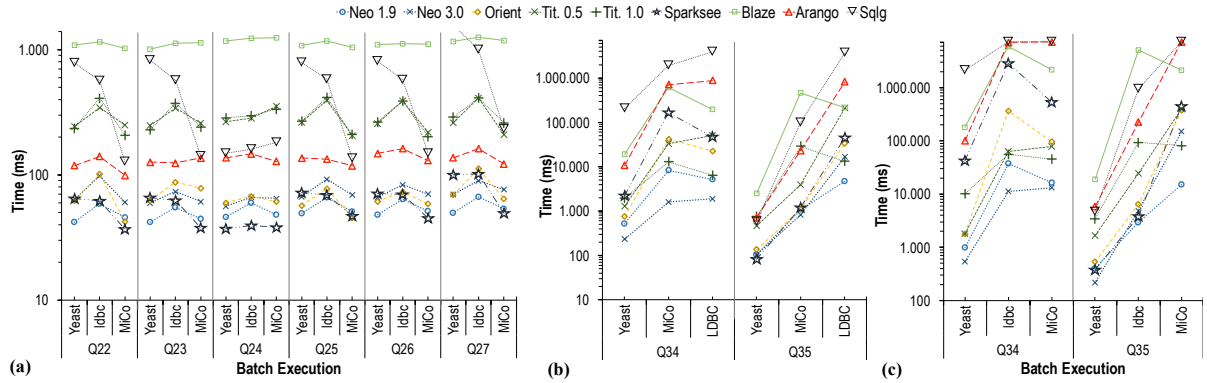

Figure 21: Time required for traversal operations for *Yeast*, *ldbc*, and *MiCo*: (a) local access to node edges in batch mode, and for shortest path search (b) in isolation, and (c) in batch mode.

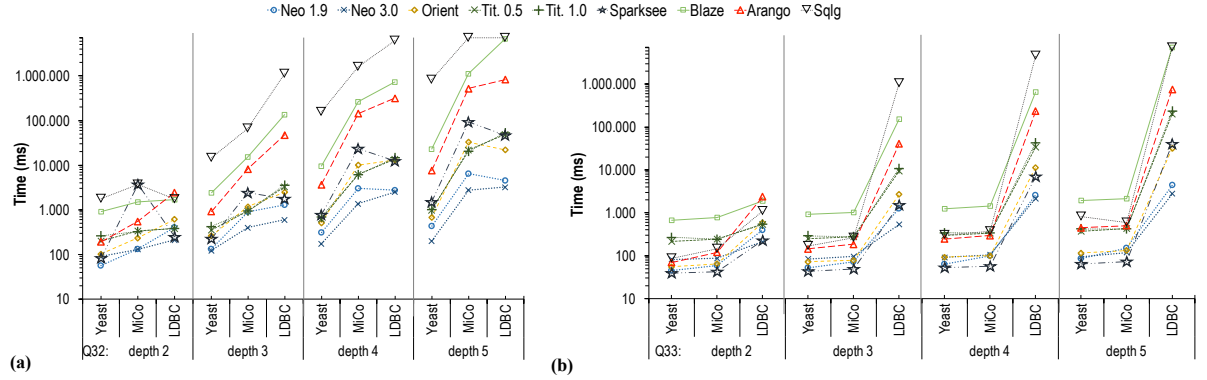

Figure 22: Time required for breadth-first traversal with (a) and without (b) label filtering for *Yeast*, *ldbc*, and *MiCo*.

