# Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation

Matteo Lissandrini
Aalborg University
matteo@cs.aau.dk

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

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

## ABSTRACT

Despite the increasing interest in graph databases their requirements and specifications are not yet fully understood by everyone, leading to a great deal of variation in the supported functionalities and the achieved performances. In this work, we provide a comprehensive study of the existing graph database systems. We introduce a novel microbenchmarking framework that provides insights on their performance that go beyond what macro-benchmarks can offer. The framework includes the largest set of queries and operators so far considered. The graph database systems are evaluated on synthetic and real data, from different domains, and at scales much larger than any previous work. The framework is materialized as an open-source suite and is easily extended to new datasets, systems, and queries[1].

**PVLDB Reference Format:**
Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. Beyond Macrobenchmarks: Microbenchmark-based
Graph Database Evaluation. *PVLDB*, 12(4): 390–403, 2018.
DOI: https://doi.org/10.14778/3297753.3297759

## 1. INTRODUCTION

Graphs have become increasingly important for a wide range of applications [17, 41] and domains, including biological data [16], knowledge graphs [61], and social networks [31]. As graph data is becoming prevalent, larger, and more complex, the need for efficient and effective graph management is becoming apparent. Since graph management systems are a relatively new technology, their features, performances, and capabilities are not yet fully understood neither agreed upon. Thus, there is a need for effective benchmarks to provide a comprehensive picture of the different systems. This is of major importance for practitioners, in order to understand the capabilities and limitations of each system, for researchers to decide where to invest their efforts, and for developers to be able to evaluate their systems and compare with competitors.

[1]`https://graphbenchmark.com`

*Proceedings of the VLDB Endowment,* Vol. 12, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3297753.3297759

There are two categories of graph management systems that address two complementary yet distinct sets of functionalities. The first is that of *graph processing systems* [27, 42, 44], which analyze graphs to discover characteristic properties, e.g., average connectivity degree, density, and modularity. They also perform batch analytics at large scale, implementing computationally expensive graph algorithms, such as PageRank [54], SVD [23], strongly connected components identification [63], and core identification [12, 20]. Those are systems like GraphLab, Giraph, Graph Engine, and GraphX [67]. The second category is that of *graph databases* (GDB for short) [5]. Their focus is on storage and querying tasks where the priority is on high-throughput and transactional operations. Examples in this category are Neo4j [51], OrientDB [53], Sparksee [60] (formerly known as DEX), Titan [64] (recently renamed to JanusGraph), ArangoDB [11] and BlazeGraph [62]. To make this distinction clear, graph processing systems, can, in some sense, be seen as the graph world parallel to OLAP systems, while graph databases as the parallel to the OLTP systems.

Graph processing systems and their evaluation have received considerable attention [27, 32, 42, 44]. Instead, graph databases lag far behind. Our focus is specifically on graph databases aiming to reduce this gap, with a two-fold contribution. First, the introduction of a novel evaluation methodology for graph databases that complements existing approaches, and second, the generation of a number of insights on the performance of the existing GDBs. Some experimental comparisons of graph databases do exist [22, 37, 38]. However, they test a limited set of features providing a partial understanding of the systems, experiment at small scale making assumptions not verifiable at large scale [37, 38], sometimes provide contradicting results, and fail to pinpoint the fine-grained limitations that each system has.

Motivated by the above, we provide a complete and systematic evaluation of existing graph databases, that is not provided by any other existing work to date. We test 35 classes of operations with both single queries and batch workloads for a total of about 70 different tests, as opposed to the 4-13 that existing studies have done, and we scale our experiments up to 76M nodes/ 314M edges, as opposed to the 250K nodes/2.2M edges of existing works. Our tests cover all the types of insert-select-update-delete queries that have so far been considered, and in addition, cover a whole new spectrum of use-cases, data-types and scales. Moreover, we study workloads and datasets both synthetic and real.

**Micro-benchmarking.** In designing the evaluation methodology we follow a principled *micro-benchmarking* approach.

To substantiate our choice, we look at the test queries provided by the popular LDBC Social Network benchmark [26], and show how the produced results are ambiguous and limited in providing a clear picture of the advantages of each system. So, instead of considering queries with such a complex structure, we opt for a set of primitive operators. The primitive operators are derived by decomposing the complex queries found in LDBC, the related literature, and some real application scenarios. Their advantage is that they are often implemented by opaque components in the system, thus, by identifying the underperformed operators one can pinpoint the exact components that underperform. Furthermore, any complex query can be typically decomposed into a combination of primitive operations, thus, its performance can be explained by the performance of the components implementing them. Query optimizers may change the order of the basic operators, or select among different implementations, but the primitive operator performance is always a significant performance factor. This evaluation model is known as *micro-benchmarking* [15] and is similar to the principles that have been successfully followed in the design of benchmarks in many other areas [2, 22, 35, 36, 37, 38]. Note micro-benchamrking is not intended to replace macro-benchamrks. Macro-benchmarks are equally important in order to evaluate the overall performance of query planners, optimizers, and caches. They are, however, limited in identifying underperforming operators at a fine grain.

Our evaluation provides numerous specific insights. Among them, three are of particular importance: (i) we highlight the different insights that micro and macro benchmarks can provide; (ii) we experimentally demonstrate limitations of the tested hybrid systems when dealing with localized traversal queries that span across multiple long paths, such as the breadth-first search; and (iii) we identify the trade-offs between the logical and physical data organizations, supporting the choice of the native graph-databases we study to separate structural information from the actual data. For example, we found that storing nodes and edges as records, directly linked to each other, and with pointers to off-loaded structures for node attributes, seems to be the most effective organization for typical graph queries.

Note that the current work does not consider any distribution features. The focus is on single machine installation.

**Contributions.** Our specific contributions are as follows: **(i)** We explain the limitations of the existing graph database evaluations and clarify the motivations for the current evaluation study (Section 2); **(ii)** We provide an extensive list of primitive operations (queries) that graph databases should support (Section 4); **(iii)** We introduce the first thorough experimental evaluation methodology based on the micro-benchmarking model for Graph Databases (Section 5); **(iv)** We materialize the methodology into an open-source testing suite [40], based on software containers, that automates the addition of new systems, tests, and datasets; **(v)** We apply this methodology on the major graph databases available today, using different real and synthetic datasets — from co-citation, biological, knowledge base, and social network domains — and discuss our findings (Section 6).

## 2. RELATED WORK

**Evaluating Graph Processing Systems.** There is a great deal of works on evaluating graph processing systems [18, 32,43,45,68]. Such systems are designed for computationally expensive algorithms that often require traversing all the graph multiple times to obtain an answer, like page rank, or community detection. Such systems are very different in nature from graph database systems, thus, in their evaluation, "needle in the haystack" queries like those that are typical of transactional workloads are not considered. Of course, there are proposals for unified graph processing and database systems [27], but this idea is in its infancy. Our focus is not on graph processing systems or their functionalities.

**Evaluating Graph Databases.** In contrast to graph processing systems, graph databases are designed for transactional workloads and "needle in the haystack" operations, i.e., queries that identify and retrieve a small part of the data. Existing evaluation works [3, 5] for such systems are limited in describing the systems implementation, data modelling, and query capabilities, but provide no experimental evaluation. A different group of studies provides an experimental comparison but is incomplete and fails to deliver a consistent picture. In particular, one work [22] analyzes only four systems, two of which are no longer supported, with small graphs and a restricted set of operations. Two other empirical works [37, 38] compared almost the same set of graph databases over datasets of comparable small sizes, but agree only partially on the concluded results. Moreover, all existing studies do not test with graphs at large scale and with rich structures. Our work comes to fill exactly this gap in graph database evaluation, by providing the most extensive evaluation of the state of the art systems in a complete and principled manner.

