Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Vojtěch Kolomičenko

Analysis and Experimental Comparison of Graph Databases

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Irena Holubová, Ph.D.
Study programme:	Informatics
Specialization:	Software Systems

Prague 2013

I wish to express my gratitude and appreciation to my supervisor, RNDr. Irena Holubová, Ph.D. for her thoughtful guidance, valuable suggestions during many discussions and her kind supervision of this diploma thesis.

Název práce: Analýza a experimentální porovnání grafových databází

Autor: Vojtěch Kolomičenko

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Holubová, Ph.D.

Abstrakt: V posledních letech se nová skupina NoSQL databází, zvaná Grafové databáze, stala velmi populární kvůli stále se zvyšující potřebě zpracovávat a ukládat grafová data. Cílem této práce je zkoumat různé možnosti a omezení grafových databází a provést analýzu a experimentální porovnání vybraných zástůpců této skupiny. Za tímto účelem byly formulovány obecné požadavky na testování grafových databází a byl vyvinut rozšiřitelný testovací nástroj, nazvaný BlueBench.

Klíčová slova: grafové databáze, NoSQL, benchmark, experimentální porovnání

Title: Analysis and Experimental Comparison of Graph Databases

Author: Vojtěch Kolomičenko

Department: Department of Software Engineering

Supervisor: RNDr. Irena Holubová, Ph.D.

Abstract: In the recent years a new type of NoSQL databases, called Graph databases (GDBs), has gained significant popularity due to the increasing need of processing and storing data in the form of a graph. The objective of this thesis is a research on possibilities and limitations of GDBs and conducting an experimental comparison of selected GDB implementations. For this purpose the requirements of a universal GDB benchmark have been formulated and an extensible benchmarking tool, named BlueBench, has been developed.

Keywords: graph databases, NoSQL, benchmark, experimental comparison

Contents

Introduction **3**

1 NoSQL Movement **4**
1.1 Introduction to NoSQL 4
1.2 Consistency in NoSQL 4
1.2.1 CAP Theorem 5
1.2.2 BASE 5
1.3 Key-value Databases 6
1.4 Column-family Stores 6
1.5 Document-oriented Databases 7
1.6 Graph Databases 7
1.6.1 Graph Model 8
1.6.2 Data Structures 8
1.6.3 RDF 9
1.6.4 TinkerPop Stack 10
1.6.5 Graph Query Languages 11
1.7 Summary 12

2 Graph Database Systems **13**
2.1 DEX 13
2.2 InfiniteGraph 14
2.3 Neo4j 15
2.4 OrientDB 16
2.5 Titan 18
2.6 Summary 19

3 Benchmarking **20**
3.1 Recommended Design 20
3.1.1 Graph Operations 20
3.1.2 Query Classification 21
3.1.3 Benchmarking Process 21
3.2 Benchmark Datasets 22
3.2.1 Real Graph Properties 22
3.2.2 Graph Data Generators 22
3.3 Summary 23

4 Related Work **25**
4.1 GraphDB-Bench 25
4.2 Survey of Graph Database Performance on the HPC-SGAB 26
4.3 Benchmarking Traversal Operations over Graph Databases 28
4.4 A Comparison of a Graph Database and a Relational Database . . 29
4.5 Summary 30

5 Final Design of BlueBench 32

5.1 General Goals . . . 32

5.2 Used Technologies . . . 33

5.3 Conducted Tests . . . 34

5.3.1 Individual Operations . . . 34

5.3.2 Benchmarks . . . 36

5.4 BlueBench Software Architecture . . . 37

5.4.1 Operation . . . 38

5.4.2 Operation Factory . . . 39

5.4.3 Core Benchmark Classes . . . 40

5.4.4 Logging of Operations . . . 40

5.4.5 Loading the Graph . . . 41

5.4.6 Addressing Vertices across Scopes . . . 42

5.5 Summary . . . 42

6 BlueBench Results 43

6.1 Testing Environment . . . 43

6.2 Input Data . . . 43

6.3 Assessed DB Systems . . . 44

6.4 Results of the Tests . . . 49

6.4.1 Graph Loading . . . 50

6.4.2 Traversal . . . 51

6.4.3 Shortest Path . . . 53

6.4.4 Non-traversing Queries . . . 55

6.4.5 Manipulation Queries . . . 56

6.4.6 Summary of the Results . . . 57

6.5 Summary . . . 58

Conclusion 59

Bibliography 61

A Contents of the CD 64

Introduction

In the recent years, there has been a huge increase of importance to store and analyze data in the form of a graph. Be it social networks, Web graphs, recommendation systems or biological networks, these graphs have at least two things in common – they are highly interconnected and huge, and so is the complexity of algorithms used for their processing and analysis of their characteristics. Unlike other types of data, networks contain lots of information in the way how particular objects are connected; in other words, the *relations* between the objects.

Graph database management systems (GDBs) have emerged to fill the gap in the market by providing solutions to the problem of storing and working with large graphs. GDBs elevate the relations to be the most important entities in the database model and are optimized for exploiting the relations to gather information from the graph. In addition, relations are not only mere connections; they contain attributes as well as the other objects in the graph do. GDBs are required to handle these attributes as efficiently as the graph topology itself, since the attributes are often used as guideposts for crawling the graph.

A variety of GDB implementations has been appearing, and most of the systems are usually accompanied with biased proprietary tests claiming that their performance is somewhat superior. Nevertheless, a large scale unbiased benchmark comprising the most substantial graph database functionality and putting a higher number of GDBs into a comparison has not been conducted yet. We believe that this is most likely to be the reason of the versatility of interfaces of particular GDB engines, their configuration specifics and different capabilities.

As a consequence, the objective of this thesis is to summarize the background behind the graph databases family and describe a chosen set of GDB representatives. Furthermore, a discussion on the aspects and implementation details of a fair and complex graph database benchmark is laid down. Finally, a benchmarking suite based on these recommendations, called BlueBench, is created and the performance of a selected set of GDBs experimentally evaluated. The tested databases are carefully chosen so that BlueBench compares not only the most popular GDB engines, but less conventional representatives as well. For this purpose we have implemented a simple wrapper around a selected relational database, which provides the basic functionality expected from GDBs. Therefore, we will be able to confirm or deny the assumptions about relational databases not being competent of handling graph data efficiently.

Contents Overview

In the first chapter we define the term NoSQL and categorize the best known representatives. Most famous implementations of graph database systems are described in detail in Chapter 2. Then, in Chapter 3 we illustrate what principles a good graph database benchmark should obey. Analysis of any existing benchmarking projects follows in Chapter 4. In Chapter 5 our proposed benchmarks are designed. Finally, the benchmarking process and the final results are discussed in Chapter 6.

1. NoSQL Movement

In recent years a new family of databases known as NoSQL[1] has gained lots of popularity, particularly because of the need of storing and effectively retrieving huge volumes of data. Graph databases belong into this family despite their slightly higher complexity; therefore, we provide a brief background to NoSQL and its core technologies.

This chapter is organized as follows. First, NoSQL and its key features are described in short. Second, the most popular types of NoSQL databases are listed. Last, we provide a little more detailed description of graph databases and define the most fundamental terms that will be used in this thesis.

1.1 Introduction to NoSQL

NoSQL is a common label for databases which reject the tradition of using relational models as it is done in Relational Database Management Systems (RDBMS). By contrast, the NoSQL family of databases focuses on providing more scalable, distributed and efficient solutions for handling huge amounts of data. This is made possible by a more relaxed consistency model and less schema-oriented database designs in comparison to RDBMS.

In particular, NoSQL might be a good choice for an application when the data that is needed to be stored does not conform to any easily definable schema or when the tables in an RDBMS database would be too sparse (i.e. having many columns, each of which is only used by the minority of rows). Furthermore, the relational model expects the data stored in separate tables to be connected by logical relations which are used for joining the tables when a more complex query arrives. This works well when the tables' size is not extensive, otherwise the performance decreases with increasing size of the data and number of required joins. NoSQL databases are usually optimized for handling very large data where particular elements are not closely related; therefore there is no need for expensive joins. How exactly the storage layer and other parts of NoSQL databases are designed is different for each NoSQL database type.

1.2 Consistency in NoSQL

In order to really make efficient storing of large volumes of data possible in NoSQL databases, the transactional model is usually relaxed and does not guarantee the same assurances as it is in RDBMS. In NoSQL, performance and scalability is preferred over consistency. In general, scaling can be performed using two distinct techniques:

- *Vertical scaling.* This only means increasing a computational power (e.g. adding memory, CPUs) of a single node so that the database system can be more efficient.

[1]Often referred to as "Not only SQL" or "No to SQL"; however, none of these expansions are precise or fully agreed upon.