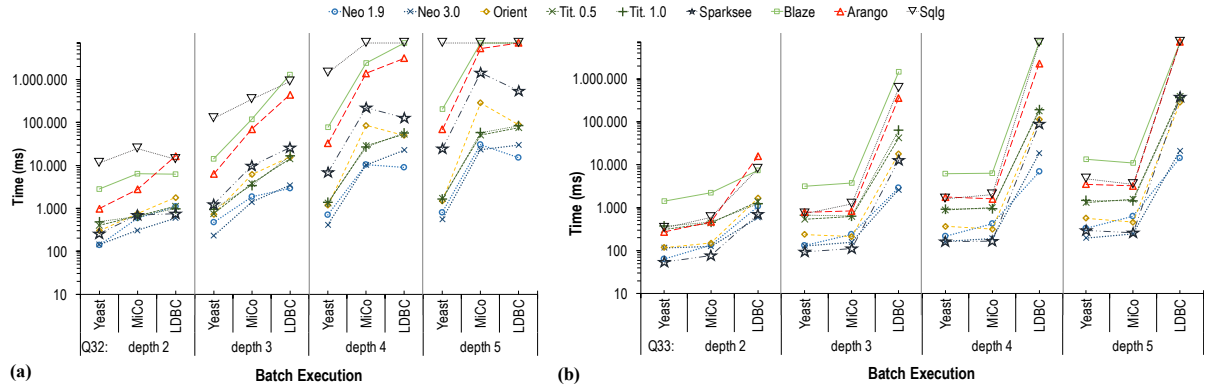

Figure 23: Time required for breadth-first traversal with (a) and without (b) label filtering in batch mode for *Yeast*, *ldbc*, and *MiCo*.

ated 40 GB of log files in about 24 hours of activity, with a single active client. For Titan instead the most important configurations are for the JVM Garbage Collection and for the Cassandra backend. Additionally, with large datasets, it is necessary to disable automatic schema creation, and to create instead the schema manually before loading the data.

All systems based on Java, were also extremely sensitive to the effect of the garbage collection routines. When dealing with data-intensive applications and a large amount of main-memory, it is necessary to provide a customized configuration to the JVM, yet, none of the systems provide clear instructions on how to tune it properly for their needs, but they only propose generic advices.

Finally we report on the Tinkerpop/Gremlin documentation. For version 2.6 the list of supported methods with some examples are provided [8], for version 3 the official manuals are much more extended [9], although not to the benefit of clarity. In this sense, we also hope that the code of the queries implemented in this study serve as more concrete tutorial for understanding the basics of the Gremlin language.

Loading problems. As already mentioned, we encountered a great deal of issues when trying to load the datasets in some of the databases tested. ArangoDB in particular,

[8] `gremlindocs.spmallette.documentup.com`
[9] `http://tinkerpop.apache.org/docs/current/reference/`

Table 5: Relative importance of queries.

Q.#	Interactive	Exploratory	Business Analytics
LOADING			
1.	3	**4**	1
CREATE			
2.	2	1	3
3.	3	1	**4**
4.	2	1	3
5.	**4**	1	**4**
6.	2	1	3
7.	**4**	1	**4**
READ			
8.	3	**4**	1
9.	3	3	1
10.	3	3	2
11.	**4**	**4**	**4**
12.	2	2	2
13.	2	3	2
14.	**4**	2	**4**
15.	1	1	3
UPDATE			
16.	**4**	2	**4**
17.	2	**4**	3
DELETE			
18.	3	1	3
19.	**4**	2	**4**
20.	**4**	2	3
21.	2	1	2
TRAVERSALS			
22.	**4**	**4**	**4**
23.	**4**	**4**	**4**
24.	**4**	**4**	**4**
25.	3	3	2
26.	3	3	2
27.	3	3	2
28.	1	3	2
29.	1	3	2
30.	1	3	2
31.	2	3	1
32.	**4**	3	2
33.	**4**	**4**	2
34.	2	3	2
35.	3	**4**	2

1 = Low importance, 4=Critical Importance

when using Gremlin for loading, sends each node and edge insertion instruction separately to the server (in a HTTP call). This method results too slow, even for small datasets, so that we were forced to use some routines provided by the back-end system itself. For BlazeGraph, with the exception of the smallest datasets, we had to activate a specific *bulk loading* feature otherwise we were facing loading times in the order of days. OrientDB as well required us to pass through some server-side implementation-specific commands in order to load the datasets. In particular, it didn't support non-alphanumeric characters in edge-label, and for the Freebase samples we had to disable some features that were limiting the maximum number of edge-labels. Also for Sqlg we re-encoded all edge labels to unique hashes that did not exceed the 63 characters limit that Postgresql imposes, after that the loading proceeded without any major issue. Finally, Titan (in both versions) for any medium to large sized datasets requires disabling the automatic schema creation during loading, otherwise its storage back-end (Cassandra) would get swamped with extra consistency check operations. This means that the complete schema of the graph, in terms of node and edge labels and properties, should be known to the system prior to the insertion of the data, the same way one should declare the schema in a relational database before loading any data. This required us to issue a set of instructions, before loading the data, to create such schema.