**Distribution & Cluster Evaluation.** In the era of Big Data, it is important to understand the abilities of graph databases in exploiting parallel processing and distributed architectures. This has already been done in graph processing systems [32,43,66]. However, distributed data processing is out of the scope of the current paper, for a number of reasons. First, not all the systems support distribution in the same way, i.e., partitioning, replication, or sharding. Second, an evaluation of distribution capabilities to be complete would require to consider additional parameters like the number of nodes and concurrency level. Third, despite the popularity of distributed graph management systems, single machine installations are still a highly popular choice [58]. For these reasons, we consider the study of distribution as our natural follow up work, since the question about which system is able to scale-out better may only come after the understanding of its inherent performance [49, 57].

**Graph Benchmarks.** There is already a number of benchmarks [1, 4, 6, 26] for evaluating systems for RDF or social data. Yet, those benchmarks are application specific. For instance, RDF benchmarks [1] focus only on finding structures that match a set of RDF triples. While another graph benchmark, LDBC [26], simulates queries on a social graph. We have used such benchmarks, among others, to create our list of test queries. Moreover, in our experiments, we illustrate the limitations of complex benchmarks. Our goal is not to replace such benchmarks but to enhance them with the extra insights that our own methodology can bring.

## 3. GRAPH DATABASES

Graph databases adopt the *attributed graph model* [5]. Graph data is data consisting of nodes (also called vertices)

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 |

and connections between them, called edges. Edges have labels, and every node or edge has a set of attributes or properties, i.e., set of name-value pairs. In the implementation of such a model, graphs and edges are typically first class citizens and are assigned internal identifiers.

## 3.1 Systems

For a fair comparison, we need all the systems to support a common access method. For this we considered systems that support the Gremlin query language [9] through officially recognized implementations. Gremlin [56] is the query language with the most widespread support across graph databases and can be seen as the SQL of the Graph Database Systems [33]. We also required that the systems we consider have a licence permitting the publication of experimental comparisons and their operation on a server without a fee. Furthermore, we wanted to consider every system that has been studied or mentioned in similar previous studies [58].

Given these requirements, we consider a number of native graph databases. Among them is Neo4j [51], that stores nodes and edges natively, but separately, and supports some schema indexes. Another such database is OrientDB [53], a multi-model database where nodes are considered documents and edges are considered links. It supports SB-Trees, hash and Lucene full-text indexes for node search.

We consider also hybrid systems, i.e., graph databases using third-party store engines. In particular, we study Sqlg and ArangoDB that have never been considered in the past, as well as Titan. ArangoDB [11] offers a REST API for interaction, uses a document store, and builds automatically indexes on edge endpoints. Sqlg [46] is an implementation of Apache TinkerPop on the relational database Postgresql [55]. It models every vertex type as a separate table and edge labels as many-to-many join tables. The indexes it supports are those provided by the relational engine. Titan [64][2] uses different third-party storage and indexing engines. For storage, it may use Cassandra [7], HBase [8], BerkeleyDB [52], or an in-memory storage engine (not intended for production use). 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. For node search, it supports *graph centric* and a *vertex centric* indexes. We have excluded from our study Caley [19] and DGraph [21] which are hybrid systems but do not support Gremlin.

We have also considered some RDF databases like BlazeGraph and Sparksee. Although tailored for RDF data, such systems look very similar to graph databases. Blazegraph [62] is an RDF store, using B+Tree indexed journal files for storing the graph data. Sparksee[3] [60] is a commercial graph storage system that provides fast queries of graph structures by partitioning them in clusters of bitmaps. Not all RDF systems, however, support Gremlin, which is why systems of that family, like Apache Jena [29], Virtuoso [59], and AllegroGraph [34], have not been considered.

Table 1 provides a summary of the main characteristics of the aforementioned systems, on which we expand below. Note that for some systems we considered two versions. We do so because we want to illustrate the degree of progress that has been achieved in these systems from one version to another, and also help stakeholders decide whether it is worth the effort to upgrade. The most recent versions have adopted a newer version of Gremlin that has cleaner semantics, less overloaded operators, and richer operator set. Gremlin is mainly a syntax, thus, any performance variation observed across the different versions of the same system will most likely be due to more effective implementation and not due to the actual language per se.

## 3.2 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 is to delegate some functionalities to other existing systems (*hybrid* databases). In both cases, the two challenges to solve are how to store the graph and how to traverse it.

**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 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.

[2]At the time of writing, Janus Graph has just started as the successor of Titan, yet here we test the last version of Titan.

[3]Formerly known as DEX [48]. We tested it through a free research licence.

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** employs separate data structures: one structure for objects, 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 [47]. In each data-structure objects are identified by sequential IDs, and each ID is linked as a key through 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. For instance, given a label, one can scan the corresponding bitmap to identify which edges share the same label. Furthermore, bitmaps identify 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.

**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 materializes 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 accelerates 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 access 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. Hence, this approach requires unions and joins 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.

**Query Processing and Evaluation:**
All the systems we considered support the Apache TinkerPop [9] framework that acts as a database-independent connectivity layer supporting Gremlin. A Gremlin query is a series of operations. Consider, for instance, Q28 in Table 2, which selects nodes with at least $k$ incoming edges. For every node (`g.V`), it applies the filter (`.filter{...}`) by counting the incoming edges (`it.inE.count()`). 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 the outer and the inner part respectively. ArangoDB does not provide any overall optimization of these parts. Note that Gremlin is a Turing-complete language and can describe complex operations that declarative languages, like AQL or Cypher, may not be able to express in one query. **Sqlg** translates all operations to a declarative query language. Moreover, Sqlg, where possible, tries to conflate operators in a single query, which is some form of query optimization. In **OrientDB** some consequent operators may get translated into queries and then their result 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. **BlazeGraph**, **Neo4J**, and **Sparksee**, instead, 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.

## 4. QUERIES

The set of queries selected for our tests adhere to a micro-benchmark approach [15] that has been repeatedly used in many cases [24, 28, 65]. The list is the result of an extensive study of the literature and of many practical scenarios. Of the many complex scenarios we found, we identified the very basic operations of which they were composed. We obtained in this way a set of common operations that are independent of 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 their properties. Specifically for the creation, we treat the initial loading of the dataset and the individual object creations as separate cases. The reason is that the first happens in bulk mode on an empty instance, while the second at runtime with data already in the database. The category of Read operations comprises statistical operations, content search, and filtering content.

We consider also **T**raversal operations across nodes and edges, which are characteristic in graph databases. We recall that operations like finding the centrality, or computing strongly connected components are typical in graph analytic systems and not in graph databases. The categorization we follow is aligned to the one found in other similar works [3, 37, 38] and benchmarks [26]. The complete list of queries can be found in Table 2 and is briefly presented next. In addition to those queries, we also run a set of complex queries in order to compare the insights they provide with the results of the other operators, as well as testing the query optimization capabilities of the systems. These queries are taken from a social network application benchmark [4].

### 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 a GraphSON [10] file. Additional operations may be needed for loading, e.g., to deactivate indexing, but in general, they are vendor specific, i.e., not found in the Gremlin specifications.