- *Horizontal scaling.* Instead of running the software on a single machine, the database is distributed on a number of nodes forming a cluster. The distribution can be done for the purpose of *replicating* the database, i.e. duplicating the data to distribute the load. That is opposed to *partitioning* (or *sharding*) the database, which means dividing the database into disjunctive parts which are then managed by the nodes separately. The two approaches can be combined for best results.

While horizontal scaling is seemingly the cheaper option because no superior hardware is needed to be used, it presents complications with data consistency and synchronization of the nodes.

1.2.1 CAP Theorem

To describe more precisely how the consistency and other important properties of a horizontally scaled system can be balanced, the CAP theorem has been formulated. It states that in a distributed system it is impossible to achieve all three of the following properties:

- *Consistency.* Any two concurrent operations see the data in the same state. This enforces a duplication of all data updates to all nodes in the cluster before the writing transaction is concluded.
- *Availability.* All requests are guaranteed to be answered; in other words, the system is expected to be available at all times.
- *Partition tolerance.* The distributed system has to continue to work even in the event of a failure of a part of the system.

While RDBMS are not partition tolerant, NoSQL databases typically do not guarantee that the data are completely consistent. That is, a write operation executed on one of the nodes does not necessarily have to wait for the update to be propagated across the entire system.

1.2.2 BASE

BASE is a set of properties defined to be a counterpart to much more pessimistic *ACID* (i.e. atomicity, consistency, isolation and durability) transaction model. BASE can be described as:

- *Basically available.* The database appears to work most of the time, although partial failures are tolerated.
- *Soft state.* The state of the system can constantly change, even at times of no requests.
- *Eventual consistency.* The state of the database will eventually become consistent.

This set of properties gives the database system the freedom of not checking that the data is consistent with every processed transaction; consequently, the operation throughput gets higher and horizontal scaling can be put into practice a lot easier. On the other hand, provided answers to database queries need not be always accurate or even successful; the responsibility of assuring higher levels of consistency is shifted onto the application.

1.3 Key-value Databases

Key-value stores are the simplest of NoSQL databases and are oriented towards performance and horizontal scalability. Their name literally describes how key-value stores work – as a hashmap data structure where opaque values are stored and retrieved by a key. The power lies in the simplicity of the store; there is no schema, the values can be of any type and the records are not considered to be related.

In theory, the value is not structured or interpreted in any way and can only be retrieved as a whole. Thus, no aggregated or other similarly advanced queries are supported. Consequently, the need to parse the value is shifted onto the application. There is a small risk presented by this architecture; if the application often retrieves big chunks of data only to use a small portion of the filtered value, it leads to wasting of resources and worse efficiency.

The choice of what entities will work as keys is mostly straightforward, it can be any unique strings coming from the application's domain (e.g. usernames, emails, session IDs, etc.). In case a key is lost or cannot be computed for some reason, key-value stores usually support some sort of full text search or can provide a list of all existing keys.

A lot of attention is understandably paid to scaling. The advantage is that sharding is particularly easy to achieve; it is only needed to hash the key and use a portion of the hash to determine on which node the record should reside. In addition, the data are often replicated to provide some level of fault tolerance. This is done in a way that records are distributed into *buckets* which are duplicated a specified number of times. Each node then stores a determined set of buckets so that no machine is an exact replica of any other. Therefore, it is possible to load-balance more efficiently during the recovery of a collapsed node [23].

1.4 Column-family Stores

Column-family databases are based on a more expressive data model than key-value stores, yet they achieve comparable levels of scalability. A *column* denotes a single key-value pair, and any number of columns can be combined into a *super column* which has a key as its identifier. Furthermore, sets of columns or super columns are stored in a *row* which is identified by its *row key*. Lastly, when a row contains only columns, it is referred to as a *column-family*, whereas a row containing only super columns is called a *super column family.*

This arrangement can be compared to relational tables because there are rows and columns, too. However, the big difference is that in column-family stores the columns do not necessarily have to be organized identically for each row;

consequently, the storage remains efficient even in the case when relational tables would be very sparse. The database works as a nested hashmap; upon a query, firstly the right column-family is identified by the row key, and, subsequently, the value from the appropriate column is found by the provided column key. In case of super columns one extra level of granularity is automatically considered.

Thanks to the higher expressiveness of the model, the application can structure the data to follow an arbitrary pattern (e.g., entities from a domain can be stored as one column family per entity, and any details of the entity are stored in particular columns). Therefore, database queries can contain elementary filtering facilities based on the column values, because the basic column-family structure is known to the database. Nonetheless, aggregated queries are usually not supported.

The columns in a column-family are designed to be mostly accessed together and the database consistency model is often based on this assumption. Therefore, atomicity is guaranteed at the column-family level; conversely, operations spanning multiple rows are not necessarily treated as a single transaction.

1.5 Document-oriented Databases

As the name suggests, document-oriented databases use *documents* as their core entity. The documents are usually retained in collections where they are expected to have a similar schema, although it is not enforced in any way. Each document has a unique identifier which can be assigned by the database or provided by the application. In this aspect document-oriented databases are very similar to key-value stores, because the identifier plays the same role as the key and the document corresponds to the value. What is, however, different, is the way the store looks at the value of the record. Documents have an examinable structure which is distinguishable by the database and can typically contain basic types, arrays, maps and so on. Moreover, the structure is stored and exposed in a specified format, e.g. XML or JSON.

By definition, document-oriented databases should be able to provide an API or query language that allows for retrieving documents based on the content. Queries should support filtering, aggregating, etc. Typically, the database retains an index on the document identifiers so that document retrieval by its key is fast. In addition, adding additional indexes on arbitrary fields in the content is often supported.

Atomicity of transactions is usually guaranteed at a single document level; however, support for transactions of an arbitrary length can be provided, too. Document-oriented databases, similarly to other NoSQL solutions, should be optimized for horizontal scaling, taking advantage of the fact that documents are mutually independent. Therefore, sharding is usually based on the document identifier, or a value of a specified field.

1.6 Graph Databases

In contrast to the other NoSQL implementations, in a graph database the relations between the objects are of primary importance. Graph databases support a *graph model* which allows for a direct persistent storing of the particular objects in the database together with the relations between them. In addition, a GDB should provide an access to query methods that not only deal with the stored objects, but also with the graph structure itself. The best known example of such an operation is *traversal*, which in its most simple form can be used to obtain the neighbors of a specified object, that is, the objects that the specified object is directly related to.

The advantage of having a direct access to heavily interconnected data comes at the cost of very complicated partitioning of the database. To be able to efficiently partition the graph data onto a number of separate machines, the graph must be reorganized so that the smallest possible amount of relations crosses the boundaries of a single machine in the cluster. Algorithms for performing this operation and then keeping the database in the same state after further data changes have not been successfully put into practice; therefore, the problem of efficient partitioning of graph data remains open [23].

1.6.1 Graph Model

The topology of a graph G can be expressed as $G = (V, E)$, where V is the set of *vertices* (also called *nodes*) and E is the set of *edges* (or *relations*). An edge connects two vertices which both have to exist, i.e. no dangling relations are allowed. Finally, a graph can be *directed* or *undirected*, which denotes whether the graph's edges have a direction or not. In addition to the topology, there usually is more information contained within the graph, amount of which distinguishes basic graph models:

- *Labeled graph.* All the edges have a *label* which denotes the type of the edge. There also is a *Vertex-labeled graph* variant where even vertices can have labels.
- *Multi-graph.* Multi-graph is a labeled graph where multiple edges can exist between any two vertices, granted that these edges have different labels.
- *Attributed graph.* Graph elements can have *attributes* (also called *properties*) appended to them to carry additional data. Properties are usually expressed as a map of keys and their associated values.
- *Property graph* is defined as an attributed directed multi-graph and it is the model which is implemented by the majority of graph databases [1].

We note that this is only a limited selection of graph types that are later referred to in this thesis.

1.6.2 Data Structures

The decision of the appropriate data structure for storing the graph, be it in the memory or on a persistent storage, has crucial effects on the resulting performance of the system. However, there is not an architecture which is ideal for all types of graph operations; consequently, the choice of the data structure necessarily depends on the way the application will be mostly used.

Adjacency Matrix

An adjacency matrix is represented as a bi-dimensional array of boolean values, with the size of both the dimensions equal to the number of vertices in the graph. The array is indexed by the identifiers of the vertices and each boolean value determines whether the two corresponding vertices are connected by an edge or not. It is a trivial observation that the adjacency matrix is symmetric when it represents an undirected graph.

This data structure is very efficient for examining the connection of two given vertices and for adding new edges; however, it lacks performance when all neighbors of a specified vertex are required. Moreover, it is highly inefficient in terms of occupied space; especially in the cases when the graph is sparse. Finally, this structure is not well suited for adding vertices because such a task depends on an allocation of space for a new row and a column.