Queries, Groovy, and Gremlin. Last, we report that using Groovy as support language for Gremlin was quite problematic in some cases. As a matter of fact the Groovy language has dynamic types, and uses type inference along with peculiar handling of variable scope. As a result, explicit type casting is needed when providing the values to queries in some systems, especially with numbers. For example, in Sparksee if one attribute is of *Long* type and size (i.e., larger than a 32 bit number), then all values for the attributes with the same name need to be passed and queried as *Long* values, otherwise values compatible with the *Integer* type will be treated as such, and the search will result in a mismatch, independently of the value they represent. For Titan, instead, when not provided by the schema declared a priori, each value should be inserted as the smaller available type, i.e., if a number is within the integer range, it should be converted to the integer type. With the other systems instead, types are handled transparently for the user, and work without explicit type casts.

A second problem with Gremlin 2.0 is the lack of explicit operations for pattern-matching queries and shortest paths queries. In the new version (Gremlin 3.0) a new '*match*' operator is introduced, but there is still no operator for the shortest-path search. Both types of queries could be implemented with the composition of basic constructors (although for weighted shortest path the implementation would be extremely hard), while would be better to have an abstract operator in the language and leave to the engine the implementation of advanced and optimized algorithms.

Finally, Gremlin doesn't provide a way to handle indexes, this as well is a limitation of the language that requires for the user to access directly the back-end system.

8. CONCLUSIONS

We performed an extensive experimental evaluation of the state-of-the-art graph databases. We scaled to levels that have not been considered before, and included systems that have not been previously considered. Furthermore, we provided a principled and systematic evaluation methodology based on micro-benchmarking that contains 35 different operations. We also described the challenges we faced in loading the large datasets and running the queries, and how we overcame these challenges. We materialized our methodology into a suite that we made available on-line [43]. It includes, scripts, datasets, and queries, among any other interesting material. To the best of our knowledge, our study is the most complete and up-to-date study of graph databases available nowadays. Apart from the direct benefits, our work can complement studies on the different (but highly related) graph analytic systems.

One of the features not tested yet is parallelism, which is part of our future work. In fact all our experiments were conducted on a single machine, not exploiting any of the

parallel features that almost all system provide. Nevertheless, it is important to notice that many systems advertise their ability to scale to multiple machines more than other features, but they seemed unable to exploit at best the resources of a single machine. In particular, in some cases even simple queries for relatively small db sizes were taking 2 or more hours to complete.

Acknowledgments: The current work has been partially supported by the Keystone EU Cost Action, a Jean d'Alembert Fellowship, and the IEEE Smart Cities Initiative.

We would like to thank Sparsity Technologies for providing us with a Sparksee license. We also thank Michael Hunger at Neo4j, Dàmaris Coll at Sparsity Technologies, and Luca Garulli at OrientDB LTD, for the extended assistance with our experiments and for providing valuable feedbacks to this work. We thank Liviu Alexandru Bogdan for the help in the early stages of this work, and we would like to thank also Jonathan Ellithorpe for helping us with the LDBC data generator, whose code we used to parse and import the data into the GraphSON format.

References

[1] Apache Cassandra. http://cassandra.apache.org.

[2] Apache Hbase. http://hbase.apache.org.

[3] Apache Lucene. http://lucene.apache.org.

[4] Apache Mesos. http://mesos.apache.org.

[5] Apache tinkerpop. http://tinkerpop.apache.org/.

[6] Arangodb. https://www.arangodb.com/.

[7] BerkeleyDB. http://www.oracle.com/technetwork/products/berkeleydb.

[8] Docker inc., docker. https://www.docker.com/.

[9] Elasticsearch. http://www.elastic.co/products/elasticsearch.

[10] Infinitegraph. http://www.objectivity.com/products/infinitegraph.

[11] Neo technology, inc., neo4j. http://neo4j.com.