**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. | *Shortest Path on 'l'* | *Same as Q.34*, but only following label $l$ | |

* [] denotes a Hash Map; $g$ is the graph; $v$ and $e$ are node/edges.

## 4.2 Create Operations

Creation operators may be for nodes, edges, or even properties on existing nodes or edges. To create a complex object, e.g., a node with a number of connections to existing nodes, these operators must often be called multiple times. We tested the insertion of nodes alongside some initial property (Query 2), the insertion of edges with and without properties attached (Query 3, and 4), the insertion of properties on top of existing nodes or edges (Query 5, and 6), and finally the insertion of a new node, alongside a number of edges that connect it to other nodes already in the database (Query 7). Note that one does not create an edge label without an edge, so edge labels are instantiated with the edge instantiation. In some of these (and other queries below) the node (or the edge) is explicitly referred through its unique id, and thus no search task is involved, as the lookup for the object is performed before the time is measured.

## 4.3 Read Operations

**Graph Statistics.** (Queries 8, 9, and 10) Among the operations that require a scan over the entire graph dataset, three are included in the evaluation set. The first one scans and counts all the nodes, the second counts all edges, and the last counts distinct edge-labels. Performing the last operation also tests the ability of the system to maintain intermediate information in memory, since it requires to eliminate duplicates before reporting the results.

**Search by Property.** (Queries 11, and 12) These are the basic operators used for content filtering since they search for nodes (or edges) that have a specific property. The name and the value of the property are provided as arguments.

**Search by Label.** (Query 13) This task is similar to the previous, but filters edges with a given label. Labels are fundamental components in a graph, and probably for this reason the syntax in Gremlin *3* has distinct operators for labels and properties, while in *2.6*, they are treated equally.

**Search by Id.** (Queries 14, and 15) As it happens in almost any other kind of database, a fundamental search operation is the search by a key, i.e., ID. These two queries retrieve a node and an edge, respectively, via their unique identifier.

## 4.4 Update Operations.

Since edges are first class citizens of the system, an update of the graph structure, e.g., on the connectivity of two or more nodes, requires either the creation of new edges or the deletion of an existing one. In contrast, updates on objects properties are possible without deletion/insertion. Thus, we include Queries 16, and 17 to test the ability of a system to change the value of a property of a specific node or edge.

## 4.5 Delete Operations.

We include three types of deletions: the deletion of a node (Query 18), which implicitly requires also the elimination of all its properties and edges; the deletion of an edge and its attached properties (Query 19); and the deletion of a property from a node or an edge (Queries 20, and 21).

## 4.6 Traversals

**Direct Neighbors.** (Queries 22, 23, and 24) A popular operation is the retrieval of all the nodes directly reachable from a given node (1-hop), i.e., those that can be found by following either an incoming or an outgoing edge. Finally, a specific query performs a 1-hop traversal only through edges having a specific label, which allows more advanced filtering.

**Node Edge-Labels.** (Queries 25, 26, and 27) Given a node, we often need to know the labels of the incoming, outgoing, or both types of edges. This set of three queries performs these three kinds of retrieval 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 first three queries retrieve nodes with at least $k$ edges. They differ from each other in considering incoming, outgoing, or both types of edges. 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) Some search operations give preference to nodes found in close proximity and are better implemented with a breadth-first search from a given node. This ability is tested with these two queries. 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 graphs is the discovery of the path between two nodes that contains the smallest number of edges. Thus, we include these two queries, with the second query being the special case that considers only edges with a specific label.

## 4.7 Complex Query Set.

In order to compare the insights obtained using the micro-benchmark approach with those using a macro-benchmark, and to test the ability of the systems to optimize complex queries, we also created a workload of 13 queries based on the LDBC Social Network benchmark [26]. These queries mimic the tasks that may be performed by a new user in the system, from the creation of an account (creating a new node with attributes) and fill-up of the profile (connecting to nodes representing the school, place of birth, and workplace), to the task of retrieving recommendations of items or other users. For these operations, we include in the workload queries composed of multiple primitive operators, multiple join predicates, sorting, top-k, and max finding. The complete list is detailed in the technical report [40].

## 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. We ensured that each query execution was performed in isolation so that it was not 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 (more details are reported in the report [40]). The goal is to perform a comparative evaluation and not an evaluation in absolute terms. Both real and synthetic datasets have been used, especially of large volumes, in order for the experiments to be able to highlight the differences across the systems also in terms of scalability. Our evaluation methodology has been materialized in an open-source test suite [40], which contains scripts, data, and queries, and is extensible to new systems and queries.

**Common Query Language.** All queries are written in Gremlin [56] that all the graph databases we tested support through implemented adapters.

**Hardware.** For the experiments we used a machine with a 24-core CPU, an Intel Xeon E5-2420 1.90GHz, 128 GB of RAM, 2TB hard disk (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 vendors.

**Test Suite.** We have materialized the evaluation procedure into a software package (a test suite) and have made it freely available, enabling repeatability and extensibility. The suite contains the scripts for installing and configuring each database in the Docker environment and for loading the datasets. It provides also the queries and a script that instantiates the Docker containers and provides the parameters required by each query. To test a new query it suffices to write it into a dedicated script, while to perform the tests on a new dataset, one only needs to place the dataset in GraphSON file (plain JSON) in the dedicated directory.

**Datasets.** We tested our system on both real and synthetic datasets. The first dataset (*MiCo*) describes co-authorship information crawled from the CS Microsoft Academic portal [25]. Nodes represent authors, while edges represent co-authorships between authors and have as a label the number of co-authored papers. The second dataset (*Yeast*) is a protein interaction network [14]. Nodes represent budding yeast proteins (*S.cerevisiae*) [16] 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 respective protein classes.

**Table 3: Dataset Characteristics**

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

The third real dataset is Freebase [30], which is one of the largest knowledge bases freely available for download nowadays. Nodes represent entities or events, and edges model relationships between them. We took the latest snapshot, cleaned it, and considered four subgraphs of it of different sizes [39, 50]. The raw data dump contains 1.9B triples [30], many of which are duplicates, technical or experimental meta-data, and links to other sources that are commonly removed [13, 50], thus leaving a clean dataset of 300M facts. The sizes of the samples were chosen to ensure that all the engines had a fair chance to process them in a reasonable time, but, on the other hand, to show also the system scalability at levels higher than those of previous works.

In this study, we created one subgraph (*Frb-O*) by considering only the nodes related to the topics of organization, business, government, finance, geography and military, alongside their respective edges. Furthermore, we created other 3 graph datasets by randomly selecting 0.1%, 1%, and 10% of the edges from the complete graph, resulting in the *Frb-S*, *Frb-M*, and *Frb-L* datasets, respectively.

We generated a synthetic dataset [40] using the data generator provided by the Linked Data Benchmark Council (LDBC) [26], which produces graphs that mimic the characteristics of a real social network with power-law structure, and real-world characteristics like assortativity based on interests or preferences (*ldbc*). We selected this in place of any available social network dataset because it is richer in attribute types, edge types and relationships, and it is the only existing dataset with attributes on the edges.

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 ($\Delta$).

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

**Evaluation Metrics.** We evaluated the systems considering the disk space, the data loading, the query execution time, as well as our experience in their usage.

## 6. EXPERIMENTAL RESULTS

We provide first an overview of the results of the experimental evaluation performed for the individual types of operations, and then, in Section 6.5 and Table 4, we provide an overall evaluation and the main insights.