Incidence Matrix

The arrangement of an incidence matrix is similar; it also has a form of a boolean bi-dimensional array. The difference is that the incidence of vertices and edges is recorded; thus, each row of the matrix represents a vertex and each column represents an edge. The incidence of an edge and its two vertices is then indicated by the values in the corresponding column. However, this data structure exhibits similar drawbacks as an adjacency matrix; in addition, its space inefficiency is even worse, because the array's size coincide with the number of edges which is usually higher than that of vertices.

Adjacency List

An adjacency list consists of a collection of lists, where each of these lists symbolizes the identifiers of neighbors of one vertex. This data structure has much lower space requirements in comparison to the matrix representations. The storage efficiency can be further increased by employing compression techniques which are usually based on the similarity of identifiers of neighboring vertices. Moreover, it is a naturally efficient solution for the type of query demanding all neighbors of a vertex, which is often needed when traversals are performed.

However, when it is needed to be decided whether two vertices are connected by an edge, the performance becomes lower because in the worst case scenario the algorithm must iterate over all the records in the list belonging to one of the vertices.

1.6.3 RDF

The Resource Description Framework[2] (RDF) is originally a standard designed by W3C[3] for representing data in the Web. It has become a popular representation for graph data and adopted as a model by a number of graph databases (called RDF stores) even though graph representation was not the primary designation of RDF [2].

[2]Resource Description Framework, `http://www.w3.org/RDF/`

[3]World Wide Web Consortium (W3C), `http://www.w3.org/`

RDF defines *triples* as statements about resources, which consist of subject, predicate and object parts. Such statements can be directly used to model a directed labeled graph where each edge has its label represented by the predicate and the source and target vertices are denoted by the subject or the object respectively. A property graph can also be modeled; however, it is not as straightforward because properties must be stored as additional objects and the relation of a vertex and a property described by another predicate.

Sail

Storage and Inference Layer (Sail) is a low-level interface created by OpenRDF[4] as an abstraction of the accessing methods supported by different RDF engines, so that various RDF stores can be used interchangeably by a single application. Several implementations of Sail include MemoryStore, which uses only memory as a data storage, and NativeStore (further referrred to as NativeSail), which is persistent.

1.6.4 TinkerPop Stack

TinkerPop[5] has been developing a stack of applications designed to simplify and unite the way of working with graph databases engines. Most of the software is created in Java, which is also meant to be the primary accessing language; nevertheless, interfaces in other languages are provided as well. Majority of the best known GDBs support the TinkerPop accessing methods together with their own interfaces, which gives strong hope that it will become a respected standard. A brief description of the most significant parts of the TinkerPop stack now follows.

Blueprints

Blueprints[6] is a set of interfaces enabling applications to take advantage of the complete functionality provided by TinkerPop, once the chosen graph database implements the common API. The interfaces are designed to be very transparent and offer mostly elementary methods which can, however, be combined to build much more complicated queries. Blueprints works with the property graph model; consequently, if a GDB exposes a stronger model, that extra functionality can only be utilized via its own native methods. In summary, Blueprints can be used individually, but its fundamental role is to abstract away the implementation differences of the database systems for the upper layers of the TinkerPop stack.

Rexster

Rexster[7] is a configurable graph server which exposes any Blueprints-compliant graph database through the REST HTTP protocol. A binary protocol, called

[4]`http://www.openrdf.org/`

[5]TinkerPop graph research team, `http://www.tinkerpop.com/`.

[6]Blueprints property graph model interface, `https://github.com/tinkerpop/blueprints/wiki`

[7]Rexster graph server, `https://github.com/tinkerpop/rexster/wiki`

RexPro, is also supported for better performance on a single machine. *Dog House* is a browser-based user interface also offered by Rexster. The whole TinkerPop stack is elegantly interconnected; communication with the Rexster server from a Java application can be performed by instantiating the *RexsterGraph* class and then using the standard Blueprints methods.

TinkerGraph

TinkerGraph[8] is a lightweight in-memory implementation of the property graph model, and is part of the Blueprints package. TinkerGraph doesn't have any support for transactions; it primarily serves only as a reference implementation for Blueprints and its graph model. However, indexing of elements based on their properties is supported and is implemented using a *HashMap* collection.

1.6.5 Graph Query Languages

There are a number of languages designed for querying graph databases; they mainly differ in expressing power and what purpose the are designed to serve. A brief description of the best known graph querying languages follows.

PQL

Pathway Query Language[9] (PQL) is an SQL-like declarative language used for matching subgraphs primarily in biological networks.

GraphQL

GraphQL[10] is a declarative querying and manipulating language considering graphs as the fundamental unit of information. GraphQL defines its own algebra where each operator takes a collection of graphs as an input and returns a collection of graphs generated according to specified rules. Structure of the graphs and even attributes of the graph's elements are considered during the calculation of the queries. GraphQL is intended to be used generally; in other words, it was not specifically designed for a single kind of graph databases.

SPARQL

SPARQL[11] is another declarative language influenced by SQL. It was developed for querying RDF and it is an official W3C Recommendation.

Gremlin

Gremlin[12] is a part of the TinkerPop stack; therefore, it works over all graph databases that implement the Blueprints interface. Gremlin is designed for iterative traversing of the graph which is controlled by the application in a procedural

[8]TinkerGraph implementation of the Blueprints graph interface, `https://github.com/tinkerpop/blueprints/wiki/TinkerGraph`

[9]`http://bioinformatics.oxfordjournals.org/content/21/suppl_2/ii33.abstract`

[10]`http://dl.acm.org/citation.cfm?id=1376660`

[11]`http://www.w3.org/TR/rdf-sparql-query/`

[12]`https://github.com/tinkerpop/gremlin/wiki`

manner. In addition, it supports graph manipulation and all other functionality from Blueprints, because it directly utilizes its methods.

1.7 Summary

In this chapter, we depicted the principles behind NoSQL and briefly explained its key features. Subsequently, the best known representatives of the NoSQL family were summarized – with closer attention being payed to graph databases, their available backend implementations and provided query methods.

2. Graph Database Systems

In the previous chapter we generally characterized Graph databases together with other types of NoSQL database solutions; however, since one of the main goals of this thesis is giving experimental analysis of a selected set of GDB systems, it is necessary to understand what main features and implementation specifics these systems have.

Consequently, in this chapter a detailed description of today's mainstream graph database management systems is provided. We believe that the most interesting properties are:

- API. What interface is exposed by the database system. In other words, how the database can be queried.
- Internals. How exactly the databases' methods of storing data to persistent storage, indexing and caching are implemented.
- Concurrency. What consistency model the databases support, possibly describing the mechanism behind transactions handling if available.
- Scalability. What approaches the database systems take to horizontal scaling; in addition, whether partitioning (or sharding) is supported.

The list of the selected GDB systems with the description of their main features now follows.

2.1 DEX

DEX[1] is a closed-source commercial graph database written in Java with a C++ core. It was first released in 2007 and its goal is to be a high-performance and scalable solution for working with very large graphs [17]. DEX is currently the third most popular graph DBMS today [18].

The graph model this database exposes is called "Labeled and directed attributed multi-graph" because the edges can be either directed or undirected, all elements belong to arbitrary types and there can exist more than one edge between two vertices. DEX provides a native API for Java, C++ and .NET platforms; in addition, it is compliant with Blueprints interface and the database server can be remotely accessed via REST methods. The database can therefore be used in a variety of applications.

DEX uses its own bitmap-based highly compressed native persistent storage which should be very effective and have a small memory footprint thanks to the light and independent data structures [20]. The fundamental data structure used is a "Link" which is a combination of a map and a number of bitmaps that enables fast conversion between an object identifier and its value and vice versa. When object's value is needed, the map is used; whereas the bitmap can return all object identifiers associated with a specified value. The whole graph can then be stored as a following combination of *links*:

[1]DEX graph database, `http://www.sparsity-technologies.com/dex`

- One *link* for each element type which allows conversion between element identifiers and their types (or labels).
- One *link* for each element attribute. The particular values in the *link* are made up by the actual values of the attribute.
- Two *links* for each edge type, one for out-going and one for in-coming edge endpoints. In this case the *link* values represent the identifiers of vertices incident with the particular edge. Therefore, to traverse from a vertex to its neighbors, firstly the appropriate bitmap in the *link* for out-going edges is returned because it contains identifiers of edges connected to the vertex. Then, the map in the other *link* is traversed to finally obtain the identifiers of the vertex's neighbors.

The structure used for the maps is a B+ tree[2], the values are stored as UTF-8 strings and the element identifiers are 37 bit unsigned integers. The identifiers are grouped and compressed in order to significantly reduce the size of the structure [20]. Attribute values can be indexed when required by the application to accelerate the speed of the element scan based on properties; this is ensured by adding the index on the values in the appropriate attribute *link*.