[12] Ontotext graphdb. http://graphdb.ontotext.com/.

[13] Orient technologies, orientdb. http://orientdb.com/orientdb/.

[14] Sparsity technologies, sparksee. http://www.sparsity-technologies.com/.

[15] Systap, llc., blazegraph. https://www.blazegraph.com/.

[16] Thinkaurelius, titan. http://titan.thinkaurelius.com/.

[17] B. Alexe, W. C. Tan, and Y. Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

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

[19] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma. The linked data benchmark council: A graph and rdf industry benchmarking effort. *SIGMOD Rec.*, 43(1):27–31, May 2014.

[20] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[21] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. In *GRADES*, pages 15:1–15:7, New York, NY, USA, 2013. ACM.

[22] Bast, Hannah and Baurle, Florian and Buchhold, Bjorn and Haussmann, Elmar. Easy access to the freebase dataset. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 95–98, New York, NY, USA, 2014. ACM.

[23] V. Batagelj and A. Mrvar. Yeast, pajek dataset. http://vlado.fmf.uni-lj.si/pub/networks/data/, 2006. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[24] C. Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, Jan. 2015.

[25] H. Boral and D. J. Dewitt. A methodology for database system performance evaluation. In *Proceedings of the International Conference on Management of Data*, pages 176–185, 1984.

[26] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.

[27] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *GRADES*, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[28] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[29] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 37–48, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Mach. Learn.*, 56(1-3):9–33, June 2004.

[31] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, Mar. 2014.

[32] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[33] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[34] Google. Freebase data dumps. https://developers.google.com/freebase/data, 2015.

[35] O. Goonetilleke, S. Sathe, T. Sellis, and X. Zhang. Microblogging queries on graph databases: An introspection. In *GRADES*, pages 5:1–5:6, New York, NY, USA, 2015. ACM.

[36] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. ACM, 2013.

[37] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, Aug. 2014.

[38] F. Holzschuher and R. Peinl. Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM.

[39] E. Ioannou, N. Rassadko, and Y. Velegrakis. On generating benchmark data for entity matching. *J. Data Semantics*, 2(1):37–56, 2013.

[40] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *Proceedings of the 2013 International Conference on Social Computing*, SOCIALCOM '13, pages 708–715, Washington, DC, USA, 2013. IEEE Computer Society.

[41] V. Kolomičenko, M. Svoboda, and I. H. Mlýnková. Experimental comparison of graph databases. In *IIWAS*, pages 115:115–115:124, 2013.

[42] M. Lissandrini. Freebase exq data dump. https://disi.unitn.it/~lissandrini/notes/freebase-data-dump.html, 2017.

[43] M. Lissandrini, M. Brugnara, and Y. Velegrakis. The Trento GDB Test Suite. https://disi.unitn.it/~lissandrini/gdb.html, 2017.

[44] M. Lissandrini, D. Mottin, T. Palpanas, D. Papadimitriou, and Y. Velegrakis. Unleashing the power of information graphs. *SIGMOD Rec.*, 43(4):21–26, Feb. 2015.

[45] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[46] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, Nov. 2014.

[47] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[48] P. Martin. Sqlg. http://www.sqlg.org/.

[49] N. Martínez-Bazan, M. A. Águila Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J.-L. Larriba-Pey.

Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, IDEAS '12, pages 110–119, New York, NY, USA, 2012. ACM.

[50] N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras. Dex: A high-performance graph database management system. In *ICDEW*, pages 124–127, Washington, DC, USA, 2011. IEEE Computer Society.

[51] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[52] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: A new way of searching. *The VLDB Journal*, 25(6):741–765, Dec. 2016.

[53] J. Okajima. Aufs: Advanced multi layered unification filesystem. `http://aufs.sourceforge.net/`.

[54] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. TR 1999-66, Stanford InfoLab, Nov.

[55] E. PrudâĂŹHommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[56] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *DBPL*, pages 1–10, 2015.

[57] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *HotCDP*, pages 2:1–2:5, 2012.

[58] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

[59] R. Tarjan. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING*, 1(2), 1972.

[60] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big graph analytics systems. In *SIGMOD*, pages 2241–2243, 2016.

[61] M. Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242, 1990.