Throughout our tests, we noticed that *MiCo* and *ldbc* were giving results similar to *Frb-M* and *Frb-O*. *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 (with 314M edges and 76M nodes), but only Neo4J, Sparksee, and Sqlg managed to do so without errors, and only Neo4J (v.3.0) successfully completed all the queries. Furthermore, the running times recorded on the full dataset respected the general trends witnessed with its subsamples. Thus, in what follows, we focus only on the results of *Frb-S*, *Frb-O*, *Frb-M*, and *Frb-L* and make reference to the other samples only when they show a behavior different from the one of Freebase. Additional details about the experimental results that are not mentioned here can be found in the extended version of the paper [40].

### 6.1 System Configuration

The system configuration is important since the different parameters affect significantly its performance. Neo4J does not need any special configuration to run. 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 in order to support more. ArangoDB requires configurations for the engine and for its V8 javascript server for logging. With default values, it generates approximately 40 GB of log files in 24 hours and is not possible to force it to allocate more than 4GB of memory. In Titan, the most important configuration is that of the JVM Garbage Collection and of the Cassandra backend. The other systems that are also based on Java, namely, BlazeGraph, Neo4J, OrientDB and Titan, are equally sensitive to the garbage collection, especially for very large datasets that require large amounts of main memory. As a general rule, the option `-XX:+UseG1GC` for the *Garbage First* (G1) garbage collector is strongly recommended. Finally, Sqlg has a limit on the maximum length of labels (due to Postgresql), which requires special handling.

As a general observation, it seems that Neo4J is a mature system in which the developers have paid attention to both usability and automatic tuning. The rest of the systems are heavily restricted by their underlying technology, which significantly affects the system performance if not well-tuned.

### 6.2 Data Loading

**Execution.** Many systems were failing or taking days to load the data through Gremlin (using query 1). The Gremlin implementation of ArangoDB was sending each node and edge insertion instruction separately to the server via HTTP calls, making it prohibitively slow to load with this method even small datasets. OrientDB had a similar limitation and was, in addition, performing a lot of bookkeeping tasks for each edge-label it was loading. For both we used implementation-specific scripts and commands, bypassing the Gremlin library, in order to load the datasets. To load the data in BlazeGraph, we had to explicitly activate a *"bulk loading"* option. Without it, the system processes each label and node separately and updates its meta-data structures after the insertion of each such item. In Titan, the delays were higher. That was due to consistency checks and schema inference tasks. Disabling automatic schema inference was significantly reducing the loading times, but required to specify the schema before inserting any data, meaning that Titan was not able to handle dynamic schema updates transparently. Neo4J, Sqlg, and Sparksee managed loading the data through the Gremlin API without issues, which indicates that they offer a good Gremlin implementation.

**Time.** In terms of loading time (Figure 3(a)), ArangoDB was the fastest, mainly thanks to the use of native scripts we had to employ to load data in reasonable time. Neo4J was almost equally fast, proving that a good implemented Gremlin API can achieve as good performance as the native scripts. The loading time of the different size datasets on Sqlg and OrientDB illustrated a high sensitivity to the edge label cardinality. This was because both Sqlg and OrientDB create and use different structures for different edge labels. BlazeGraph, on the other hand, updates and balances its B+Tree index structure after every insertion, and this made it up to 3 orders of magnitude slower than the other engines.

**Space.** We studied the disk space that the datasets occupied in the different systems to identify those with the most effective compression strategies. Although disk space may not be a major concern nowadays, it becomes relevant, for instance, in systems with solid state drives. The results of the experiments are shown in Figures 1(a) and 1(b). The results of the datasets *Frb-O*, *Frb-M*, and *Frb-L* show Titan as the one with the best space performance. Its strategy is to compact node identifiers in each adjacency list with a form of delta encoding, a strategy very effective in graphs with nodes of high degree. For the *ldbc* dataset, instead, where much textual information is shared by many objects, OrientDB and Sparksee achieved the least space consumption because they de-duplicate attribute values. Given that OrientDB creates different files for each distinct edge label, we see that it is the second last in terms of space on the *Frb-S* dataset that contains many different edge labels (∼1.8K) for relatively few edges (∼300K). Finally, we can see that BlazeGraph requires, on average, three times the size of any other system, on all the datasets, and this is due to the fact that it instantiates a journal file of fixed size, and produces a lot of data replication with its different indexes.

### 6.3 Complex Queries

For completeness, we first evaluate the graph databases using complex queries of an existing benchmark, the LDBC, applied on their *ldbc* dataset (Figure 2). For these as well we set a time-out of 2-hours. BlazeGraph is not reported in the figure because the queries timed-out. ArangoDB and Titan (v.0.5) were, in general, the slowest, which indicates that they could not effectively exploit the index structures and neither employ any advanced optimization. Yet, for ArangoDB this result fails to demonstrate that there are cases (identified below) in which it can actually perform better than others. Titan (v.1.0) was very fast for some queries involving short joins and for some with single-label selections. Yet, *the micro-benchmark analysis* (below) shows that this result doesn't generalize. This type of performance is due to the specific query and to the help of caching from the Cassandra back-end. As we will see below in the result of our microbenchmark, Titan (v.1.0) is consistently slower

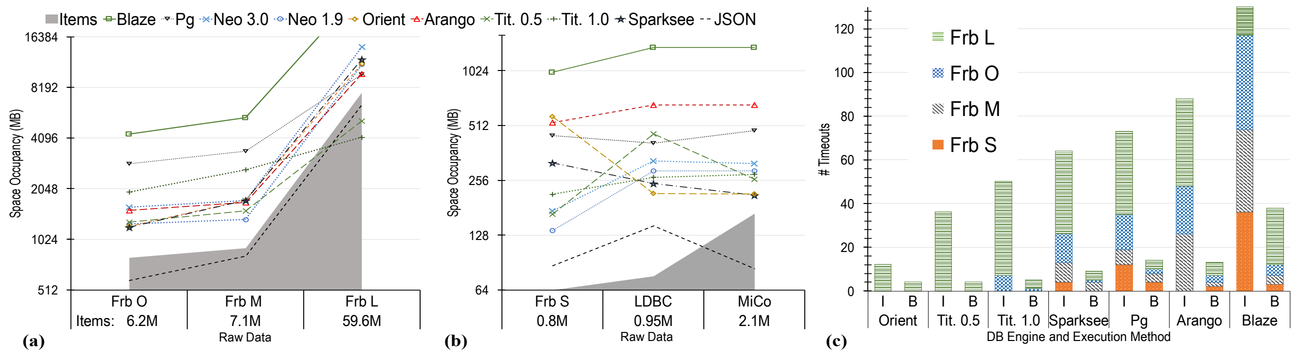

**Figure 1: Space occupancy (left and center) and Time-outs for Interactive (I) and Batch (B) modes (right).**

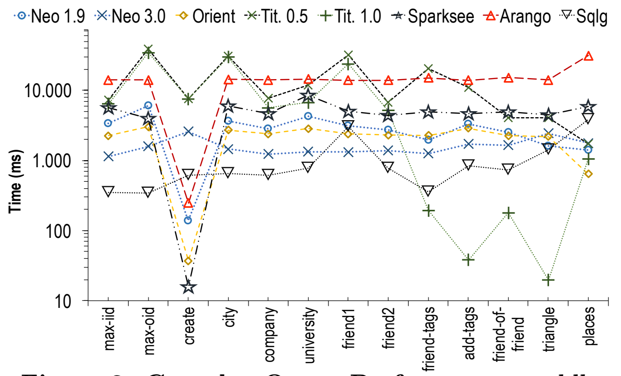

**Figure 2: Complex Query Performance on *ldbc*.**

than other systems when the graph gets larger and when the query cannot exploit any cache. Sqlg is the fastest in almost half the queries. Hence, we question the reason why in some cases (e.g., the last query) Sqlg is much slower than the competition. Especially, it is puzzling to compare the last and the second to last queries, both performing some sort of traversal. Again, *the micro-benchmark analysis* identifies the characteristics of the best performing operators in Sqlg, which are exploited by those queries that can be translated to a single relational operator or to conditional join queries, with no recursion and short join chains that traverse only few edge labels with limited cardinality. In these queries, the system does not incur in expensive joins and it is able to take advantage of the relational optimizer and exploit indexes. Those cases in which Sqlg is slower are, instead, queries that traverse many edges and do not filter on a single edge label, and thus generate large intermediate results.