DEX offers a partial ACID transaction support, called "aCiD", because the isolation and atomicity cannot be always guaranteed [17]. The transaction concurrency model is based on the N-readers 1-writer model[3].

There is a DEXHA[4] extension enabling horizontal scaling for larger workloads that are preferably read-mostly. The replication is achieved using the Master/Slave[5] model with the help of Apache Zookeper[6]. In the event of writing data into the database, the current slave immediately synchronizes with the master and then the changes get eventually propagated to the other slaves. Therefore, the database's consistency is relaxed to an eventual consistency in the DEXHA mode. There is lots of work still to be done in this field; for example, when a slave which is in the middle of a write transaction is disconnected from the master, the whole system gets into a lock because the master cannot force the transaction to be rolled back. Furthermore, the system cannot deal with the situation when the master crashes. However, the automatic election of a new master will be implemented in the future [21].

2.2 InfiniteGraph

InfiniteGraph[7] is another commercial graph database written in Java with a C++ core. It was initially released in 2009. The priorities of this graph database system

[2]B+ Tree data structure, `http://www.seanster.com/BplusTree/BplusTree.html`

[3]N-readers 1-writer model allows any number of read transactions to be executed concurrently, whereas a write transaction can exist as the only transaction at a give time

[4]DEX high-availability,
`http://www.sparsity-technologies.com/dex_tutorials4?name=Introduction`

[5]Master/Slave replication. Multi server architecture where one of the servers is a master which has to be communicated with upon any data update. By contrast, reading of data can be performed without the supervision of the master.

[6]Apache Zookeper, `http://zookeeper.apache.org/`

[7]InfiniteGraph, `http://www.objectivity.com/infinitegraph`

lie in scalability, distributed approach, parallel processing and graph partitioning [22].

InfiniteGraph uses a “Labeled directed multigraph” model which also includes bidirectional edges. The database can be exposed through an API in various languages (Java, C++, C#, Python) and provides a distinct Traverser interface which can also be used for distributed traversal queries. This is accompanied with Blueprints support; as a consequence, the database can also be accessed using Rexster or Gremlin facilities.

The database system conforms to a full ACID model. In addition, it also offers relaxing of the consistency for accelerated temporary ingests of data [24]. The graph database uses Objectivity/DB as a backend; and thus adopts its distributed properties allowing scaling and replication. Unfortunately, we could not obtain any more information on this matter because, to our knowledge, there is no accessible technical report or publication with any description of transaction handling or InfiniteGraph internals in general.

2.3 Neo4j

Neo4j[8] is written completely in Java, it is an open-source project and its first release dates back to 2007. Neo4j is a very well known graph database system, in fact currently the most popular one by a great margin [22].

Neo4j features a graph model called “Property graph” which is in reality very similar to the models the afore mentioned databases offer. The native API is exposed through a whole range of different languages, e.g. Java, Python, Ruby, JavaScript, PHP, .NET, etc. There also are many kinds of ways how to query the database, for example via the native Traverser API, or using SPARQL or Cypher query languages[9]. The database system also implements the Blueprints interface and a native REST interface to further expand the versatility of ways how to communicate with the database. Neo4j also supports custom indexes on elements’ properties using external indexing engines, currently employing Apache Lucene[10] as the default engine.

Persistency of Neo4j database is provided by a native graph storage back-end mainly using adjacency lists architecture. The three main components of the graph (i.e. vertices, edges and element properties) are stored in three separate *store files*. Vertices are stored with a pointer to their first edge and the first property. Properties store is a group of linked lists, where there is one linked list per vertex. Finally edges are stored as two double-linked lists (one for each endpoint of the edge) along with the edge’s type and pointers to the current edge’s endpoints and first property. All pointers are expressed in the form of an integer object identifier and all records in the three files have a fixed size and absolute position corresponding to the object identifier. Therefore, addressing in the *store files* can be performed by directly computing the byte where the record is starting. Such arrangement can simplify loading of the data from the persistent storage and enables fast direct traversing of the graph [23]. Furthermore, there is

[8]Neo4j graph database, `http://www.neo4j.org/`

[9]Cypher query language, `http://docs.neo4j.org/chunked/stable/cypher-query-lang.html`

[10]Apache Lucene search engine library, `http://lucene.apache.org/core/`

a two-tiered caching architecture in Neo4j which should limit the number of disk reads. The lower tier, *filesystem cache*, divides the *store files* into pages which are held in the application's virtual memory so that their management can be left up to the operating system. In the higher tear, *object cache*, the particular elements and other graph's objects are kept in a state much more closely resembling the current application needs, which is achieved by analyzing read patterns in the application [23].

Full ACID transaction concept is supported by Neo4j. This is achieved by having in-memory transaction logs and a lock manager applying locks on any database objects altered during the transaction. On successful completion of the transaction the changes in the log are flushed to disk, whereas a transaction rollback means only discarding of the log.

Neo4j database can be distributed on a cloud of servers using the Master/Slave replication model and utilizing Apache Zookeeper to manage the process. Should any write request be performed on a slave node, this slave will synchronize with the master and the updates will be eventually pushed to all the other slaves. As a consequence, the database consistency property is loosen to eventual consistency while the rest of ACID characteristics stays the same [26]. However, the database distribution solution applies only to replication of the data which helps the system to handle higher read load; at the moment Neo4j does not support sharding of the graph [27].

2.4 OrientDB

OrientDB[11] is not only a graph database management system, but a wide-range NoSQL solution providing a key/value store, document-oriented database and finally a graph database. OrientDB is written solely in Java and has become very popular shortly after its initial release in 2010 [22].

The GDB system adopts "Property graph" as its graph model and provides API in many programming languages. Many approaches can also be used to query the database, namely the native Traverser API for Java embedded solutions, REST interface for remote access or an SQL-like query language which is called *Extended SQL* and has been developed alongside OrientDB. The database is also Blueprints compliant and thus several other ways of accessing the engine are available. In addition, unlike the majority of other GDB systems, OrientDB provides support for basic security management which is based on the users and roles model.

OrientDB's graph database is built atop the document database for which a native model of a persistent storage was created. Specifically, the storage is divided into three parts:

- *Data segments.* The contents of all the records are kept here. Each segment is represented by one physical file and it is dependent on the configuration provided by the application how exactly the data is segmented.
- *Clusters.* This part keeps the references to the content in the *Data segments* part of the storage. The usage of clusters is application specific, usually one

[11]OrientDB Graph-Document NoSQL DBMS, `http://www.orientdb.org/`

cluster corresponds to one logical type of data. For the OrientDB graph database implementation, separate clusters are used for the distinct entities in the graph (e.g. vertices, edges, indexes, object identifiers, users, etc.).

- *TxSegment.* Logs for all the open transactions are kept at this place.

Both the *Cluster* and *Segment* parts further consist of two file types, namely *Data files*, which store the data, and *Hole files*, which keep the information about any free space left after records deleted from the *Data files.* This space can be reused during successive allocation. All identifiers of objects in the database consist of two parts, which are the cluster number and position within the cluster. Therefore, whenever a record is to be loaded, the appropriate *Cluster* is queried for the record location within the *Data segments* and that information is used to obtain the record data from there. While the actual data can be accessed locally, remotely, or can be stored only in memory, the principles of the storage stay the same [30].

OrientDB uses its own data structure also for indexing properties of elements. It is an innovative algorithm called *MVRB-Tree* which is a combination of a B+ tree and a Red-Black tree[12] and which consumes only half as much memory as a Red-Black tree while keeping the speed of balancing the structure after an insert or update [28]. The caching mechanism is also quite sophisticated and is spread over several levels; from caches exclusive to a single application thread to caches on the physical storage level. However, each caching level has a different impact on the isolation of the changes of records during the transaction and must be manually configured or periodically invalidated in order to ensure that the data is up to date [29].

Transactions in OrientDB have all ACID properties which is ensured by utilizing the MVCC[13] method. Thanks to this approach, there can be concurrent reads and writes on the same records without the need to lock the database; however, all the records must carry their version so that the age of the record can be checked on a transaction commit [31]. The resolution of any transaction conflicts is left up to the application, which can either repeat or rollback the transaction.

This database system can also be distributed across a number of servers in a cluster, using Hazelcast[14] for the clustering management. Amongst the nodes in the cluster the Multi Master[15] replication method is supported. Therefore, all of the servers are allowed to perform read and write operations on their own replicas of the database while notifying the rest of the nodes in the cluster. Such notification can be synchronous or asynchronous, the latter being faster, however guaranteeing only *eventual consistency* property. Each node keeps its *Operation log* in case an alignment operation with other nodes has to be executed. Furthermore, *Synchronization logs* are kept for the situations when a conflict occurs

[12]Red-Black binary search tree data structure,
`http://cs.wellesley.edu/~cs231/fall01/red-black.pdf`

[13]Multiversion concurrency control method often used by database systems to provide data integrity during a concurrent access by having each transaction work on a different version of the database and merge changes during the commit. `http://clojure.org/refs`

[14]Hazelcast Open Source project, `http://www.hazelcast.com/`

[15]Multi Master replication. Multi server architecture where all the servers have an equal role in the cluster and are allowed to modify the stored data. Any modifications to the data are then propagated to the rest of the servers and any discovered conflicts resolved.

during the replication. The default policy behavior is that the older changes take precedence over the newer ones, which get logged into the *Synchronization log* instead. Entries in the log can be manually resolved later. Similarly to the other graph database systems, the discussed distributed solutions only work with replicas as sharding is not supported by OrientDB yet.

2.5 Titan

Titan[16] is one of the newest graph database systems as it has emerged very recently, in 2012. Similarly to other GDB engines, Titan is written in Java and it is an open-source project. The authors claim that it is a highly scalable solution specialized on handling graphs distributed across a cluster of many servers [33].

The "Property graph" model used by Titan can be accessed via two provided interfaces. Titan can either be run as a standalone server which should be queried by REST methods, or be embedded in the user's Java application for which case it supports the Blueprints interface. Compliance to Blueprints also naturally opens up the Rexster and Gremlin possibilities. Furthermore, to index elements by their properties users can choose between two external indexing engines depending on the application needs. Specifically, the engines are Apache Lucene and ElasticSearch[17] which can be used to perform effective full-text searches or numeric-range and geo searches respectively [34].

As a backend, Titan supports three particular key-value or column-family databases which have a contrasting influence on the transaction and scalability properties of the resulting system. Therefore, it is possible to choose between Cassandra[18], HBase[19] and Berkeley DB[20] depending on the application's business requirements. In particular, selecting Berkeley DB has the effect of very limited horizontal scalability and concurrency; nonetheless, it is the best performing setup for databases running on a single machine. On the other hand, Cassandra and HBase provide native support for distributed solutions at the cost of uncertain consistency, or availability respectively [35]. The main features of the three backends can be seen in Table 2.1.

Backend	**Consistency**	**Scalability**	**Replication**
Cassandra	eventually consistent	linear	yes
HBase	vertex consistent	linear	yes
Berkeley DB	ACID	single machine	not supported

Table 2.1: Backends supported by Titan [35].

In order to store the graph to a persistent storage Titan uses adjacency lists data structure. Namely, one column family in the underlying backend represents one vertex's adjacency list with a row key equal to the vertex's identifier. Moreover, each element property and edge is stored in a single column with the

[16]Titan: Distributed Graph Database, `http://thinkaurelius.github.com/titan/`
[17]ElasticSearch search engine server, `http://www.elasticsearch.org/`
[18]Apache Cassandra key-value store, `http://cassandra.apache.org/`
[19]Apache HBase column-family database, `http://hbase.apache.org/`
[20]Oracle Berkeley DB, `http://www.oracle.com/technetwork/products/berkeleydb`

edge direction, label, or key saved as the column prefix [36]. An index can be created on any vertex property so that the vertex can be retrieved faster when the property's key-value pair is provided. The index is then simply stored in the database as another column family. Lastly, Titan uses a few other optimization techniques (e.g. vertex-centric indexes, edge compression) to accelerate several types of queries against the database [37].

2.6 Summary

This chapter provides the description of selected graph database management systems and their most significant properties. Naturally, there are many more databases in today's market, but as it cannot be in the scope of this thesis to go through them all, only a small representative set of the best known GDB systems was picked.

The most important features of the databases are summarized in Table 2.2. In short, all systems are programmed completely or at least partially in Java, which also is their primary interface. All the databases also implement the Blueprints interface with the advantage of an automatic support for Gremlin and Rexster. While nearly all of the database vendors use a different name for the graph model their database system exposes, it is always effectively equal to the *Property graph* model; consequently, there is no need to make any distinction. Two of the systems use a native persistent storage specifically developed for working with graph data while the rest utilizes existing NoSQL databases. All the evaluated systems support some level of transaction standards and replication strategies; and only InfiniteGraph claims to be able to partition the data. Finally, none of the engines take any support for security into consideration. This, however, does not hold for OrientDB.

	DEX	**InfiniteGr.**	**Neo4j**	**OrientDB**	**Titan**
Language	Java, C++	Java, C++	Java	Java	Java
API	Java, C++, .NET	Java	plenty	plenty	Java, REST
Querying	BP	BP, PQL	BP, Cypher	BP, SQL-like	BP
Graph Model	Property graph (or very similar)				
Persistence	native	Objectivity/DB	native	own Doc. DB	a Key/value DB
Concurrency	aCiD	ACID	ACID	ACID	backend specific
Scalability	Master/Slave replication	sharding	Master/Slave replication	Multi-master	

Table 2.2: Database systems' features [3, 38, 39].

3. Benchmarking

Benchmark is an experimental assessment of the performance of an object, in this case a graph database system. To ensure that a benchmark provides meaningful and trustworthy results, it is necessary to guarantee its fairness and accuracy. Doing so is not always straightforward as different graph database products can target and be optimized for different scenarios. Therefore, a good GDB benchmark should address a wide range of the most usual functionality expected from graph databases while ensuring that the benchmarking environment and the whole process in general is unbiased and deterministic.

The chapter is organized as follows. We begin by stating what the most essential functional requirements from a graph database are nowadays. This will help us to define categories of graph queries which should be examined by a graph database benchmark. Finally, we will focus on the problem of obtaining datasets that each benchmark must logically work with – and abilities of artificial graph data generators.

3.1 Recommended Design

We believe that a comprehensive and thorough benchmark should address the most significant features and usage patterns of the software being tested. As a consequence, benchmarks written for graph database systems must inevitably be different from the ones made for other types of databases. Specifically, graph databases are optimized for working with relations between the stored objects and their more complicated traversals. Although there are other database types considering relations between objects (e.g. Object databases, XML databases), benchmarks created for them cannot be efficiently used also for assessing graph databases, because they do not take advantage of the relations to the necessary extent [4, 6].

To be able to name the recommendations for a benchmark which reflects the most common usages of graph databases, it is crucial to analyze typical operations on graph databases and categorize them by their characteristics. Furthermore, appropriate benchmark setting and results interpretation must be considered.

3.1.1 Graph Operations

There is a number of operations that are expected to be handled by a graph database system and that vary in complexity and other aspects. Firstly, there naturally are the elementary CRUD[1] operations that work with graph elements and their properties. Furthermore, relations between vertices are mainly exploited by the traversal operations; be it the simplest traversal only giving the neighbors of a vertex or complex operations exploring big parts of the graph. Two illustrative traversal operations are *search* and *path* algorithms which go across the graph until a certain condition is met and return the desired visited elements.

[1]CRUD data operations (Create, Read, Update and Delete) and the elementary functions of any persistent storage.

The afore mentioned set of generic operations is often used by applications to compute more specific and complex procedures. Namely, this includes graph analysis algorithms, calculations of graph properties, graph matching methods, etc. These operations differ in the way they access the elements of the graph and must be taken into account at the time of selecting the right representative set of operations to be incorporated into the benchmark.

3.1.2 Query Classification

To be able to select a set of characteristic operations for a benchmark, it is useful to begin with classifying the operations by different criteria respecting particular aspects of dealing with the database. Specifically, there is a number of features in which particular database queries contrast each other:

- Traversing/Non-traversing queries. This categorization is based on whether the operation considers the relationships between vertices or not. While graph databases' main focus lies on vertex relationships and thus on traversal queries, non-traversing operations cannot be ignored either as they are important in many use-cases (e.g. counting elements with a certain attribute).

- Respecting attributes or labels. Whether the query takes element properties into account or not.

- Manipulation/Analysis queries. This means classifying the operations into those transforming the database and those that do not modify the data at all. It should be noted that queries storing any calculated intermediate information into the properties of the graph elements, in fact, manipulate the graph.

- Locality of the queries. We categorize the operations depending on the number of elements processed. Local queries get into contact with only a limited neighborhood of the starting point of the operation, whereas global queries access significantly large parts of the graph.

- Using indexes. Distinguishing whether the operation utilizes any sort of database indexing or not is also of importance.

All these categories should be represented by at least one operation so that the various aspects of graph queries are covered by the resulting benchmark [4].

3.1.3 Benchmarking Process

Setting up the benchmarking environment correctly is crucial for the experiments to be fair. In other words, all the tested graph databases should be run under equal conditions; otherwise, the results are not guaranteed to be accurate. Namely, the testing graph datasets and input parameters, should they be generated at random, must be identical for all the analyzed systems. Equivalently, the order of operations executed on the database must stay the same. The databases also need to be configured correspondingly; for example, to work at the same level of data consistency (having the same transaction properties) and so on.