## 6.4 Micro-benchmark Results

We now turn to our micro-benchmark queries.

**Completion Rate.** For online applications, it is important to ensure that the queries terminate in a reasonable amount of time. For this, we count the queries that did not complete within 2 hours, either in isolation or in batch mode, and illustrate the results in Figure 1(c).

Neo4J, in both versions, is the only system which successfully completed all the tests with all the parameters on all the datasets (omitted in the figure). OrientDB is the second best, with just a 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 for some batch executions on *Yeast*, and almost on all the 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 breath-first traversals with depth 3 or more reach the timeout on *Frb-O*, *Frb-M* and *Frb-L* in most databases. The 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 the databases apart from Neo4J and Titan (v.1.0). In particular for Sparksee, on all the Freebase subsamples, these queries cause the system to exhaust the entire available RAM and swap space (this has been linked to a known problem in the Gremlin implementation). ArangoDB failed these last queries only on *Frb-M* and *Frb-L*, and OrientDB only on *Frb-L*. These results highlight the benefits of separating the graph structure from the attribute values, allowing native systems to execute fast even queries that require access to large portions of the graph.

**Insertions, Updates and Deletions.** For operations that add new objects (nodes, edges, or properties), we experienced extremely fast performances for Sparksee, Neo4J (v.1.9), and ArangoDB, with times below 100ms, with Sparksee being generally the fastest (Figure 3(b)). Moreover, with the only exception of BlazeGraph, all the databases are almost unaffected by the size of the dataset. We attribute this result to the internal configuration of the datastructures adopted where elements are stored in append-only lists, while for ArangoDB these operations are registered in RAM and asynchronously flushed to disk. BlazeGraph, on the other hand, is the slowest with times between 10 seconds and more than a minute as each of these operations require multiple index updates. Both versions of Titan are the second slowest systems, with times around 7 seconds for the insertion of nodes, and 3 seconds for the 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 insertions 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, showing inconsistent behavior, for edge insertions. 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. We will see below that this depends on some initialization procedure. Sqlg is one of the fastest for insertions of nodes as these operations translate into inserting a tuple into a relational table, while is much

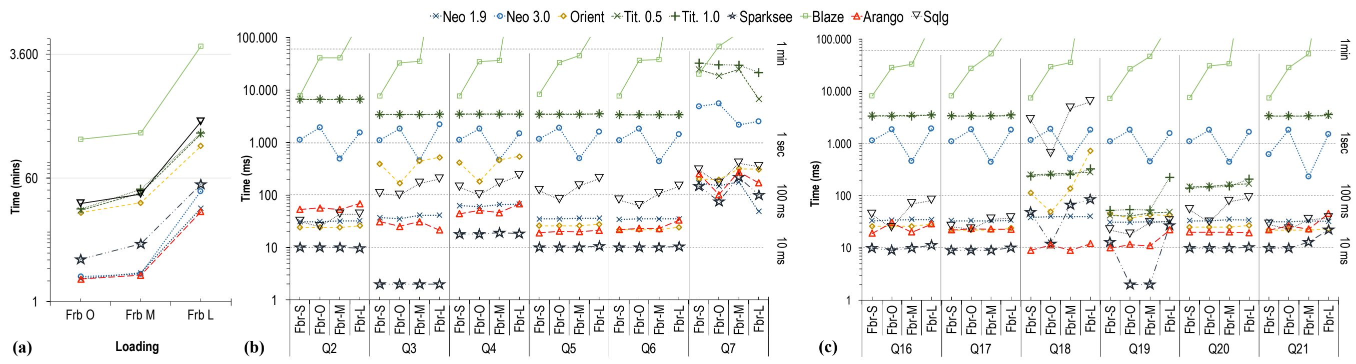

**Figure 3: Time required for (a) loading, (b) insertions, and (c) updates and deletions.**

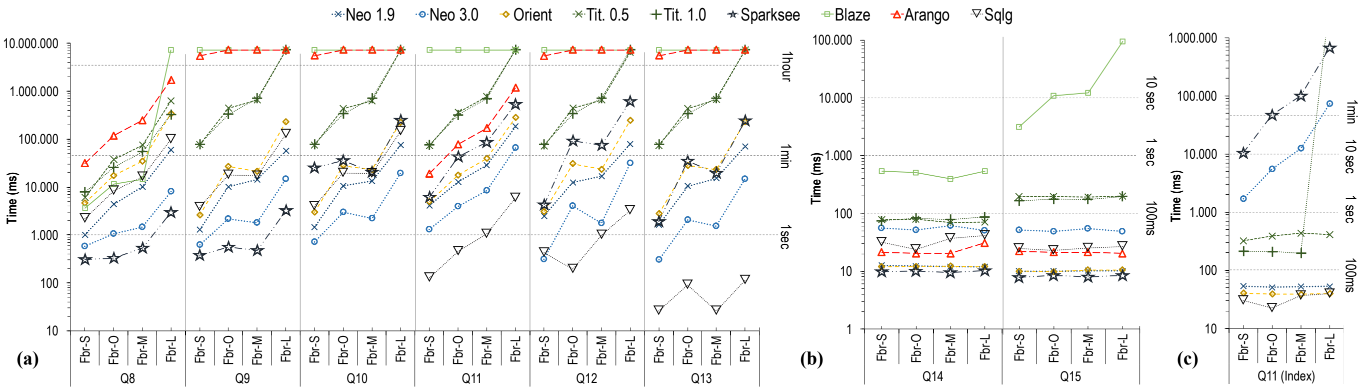

**Figure 4: Selection Queries: general selections (a), search by id (b), and indexed property search (c).**

slower for all other queries where it has to change the table structure. 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).

On the other hand, the performance of node removal (Q.18) for OrientDB, Sqlg, and Sparksee seems highly affected by the structure and size of the graphs (Figure 3(c)). While ArangoDB and Neo4J (v.1.9) remain almost constantly below the 100ms threshold, Neo4J (v.3.0) completes all the deletions between 0.5 and 2 seconds, showing that there is some overhead intervening. Finally, for the removal of nodes, edges, and node properties, Titan obtains an improvement of almost one order of magnitude, by exploiting the benefits of the data organization in the column stores. Note that ArangoDB is also consistently among the fastest, but, its interaction through REST calls, the fact that updates are asynchronous, and the missing support for transactions, constitute a bias on those results in its favor because the time is measured on the client side and we have no control on when those operations get materialized on disk.