There are many approaches to interpreting the benchmark results. Firstly, all operations should be run a fixed number of times to provide average or similarly aggregated results that can compensate any possible deviations caused by the running environment. In addition, it is not only the execution time of particular operations that can be measured; metrics like database's persistent storage size or memory footprint of the application can be also analyzed.

3.2 Benchmark Datasets

The right choice of datasets that will be used for running database benchmarks is important to obtain precise and meaningful results. In addition, it is necessary to test the databases on a sufficient number of datasets of different sizes and complexity to get an approximation of the databases' scaling abilities. The selection of appropriate datasets is even more complicated in the graph data context as there are many unforeseen aspects that have to be taken into consideration.

3.2.1 Real Graph Properties

Although graphs are used to represent networks in many diverse fields (e.g. social networks, biological networks, citation networks, recommendation systems and many more), they usually share a number of common properties which cannot be observed in completely random graphs following the Erdös-Rényi[2] model [40, 41]. The most significant properties of real networks are:

- They are *scale-free* which means that the degree distribution of the nodes in the network follows a power law distribution[3]. Therefore, there is a minority of nodes in the graph that have an extraordinarily high degree while most of the vertices are very loosely connected with the rest of the graph.
- The *clustering coefficient* of the network is high, i.e. there are communities of nodes which are connected much more densely within the community than with the rest of the graph.
- They have the *small world property* because the average diameter[4] of the graph tends to be rather low. Consequently, graph traversals are usually not very deep.

3.2.2 Graph Data Generators

In order to estimate a graph database's performance in real situations, it would be most accurate to benchmark the database on the authentic data coming from these situations. However, it is often complicated to obtain data that match the benchmarking application's needs – in terms of both the format and the size of

[2]Erdös-Rényi model of a random graph defines that each edge in that graph appears between a pair of nodes with an equal probability and independently of the other edges in the graph.

[3]Probability distribution of vertices follows the power of their degree, `http://www.sciencemag.org/content/287/5461/2115.short`

[4]Average diameter of a graph is the mean length of the shortest path between any pair of vertices in the graph.

the data. This is the reason why it is generally more convenient to use a graph data generator which synthesizes the data according to provided parameters. Nonetheless, such generator should build artificial graphs that are as similar as possible to the ones created naturally; in other words, the generated graphs should posses the above mentioned real graph properties.

The generators that were found to be utilized in existing graph database benchmarks include:

- R-MAT generator[5]. R-MAT features a transparent and parameterized method of a direct creation of the resulting graph's adjacency matrix [4]. There is a minor limitation coming from the used algorithm; namely, the number of vertices of the target graph must be a power of two. On the other hand, the model can be applied for the generation of weighted, directed and bipartite graphs and, most importantly, the produced networks have the power law distribution property [5].

- Barabasi-Albert model[6]. This graph model iteratively creates the network by adding one vertex at a time until the desired size of the graph is achieved. After a vertex is added, it is connected with a random set of already existing vertices; this set is chosen proportionally to their degree. As a consequence, the power law distribution property is necessarily present in the graph. The algorithm can serve for the creation of both directed and undirected graphs [42].

- LFR-Benchmark generator[7]. Similarly to the above mentioned models, this generator also creates network datasets that have the power law distribution property [6]. Moreover, artificial communities of tightly coupled vertices can be automatically implanted within the generated networks so that the result's resemblance of real data is even stronger [7].

3.3 Summary

In this chapter, the problem of developing a graph database benchmark was depicted and the most elementary recommendations outlined. It is apparent that the most detailed decisions always depend on the actual implementation of the benchmark and cannot be generally summarized. Consequently, this chapter focused mainly on providing the testing data and distributing various operations into categories so that no important aspects of a real graph database usage scenarios are omitted in the resulting benchmark.

In summary, there is a strong importance of traversal operations when working with graph data [6]. In addition, the most important graph features a GDB should be able to work with are: edge labels, edge direction and element attributes [4]. Therefore, assessing the performance of operations addressing these factors should become a center of attraction of a sound graph database benchmark. The

[5]R-MAT recursive artificial graph generator, `http://repository.cmu.edu/compsci/541/`

[6]Barabasi-Albert model is a growing model for generating synthetic scale-free networks.

[7]LFR-Benchmark artificial network generator,
`https://sites.google.com/site/andrealancichinetti/`

provided testing datasets and the benchmarking process should respect these facts.

4. Related Work

In this chapter we describe and compare various existing works presenting a graph databases benchmark. Any advantages and disadvantages will be discussed and taken into account at the time of our own benchmark implementation.

4.1 GraphDB-Bench

GraphDB-Bench[1] is an extensible graph database benchmarking framework built on the TinkerPop stack, heavily using Blueprints, Pipes and Gremlin in order to be easily executed on any Blueprints enabled graph database system. It provides an interface for user-defined graph database testing operations, automatically measures their execution time and logs all results. GraphDB-Bench also contains scripts for an automatic generation of synthetic graphs and then plotting benchmark results, using the iGraph[2] library. In case the users want to provide their own graph data, thanks to Blueprints, input files in the GraphML notation are accepted.

The project was started in the year 2010 and the original intention was to become a main stream unbiased benchmarking tool which is open and easily extensible. It would come to be a part of the TinkerPop stack and be used to benchmark graph database systems automatically with every new version of Blueprints. All the benchmark results would be kept in a public repository for anyone's reference. Therefore, if anyone could contribute with their own sets of tests suites, gradually most of the bias related to benchmarking would be reduced and a reasonably fair benchmarking tool developed [10]. However, GraphDB-Bench has not been further expanded since about 2011 and is nearly a dead project now, adopted only by individuals for their own private testing purposes.

Tested Scenarios

In the currently accessible package, the framework comes out with an example benchmark (and with its results) comprising the following tests:

1. Simple breath first search traversals starting from a number of randomly selected vertices, running to a range of depths.
2. Loading graph data from a GraphML file into the database.
3. Performance of writing into an index. In this test, all the graph's vertices and edges have a string property which is filled in the graph's index.
4. Reading from an index. A number of random vertices are looked up in the database's index based on the vertices' property.

[1]`https://github.com/tinkerpop/tinkubator/tree/master/graphdb-bench`
[2]`http://igraph.sourceforge.net/`

All these tests are run against a dataset of a varying size, from a tiny graph of a thousand vertices to a large graph consisting of a million of vertices. Nonetheless, the benchmark itself claims the results should not be taken too seriously but only as the framework's proof of concept.

Tested Systems

TinkerGraph, OrientDB and Neo4j were tested in this exemplary benchmark. TinkerGraph, which is only an in-memory implementation of the Blueprints interface, was much faster in almost all tests. Apart from that, Neo4j surpassed OrientDB in graph loading and reading from the index. On the other hand, OrientDB was faster during index writes. Interesting situation arrived during the test measuring the performance of traversals. Neo4j was faster in traversals when the depth was up to four and OrientDB began to dominate when the traversal depth was higher than five.

Conclusions

GraphDB-Bench seems to be a very handy tool for quick assessing of any graph database supporting the Blueprints interface. Thanks to Blueprints, any implementation details specific to the actually tested database are abstracted away. The framework is written in Java as it is the primary graph databases accessing language.

The benchmark attached to the distribution of GraphDB-Bench is a small-scale one which is not surprising since it is only an example usage of the framework. But it still shows that such a tool can be used for serious benchmarking in the future.

There is a clear disadvantage of the project. Its dependency on Blueprints and the TinkerPop stack in general does not allow any database which is not Blueprints enabled to be benchmarked by the framework. In addition, for the benchmarks to be really fair, some experts would be needed to fine tune the various GDB systems' configuration for the best performance. The truth is that this problem could by easily bypassed by stating that only basic initial configuration would be considered. This, however, would be inconsistent with the typical graph database usage in a real production environment.

In summary, GraphDB-Bench is a very nice attempt to bring some standard and routine into benchmarking of graph databases. We can only hope that the project will be revived in the future and put into practice again.

4.2 Survey of Graph Database Performance on the HPC-SGAB

In paper [3] the authors describe and implement guidelines from the HPC[3] Scalable Graph Analysis Benchmark [4] which is normally used to assess the performance of graph algorithm libraries.

[3]High Performance Computing.

Tested Scenarios

The benchmark consists of four testing scenarios where the execution time is measured:

1. Loading the graph into the database. This includes all necessary indices as well. The created graph is then used in the following tests and is not changed any more.

2. A query against the database which ignores any relations. They implement a simple query finding all edges having an attribute equal to a certain value.

3. A traversing query. Namely a number of breadth first search traversals to the depth of three which are started from all edges found in the previous test.

4. A query which traverses the whole graph. They calculate the Betwenness centrality[4] of the graph, which is a graph property that can be found only by traversing the whole graph several times. The high number of traversals gives the opportunity the estimate the TEPS[5] value.

The graph data used in this benchmark do not come from a real environment but are artificially created using the R-MAT generator [5]. R-MAT is capable of building a graph with properties that are typical of graphs in real life situations, most importantly having the power law distribution property.

The benchmark is implemented in Java as all the tested databases are either written in Java or provide an interface in it; nevertheless, Blueprints or any similar high level wrapper is not used to the advantage of being able to configure the particular databases to perform as well as possible.

Tested Systems

Four databases were selected for the benchmark, namely Neo4j, DEX, Jena and HypergraphDB. Regarding the results, DEX was shown to be the best performing of the analyzed engines in almost all of the tests. Apart from that, Neo4j performed much better than Jena during traversing operations, and Jena surpassed Neo4j in the first two tests (loading the graph and scanning the edges). HypergraphDB could not be assessed on most of the operations because it was not possible to load the graph into the database for majority of the initial dataset sizes in time.

Conclusions

All tests were conducted for various sizes of the initial graph, including sizes where it is not possible to load the whole graph into the memory. This approach does not only reveal how well the graph databases scale but also how advanced caching techniques they employ. The authors also discuss possible improvements

[4]Betweenness centrality is a measure of each vertex and is defined as ratio of the number of the shortest paths between any pair of vertices that pass through the mentioned vertex to all the shortest paths that do not.

[5]Traversed edges per second, commonly used to express the speed of a traversal.

of the HPC Benchmark which would ensure the four tests would be balanced more thoughtfully in terms of execution time and scaling; thus giving more accurate results. The warm-up time of the operations, which basically means how long it takes for all the necessary data and structures to be cached into the memory, is also described in great detail. In addition, many different metrics are measured and discussed (physical size of the data in the database, TEPS against the graph size, objects loaded per second against the graph size, etc.), which leaves the impression that this article is very thorough and precise.

4.3 Benchmarking Traversal Operations over Graph Databases

The article [6] provides a thorough description of a benchmark created by the same authors. The benchmark is implemented in Java, uses Blueprints API, and most importantly, heavily utilises GraphDB-Bench[6] framework. Graph data used for the testing are synthesised via the LFR-Benchmark [7] generator which produces graphs with properties similar to those of the real world graphs. It generates network data sets with power law distribution property and with artificial communities implanted within the networks [6].

Tested Scenarios

The following tests are described in the paper and their execution time is measured by the benchmark:

1. Loading the graph into the database.
2. Computing the local clustering coefficient of ten thousand randomly chosen vertices. To achieve that, breath first search traversals two hops away from the chosen vertices are needed to be performed.
3. Performing breath first search traversals for three hops from ten thousand randomly chosen vertices.
4. Running an algorithm for detection of connected components which requires a traversal of the whole graph. The algorithm needs to store some intermediate state information as it works its way through the graph. Two possibilities were considered:

   Store the information in memory (not using the database).

   Use the GDB and store the information as a vertex attribute. Retrieve it when needed.

Tested Systems

The benchmark described in this article is run against these GDB systems: Neo4j, DEX, OrientDB, NativeSail and SGDB (which is their own research prototype

[6]GraphDB-Bench benchmarking framework,
`http://code.google.com/p/graphdb-bench/`

of a GDB). The results of the benchmark are preliminary and not very clear, but SGDB seemed to perform significantly better than the rest of the systems. However, SGDB is only a research prototype with some concurrency issues and not a ready product [15]. Amongst the rest of the engines, DEX and Neo4j showed to be more efficient than OrientDB and NativeSail for most of the tested operations.

Conclusions

The tests were also run on graphs of various sizes, where the biggest ones were so large that it was not possible to load them into the actual database in time; and thus were left out from the benchmark. The last of the tests which stores the temporary information required by the algorithm either in memory or the database itself proves the latter to be about an order of magnitude slower; therefore it reveals quite a big room for improvement of the particular systems.

Another interesting observation was shown during the benchmark runs, which is probably related to the way how some of the systems implement the Blueprints API. For traversing, the authors tried to do the expansion either through outgoing or through incoming edges, where both of them should give the same results and execution times. However, half of the tested systems were significantly slower when the approach of outgoing edges was used.

In summary, this paper and the corresponding benchmark are well written and show a nice usage of Blueprints and GraphDB-Bench for benchmarking graph databases. On the other hand, only traversing operations are tested on the GDB systems and any operations ignoring relations and focusing mainly on attributes are missing.

4.4 A Comparison of a Graph Database and a Relational Database

In the article [8] a comparison between a chosen graph database and a more traditional relational database is described in detail. Not only do the authors compare the systems objectively using a benchmark, they also provide a subjective view on other aspects of the systems, such as quality of documentation, support, level of security, ease of programming etc. For the data generation of the benchmark, they come up with their own generator of random directed acyclic graphs. Those graphs, however, do not seem to be satisfying any of the real world graphs' properties (e.g., power law distribution property, small diameter, high clustering coefficient).

Tested Scenarios

Following operations were tested and measured in this benchmark. They divide into two groups, traversal and non-traversal queries:

1. Finding all vertices with the incoming and outgoing degree being equal to zero.

2. Traverse the graph to a depth of 4 from a single starting point.
3. The same as the previous test with the depth of 128.
4. Count all nodes having their random integer payload equal to some value.
5. Count all nodes having their random integer payload lower than some value.
6. Count all nodes whose string payload contains some search string.

   Using completely random strings of a certain length.

   Using a dictionary.

For the non-traversal part of the queries, a few different types of payloads were tried one by one.

The authors also considered various sizes of the random graph to point out how the database systems scale. In addition, they measured the physical size of the DB once all data were loaded; in other words they compared the systems in terms of disk usage efficiency. On the other hand, the loading times of the data into the DB were not covered at all.

Tested Systems

Neo4j was selected as a representative of graph databases and MySQL (accessed via JDBC) was chosen for relational database systems. Regarding the results, Neo4j clearly outperformed MySQL in the traversal tests. Performance results of the non-traversing queries were mixed because MySQL was faster with integer payload handling, in contrast to string payloads where Neo4j was faster in most cases.

Conclusions

The article describes the objective and subjective comparison between the two distinct database systems in a very neat and clear fashion. In also discusses the main reasons behind any performance inconsistencies. On the downside, the traversal tests of MySQL could have been implemented to benefit more from various constructs SQL offers. Furthermore, the set of conducted tests could have been larger to cover more aspects of a typical usage of a graph database system.

In summary, this paper and the attached benchmark show an interesting comparison between selected representatives of an SQL and a graph database system. This is important to support the assertion that graph databases are really performing much better in situations related to graph querying and updating.

4.5 Summary

In this chapter we described a few benchmarks tailored for comparison of graph database systems. Each of them covers slightly different aspects of an overall performance of a GDB and they all even come up with more or less different results. All the benchmarks test traversing of a graph stored in the database and some of them assess the speed of non-traversal queries (e.g., index lookups, filtering based on properties of elements) against the DB system. All of them are written in Java as it has been adopted as a main accessing language by all the tested graph databases. To provide data for the testing all the works choose artificial graph synthesizers over real world datasets; with the advantage of being able to quickly obtain graphs having needed properties and sizes.

5. Final Design of BlueBench

In this chapter the constitution of our own benchmark suite, BlueBench, is discussed and designed. We begin with a brief summary of aspects of existing benchmarks, focusing on their shortcomings. The description of the testing environment and used platform then follows. Finally, the exact tests that BlueBench consists of are specified and the software architecture of the benchmarking project is explained.

5.1 General Goals

During the analysis of existing benchmarks we gathered some important facts that could help in designing BlueBench. In this thesis we will try to confirm the results from those benchmarks and come up with some more interesting tests covering important aspects that have been neglected so far.

To begin with, in all of the existing works various sizes of the input graph data are used – to the advantage of being able to determine how well the particular database systems scale with data size. On the other hand, only size in terms of vertex count is altered. None of the works considered varying the number of edges, or average vertex degree. We will try to accommodate this feature to be able to see how the graph density influences the performance of the assessed database system. Furthermore, none of the described benchmarks were run on real datasets, but only on synthesized graphs. Using generated data provides the convenience of custom selection of the input graph properties; however, one must fully rely on the utilized data generator. In BlueBench at least one set of real data will be used to further support the results obtained from the runs on generated datasets.

Another issue was well simplified in all the existing works. Namely, in all the tests assessing graph traversal performance, only relations between vertices are taken into account. Despite the fact that graph traversing is the most important feature of graph database engines and all the engines support further traversal parameters, such as filtering the traversing path by edge labels or vertex/edge attributes, these parameters were completely ignored. Therefore, BlueBench will also cover situations where the above mentioned attributes are employed for the traversal queries.