**General Selections.** With *read* queries, some heterogeneous behaviors show up. The search by ID (Figure 4(b)) differs significantly from all the other selection queries (Figure 4(a)), and it's in general much faster, this indicates special attention from all vendors on this operation. BlazeGraph is again the slowest. All other systems take between 10ms (Sparksee) to 400ms (Titan) to satisfy both queries.

In counting nodes and edges (Q.8, and Q.9), Sparksee has the best performance followed by Neo4J (v.3.0). For BlazeGraph and ArangoDB, node counting is one of the few queries in this category that complete before timeout. Edge iteration, on the other hand, seems hard for ArangoDB, which rarely completes within 2 hours for the Freebase datasets, as it materializes all edges while counting them.

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 is a little slower. Since the previous experiments showed that Sparksee is fast in iterating over the edges, we identified here a sub-optimal implementation of the de-duplication step.

The search for nodes (Q.11) and edges (Q.12) based on property values perform similar to the search for edges based on labels (Q.13), for almost all the 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 others. Hence, equality search on edge labels has not received special optimizations in the various native systems. This is despite Sparksee and OrientDB have data-structures that should help optimizing this operation.

In general, the above results support the choice of separating structure and data records since it allows to iterate over the entire set of objects without materializing them. They also indicate the importance of identifying the properties to be indexed, since, in all the systems, the search task became problematic for large datasets. The RDBMS was less affected in this situation, especially for edge labels due to the storage of the relations in separate tables.

**Traversals.** For traversal queries that access the direct neighborhood of a specific node (Q.22 to Q.27), OrientDB, Neo4J (v.1.9), ArangoDB, and then Neo4J (v.3.0) are the fastest (Figure 5(a)), with response times below the 60ms, and are robust to variations in graph size and structure.

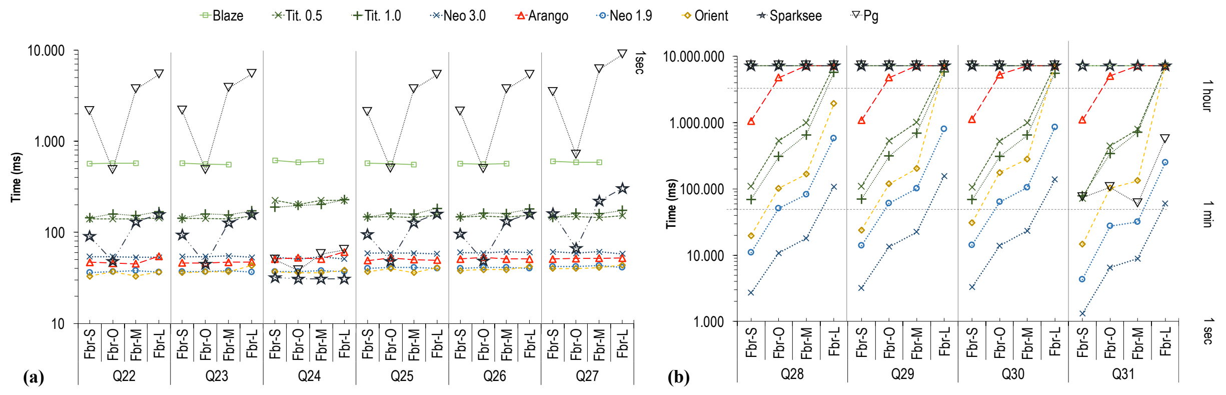
Figure 5: Time required for traversal operations: (a) local access to node edges, and (b) filtering on all nodes

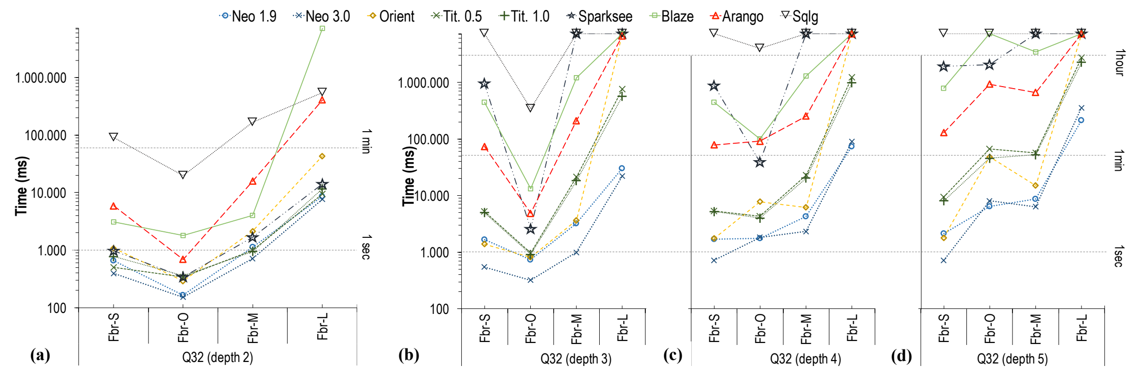
Figure 6: Time required for breadth-first traversal (a) at depth= 2, and (b,c,d) at depth 3, 4, and 5

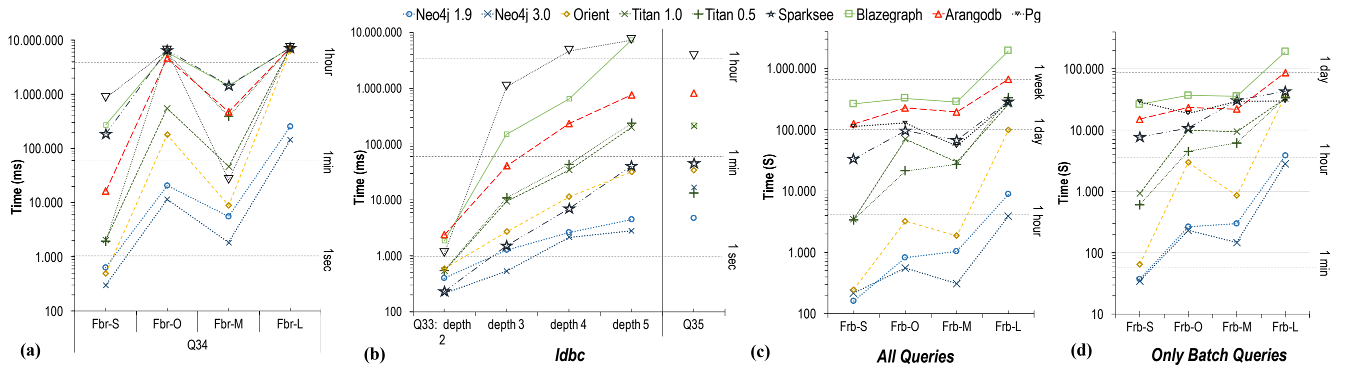
Figure 7: Performance of (a) SP, (b) label-constrained BFS and SP, and (c,d) Overall for Single and Batch.

In contrast, Sparksee seems to be more sensitive, requiring around 150ms on *Frb-L*. The only exception for Sparksee is the visit of the direct neighborhood of a node filtered by edge labels, in which case it is on par with the former systems. BlazeGraph is 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 (explaining the good performance in Figure 2).

When comparing the performance of queries from Q.28 to Q.31, which traverse the entire graph filtering the nodes based on the edges around them, as shown in Figure 5(b), Neo4J (v.3.0) presents the best performance, with its older version being the second fastest. Those two are also the only two engines that complete these queries on all datasets. All the systems tested are 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, identifying a problem in the implementation, as this never happens in any other case. BlazeGraph as well hits the time 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, with time comparable to Neo4J (v.1.9). Yet, all systems complete the tasks on *Yeast*, *ldbc* and *MiCo*.

Breadth-first (BFS) (Q.32 and Q.33) and shortest path (SP) search (Q.34 and Q.35) are important operations in graphs. The performance of the unlabeled version of BFS, shown in Figure 6, highlights the good scalability of Neo4J at all depths. OrientDB and Titan give the second fastest times for depth 2, with times 50% slower than those of Neo4J.

For depth 3 and higher (Figures 6(b,c,d)), 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 sometimes. For the shortest path with no label constraint (Q.34, Figure 7(a)), the performance of the systems was similar to the above, even though BlazeGraph and Sparksee are in this case very similar, while Sqlg is still the slowest since it accesses all tables for all edges, and performs very large joins.

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 stops the exploration almost immediately, i.e., beyond 1-hop the query returned an empty set, hence the running time was not showing any interesting result. Running the same queries on *ldbc* we still observe (Figure 7(b)) that Neo4J is the fastest engine, while Sparksee is the second fastest in par with OrientDB for the breadth-first search. Instead, on the shortest path search filtered on labels, Titan (v.1.0) gets the second place.

Such results support the choice of index-free traversals implemented by the native systems for large and expensive visits on the graph. Yet, the dedicated structural index employed by Titan reaches the second-best performance.

**Effect of Indexing.** The existing systems provide no support for structural indexes, i.e., user-specified indexes for graph structures. They all have some form of indexes already implemented and hard-wired into the system. The only kind of index that can be controlled by the users is on attributes, and this is what we study. BlazeGraph provides no such capability, so is not considered. ArangoDB showed no difference in running times, so we suspect some defect in the Gremlin implementation. Insertions, updates, and deletions, as expected, become slower since the index structures have to be maintained, but not more than 10% in general, apart from Neo4J (v.3.0) and OrientDB that showed delays of about 30% and 100%, respectively. Despite this increase, Neo4J (v.1.9), Sparksee, and OrientDB remain the fastest systems for *CUD* operations. For search queries on node attributes, i.e., Q.11 (Figure 4(c)), the presence of indexes gives 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 witnesses up to a 600x speed up. We also see that Titan (v.1.0) still encounters problems on *Frb-L*. Sparksee and Neo4J (v.3.0) are not able to take advantage of such indexes. As a conclusion, it seems that there is space for optimization in this sector.

**Single vs Batch Execution.** We looked at the time differences between single (run in isolation) and batch executions (Figure 7(c) and (d)). Tests in batch mode do not create any major changes in the way the systems compare to each other. For the retrieval queries, the batch executions of 10 queries take exactly 10 times the time of one iteration, i.e., no benefit is obtained from the batch execution. Instead, for the "CUD" operations, the batch takes less than 10 times the time needed for one iteration, meaning that in single mode most of the running time is due to some initial set-up for the operation. For traversal queries, the batch executions only stress the differences between faster and slower databases.

**Progress across Versions.** An important observation regards the difference in performance observed across different versions of the same system, e.g., Neo4J. For Neo4J, in some

**Table 4: Evaluation Summary**

| | L | C | R | | | U | D | | T | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Load | Insertions | Graph Statistics | Search by Property\Label | Search by Id | Updates | Delete Node | Other Deletions | Neighbors | Node Edge-Labels | Degree Filter | BFS | Shortest Path |
| ArangoDB | | ✓ | ⚠ | ⚠ | | ✓ | ✓ | ✓ | | | ⚠ | ⚠ | ⚠ |
| BlazeGraph | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ |
| Neo4J (v.1.9) | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Neo4J (v.3.0) | ✓ | | ✓ | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| OrientDB | | ✓ | | | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | |
| Sparksee | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | | | | | ⚠ |
| Titan (v.0.5) | ⚠ | | ⚠ | ⚠ | | | | | | | | | |
| Titan (v.1.0) | ⚠ | | ⚠ | ⚠ | | | | | | | | | ⚠ |
| Sqlg | | | | ✓ | | ✓ | | ✓ | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ |

cases, we see improvements or similar performance of the newer version compared to the old. Yet, we observe the opposite in very fast operations, i.e., "CUD" queries and Search by ID. This is due to the overhead for accessing a wrapper library that was added in the newer version to cope with some licencing issues (i.e., Tinkerpop adopted the Apache licence, and the incompatibility in the licences forced the developer to add a wrapper). Furthermore, the global node filtering based on degree (Q28-31) and also some other traversals not filtered by type, show worsened performance. We found that in the new version the storage format and the part for traversal of relationships have been completely rewritten to improve filtered traversals (so traversals restricted to a single edge type). In particular, relationship chains are now split by type and direction. Disk storage format changed as well. This probably adds an overhead to queries that access many edges of different types. Titan, on the other hand, demonstrated a slightly improved performance in the newer version. There, the main difference is that the software became production ready (from v.0.5 to v.1). It is important to note here that the newer version supported also a newer version of Gremlin language. The new version is offering a cleaner syntax, but the way the operators have been implemented across versions is orthogonal to the language.

## 6.5 Overall Evaluation and Insights

Table 4 provides a summarization of the observed performance of the different GDBs in our experiments. The tick symbol (✓) means that the system achieved the best or near-to-best performance. The warning symbol (⚠) means that the system performance was towards the low end or indicated execution problems. Through this table, it is possible for a practitioner to identify the best system for a specific workload or scenario. One can see for instance that the native graph databases Neo4J, OrientDB and in part Sparksee, are better candidates for graph traversals operators (**T**). On the other hand, with data of few node and edge types, and a heavy search workload, hybrid systems may be a better fit.

**Neo4J** is the system with the shortest execution time when looking at the cumulative time taken by each system to complete the entire set of queries in both single and batch executions (Figure 7(c) and (d)).

**OrientDB** also obtained relatively fast running times, which are often on par with Neo4J, and in some cases better than one of its two versions.

However, it doesn't perform well when large portions of the graph have to be processed and kept in memory, e.g., computing graph statistics on *Frb-L*.

**Titan** results are quite often one order of magnitude slower than the best engine. It is slower in create and update operations but faster in deletions. This is most likely due to the tombstone mechanism, that in deletions marks an item as removed instead of actually removing it.

**Sparksee** gives almost consistently the best execution time in create, update and delete operations. Although it is not very fast with deletions of nodes having lots of edges, it is still better than many of the others. It performs better also in edge and node counts, as well as in the retrieval of nodes and edges by ID, thanks to its internal compressed data structures. Nevertheless, it performs worse than the others for the remaining queries due to suboptimal filtering and memory management. For instance, it gives a lot of timeouts on the degree-based node search queries.

**ArangoDB** excels only in few queries. For creations, updates and deletions, it ranks among the best. This is also because all updates are kept in main memory and synced on disk asynchronously. For retrievals and search, its performance is in general poor. This is due to the way Gremlin primitives are translated into the engine, where ArangoDB has to materialize all the objects in order to iterate through them. An exception is when searching by ID, which is expected since at the core it is a key-value store, while for traversals it has a narrow lead over Sparksee and BlazeGraph demonstrating some effectiveness of its edge-specific hash index.

**Sqlg** shows the expected low performance for all the traversal operations, due to the need to traverse the graph via relational joins instead of via direct links to node/edges. However, for queries containing 1 or 2-hop traversals restricted to a single edge-label, like some of the complex queries, it performs extremely well. In these cases, it takes advantage of the ability to conflate multiple operations in a single query and filter using foreign key indexes for specific edge-labels.

**BlazeGraph** results show in general that the indexes it builds automatically do not help much. Moreover, since each single step is executed against some specific graph API, instead of having the Gremlin query translated into SPARQL and executed as such, its query processing is, in general, the less efficient. This graph API implementation does not allow it to exploit any of the optimization implemented by the SPARQL query engine.

**System Selection.** All the above observations can serve as a guide in the choice of the right system for the different scenarios. The two main factors that should be considered in each scenario are the characteristics of the dataset and the intended workload, with the latter weighing more. When most of the intended operations are search on node properties, with few traversals, a hybrid system is preferable, e.g., Sqlg. Such systems allow also the re-utilization of the existing technologies in an enterprise and allow the exploitation of robust optimizers and advanced index mechanisms. The choice of hybrid systems is also preferable for data in large enterprises that have a low degree of heterogeneity. On the other hand, when the data is highly heterogeneous, i.e., many different edge types, and in the workload is predominant the presence of long traversals, native graph systems appear to be a better choice. This kind of data is often coming from social networks or is the result of the integration of many different heterogeneous sources. Another factor to consider is the dynamicity of the data. If many insert, update, and delete operators are to be performed, Sparksee, and ArangoDB are the best performing in our study. If however the data is going to be relatively static and the majority of operations are going to be search queries, then Neo4J and OrientDB perform better on graph search and Sqlg on content filters. It is important to note that having studied all the well-known systems characterised as GDB, our findings offer a good understanding of the behavior of GDB solutions. There may be of course proprietary or special-purpose solutions, which are not characterised as GDBs, yet, they offer some graph data storage and querying functionality. Such a system may show a different behavior but are not part of our focus of the current work.

**Hybrid and Native systems.** The experiments show that hybrid and native systems perform differently. For a limited set of use cases the hybrid systems in our study perform equally well as the native, but for traversal queries, like finding the connectivity between two nodes, BFS visits, and the enumeration of edges, these hybrid systems under-perform significantly. Hence, this suggests that the design choices made in native systems, e.g., the separation of the graph structure from other data values, are more effective than the strategies adopted by the hybrid systems in our study.

The benefit of the effectiveness of the native GDBs against the hybrid may, however, vary based on the context. In graph analytic pipelines there are many tasks that need to be performed on the data from different tools. Using a native GDB forces the data to be imported in the GDB for the management and exported for other tasks, diminishing the benefit of the effective management the native GDB offers. A hybrid GDB, however, can process the data even if residing in an external storage, thus, big analytic pipeline systems, like SAP Hana, may opt for a hybrid GDB solution.

**Query language.** Although all the systems we studied support Gremlin, each offers also its own native query language and performs all the optimizations on it. Many translate Gremlin queries in a one-to-one fashion to native primitives, but in that way, many Gremlin optimizations cannot be done. This behavior indicates that for many GDBs, Gremlin is not their first priority. The fact that, in some cases, data loading was not possible through Gremlin but only through native calls, and the problems with large intermediate results is an additional evidence of this priority. This optimization, however, is for complex queries. Our microbenchmark approach is based on primitive queries, hence, our findings are not affected by this limitation.

## 7. CONCLUSION

We have performed an extensive experimental evaluation of the state-of-the-art graph databases in ways not tested before. We provided a principled and systematic evaluation methodology based on microbenchmarks. We have materialized it into an evaluation suite, designed with extensibility in mind, and containing datasets, queries and scripts. Our findings have illustrated the advantages microbenchmarks can offer to practitioners, developers, and researchers and how it can help them understand better design choices, performances, and functionalities of the graph database system.

## References

[1] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A survey and experimental comparison of distributed sparql engines for very large rdf data. *PVLDB*, 10(13):2049–2060, 2017.

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

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

[4] 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.

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

[6] 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.

[7] Apache Cassandra. `http://cassandra.apache.org`.

[8] Apache Hbase. `http://hbase.apache.org`.

[9] Apache Tinkerpop. `http://tinkerpop.apache.org/`.

[10] Apache Tinkerpop. Graphson data format. `http://tinkerpop.apache.org/docs/current/reference/#graphson-io-format`, 2015.

[11] Arangodb. `https://www.arangodb.com/`.

[12] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, pages 161–168, 2016.

[13] 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.

[14] Batagelj, Vladimir and Mrvar, Andrej. Yeast, Pajek dataset. `http://vlado.fmf.uni-lj.si/pub/networks/data/`, 2006.

[15] 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.

[16] 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.

[17] E. Bullmore and O. Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186, 2009.

[18] 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.

[19] CayleyGraph. Cayley. `https://cayley.io/`.

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

[21] I. Dgraph Labs. Dgraph. `https://dgraph.io/`.

[22] 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.

[23] 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.

[24] A. Dziedzic, J. Wang, S. Das, B. Ding, V. R. Narasayya, and M. Syamala. Columnstore and b+ tree-are hybrid physical designs important? In *SIGMOD*, pages 177–190, 2018.

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

[26] 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.

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

[28] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. *PVLDB*, 2(1):467–478, 2009.

[29] A. S. Foundation. Apache jena. `https://jena.apache.org/`.

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

[31] 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.

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

[33] 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.

[34] F. Inc. Allegrograph. `https://franz.com/agraph/allegrograph/`.

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

[36] E. Ioanou and Y. Velegrakis. Embench++: Data for a thorough benchmarking of matching-related methods. *Semantic Web Journal*, 9, 2018.

[37] 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.

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

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

[40] M. Lissandrini, M. Brugnara, and Y. Velegrakis. An Evaluation Methodology and Experimental Comparison of Graph Databases. Technical report, University of Trento, 04 2017. Available at `https://graphbenchmark.com`.

[41] 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.

[42] 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. *PVLDB*, 5(8):716–727, 2012.

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

[44] 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.

[45] J. Malicevic, B. Lepers, and W. Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643. USENIX Association, 2017.

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

[47] 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 &#38; Applications Sysmposium*, IDEAS '12, pages 110–119, New York, NY, USA, 2012. ACM.

[48] 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.

[49] 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.

[50] 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.

[51] Neo4j. `http://neo4j.com`.

[52] Oracle, BerkeleyDB. `http://www.oracle.com/technetwork/products/berkeleydb`.

[53] Orientdb. `http://orientdb.com/orientdb/`.

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

[55] E. Prud'Hommeaux, 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] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[59] O. Software. Openlink virtuoso. `https://virtuoso.openlinksw.com/`.

[60] Sparsity Technologies, Sparksee. `http://www.sparsity-technologies.com/`.

[61] 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.

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

[63] R. Tarjan. Depth first search and linear graph algorithms. *Siam Journal On Computing*, 1(2), 1972.

[64] Thinkaurelius, Titan. `http://titan.thinkaurelius.com/`.

[65] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck. Contention-aware lock scheduling for transactional databases. *PVLDB*, 11(5):648–662, 2018.

[66] K. Ueno and T. Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 149–160, New York, NY, USA, 2012. ACM.

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

[68] Q. Zhang, H. Chen, D. Yan, J. Cheng, B. T. Loo, and P. Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 40–51, New York, NY, USA, 2017. ACM.