There also is some room for improvement regarding the selection of tested DBs. Even though various types of systems (e.g. traditional databases, RDF data stores) were tested in the previous works, the choice still tends to be rather narrow, with maximum being five. In addition, although relational database systems are usually depicted as a typical example of DBs that are outclassed by graph databases when it comes to graph traversals, only one of the mentioned works benchmarked the performance of such a system on graph data. Thus, we will try to include a bigger number of database systems in our benchmark while keeping the versatile selection and including a representative of RDBMS.

In summary, the above mentioned disadvantages of existing benchmarks impose some requirements on BlueBench. Namely, it is the variety of input graph data, also the choice of tested scenarios, and lastly the richness of the selection of tested database engines. By contrast, some of the common aspects of the existing benchmarks seem to be handy; most importantly it is the environment to conduct the tests in and how to interpret the results.

5.2 Used Technologies

For the implementation of the BlueBench benchmarking suite we have selected Java as the main programming language. This was done for the following reasons:

1. Java is without doubt the primary API language for a majority of graph database systems. Many of them are even implemented in Java. This fact gives the advantage of accessing the GDBs using usually the best documented, feature richest and most debugged interface with lots of online sources.

2. The same rule applies to many of the libraries or frameworks dealing with graph data.

3. It is fairly easy to set the Java Virtual Machine (JVM) to reflect the needs of the benchmark suite. From limiting the maximum allowed memory to setting what type of garbage collector will be used.

4. The suite can be effortlessly run on any platform with the JVM installed.

5. Finally, it leaves the opportunity to run any of the GDBs inside the JVM and thus saving interprocess communication and getting more accurate results.

Then it was crucial to decide how to access the particular databases; whether to use their native Java APIs or use Blueprints interface from the Tinkerpop stack. Blueprints could abstract away any implementation details, simplify the part of the suite which communicates with the DBs and most importantly guarantee the fairness of the algorithms; on the other hand, such abstraction also disables the usage of any advanced specific speed-ups the DBs might provide. It was eventually determined that Blueprints would be used primarily because it was nearly impossible to find any common ground between all various interfaces that the DB vendors provide. However, for any situations where at least some of the interfaces followed similar principles the tests were programmed twice using both ways. Therefore, it could be measured which of the implementations performs better.

Another important decision to make was about where to run the tested database systems. As already mentioned, most of the DBs enable their incorporating into the benchmark process, i.e. running them in its JVM. Unfortunately, the databases that do not support this would be put into a slight disadvantage because they would have to be accessed remotely, that implies at higher transfer costs. In the end it was decided to run the systems in the same JVM since the delays in communication with the remotely accessed DBs due to local network usage were hardly noticeable. Furthermore, thanks to Blueprints as the primary GDB interface, the option of running all the tested systems remotely through Rexster was still left open in case it was needed.

Lastly, utilizing Gremlin for most of the graph queries was considered. Gremlin is the graph accessing language from the TinkerPop stack, so it would be trivial to include it in the benchmark suite. However, since no extra functionality is offered by Gremlin because the methods from the Blueprints interface still have to be called after parsing the query, we have decided to use the provided Blueprints interface directly so as to have a better control over the execution.

To sum up, BlueBench is implemented in Java and the database engines are run in the same JVM as the benchmark itself if possible. In addition, BlueBench heavily utilizes Blueprints as the main but not only interface to access the tested GDB systems. The latest version of Blueprints at the time of writing the benchmark suite was 2.2.0.

5.3 Conducted Tests

BlueBench is divided into three different standalone benchmarks which, however, share most of the input data and most of the tested operations. That way the individual operations in the particular benchmarks can be compared across benchmarks while the consistency and fairness is guaranteed. How exactly this is achieved is explained in this section.

5.3.1 Individual Operations

Each benchmark consists of operations which have clearly defined boundaries and execution time of which is always measured. Each operation gets the Blueprints graph (hiding the actual database system implementation) and when it is done, it must leave the database in a consistent state. Every operation can receive any number of arguments which are prepared by the benchmark itself. The set of operations has been selected to reflect most of the requirements from a graph database system and to incorporate operations from very complicated ones to very trivial ones. This is the complete ordered list of all operations used in BlueBench:

1. *DeleteGraph.* This operation completely clears the database along with the data and indexes from the disk. It is essential to run *DeleteGraph* at the very beginning of the benchmark to ensure that the database is in its initial state.

2. *CreateIndexes.* Indexing of vertices by selected property keys is begun. This operation has to be performed before any elements are loaded into the database, as explained in Section 6.3.

3. *LoadGraphML.* This operation inserts elements into the database according to a GraphML input file. Not only does *LoadGraphML* prepare the graph for the rest of the benchmark; equally importantly, the execution time of this operation gives away how efficiently the database system is able to insert data. In addition, most engines work in a transactional mode by default, and not respecting that fact would result into a very low performance as every single insert operation would be wrapped in an individual transaction.

Therefore, to prevent this from happening the transactions are handled manually and work with a buffer of a certain size.

4. *Dijkstra.* The shortest paths between a randomly selected vertex and all other reachable vertices in the graph are computed. Only paths using edges with a certain label are considered. This is the most complex traversal operation in the benchmarks because it works with relations between vertices, edge labels and even edge properties (weight of the edge). In addition, it must inevitably traverse the whole graph.

5. *Traversal.* A simple breadth first search traversal to the depth of five from a given starting vertex is conducted. Unlike the previous operation, only a proportional part of the graph should be visited by this traversal.

6. *TraversalNative.* As opposed to the *Traversal* operation, the native API of the underlying GDB engine is used instead of the standard Blueprints API whenever possible. These two operations get executed with the same parameters; it is thus possible to compare the performance of one against another.

7. *ShortestPath.* This operation computes the shortest path between two vertices while ignoring edge labels and all properties. An important difference between *Traversal* and *ShortestPath* operations is that the former uses *getVertices()* Blueprints method whereas the latter utilizes *getEdges()*. Both methods are crucial for any complicated graph algorithm; however, their implementation and thus performance might be significantly different.

8. *ShortestPathLabeled.* The same as the previous operation; however, only edges with a certain label are considered. On top of that, the path is computed twice because the label restriction eliminates too many results. Firstly, one of the vertices is taken as the source of the path and the other as the target, then vice versa.

9. *FindNeighbors.* This is the most primitive traversal operation – it only finds the closest neighbors of a randomly selected vertex.

10. *FindEdgesByProperty.* This is the first of the non-traversing operations and it browses the graph database while looking for all edges with a certain property equal to some string value. The value is not random but it is chosen from a dictionary of all the values that the edges could posses.

11. *FindVerticesByProperty.* Precisely the same as the previous operation, with the difference that the search is conducted for vertices instead of edges.

12. *UpdateProperties.* This operation tests how efficiently the database engine updates properties of elements. A predefined set of vertices is firstly selected, then this set is divided into pairs, and finally, every two vertices in each pair swap their properties. The initial set is created by adding every visited vertex during a shallow breath first search performed from a randomly chosen starting vertex. However, the creation of the set does not count towards the total execution time of the operation as only updating of

properties should be measured in this test. Similarly to the *LoadGraphML* operation, the transaction buffer is used.

13. *RemoveVertices*. By contrast, this operation tests the performance in deleting vertices. A set of vertices is selected identically to the previous operation, and those vertices are deleted from the graph database. This obviously includes all edges that are attached to them. Again, the transaction buffer is used to speed up the operation.

It should be noted that the graph database is not modified anymore once all the data is loaded, naturally with the exception of the last two operations which test the engines update/remove performance. Also, all the operations except TraversalNative are implemented using the DB systems' Blueprints API whenever possible. Any exceptions to this rule are described in Section 6.3.

5.3.2 Benchmarks

The three benchmarks that are part of BlueBench constitute of individual operations as it is depicted in Table 5.1. Each benchmark focuses on measuring of different aspects of the GDB system's performance and the choice of operations reflects it. As most of the operations are shared amongst the benchmarks, eventually the results could be compared BlueBench wide (i.e. observing the difference of performance of a single test in two different benchmarks).

Operation	Labeled Graph	Property Graph	Indexed Graph
DeleteGraph	Yes	Yes	Yes
CreateIndexes			Yes
LoadGraphML	Yes	Yes	Yes
Dijkstra		Yes	Yes
TraversalNative	Yes	Yes	Yes
Traversal	Yes	Yes	Yes
ShortestPath	Yes	Yes	Yes
ShortestPathLabeled	Yes	Yes	Yes
FindNeighbors	Yes	Yes	Yes
FindEdgesByProperty		Yes	Yes
FindVerticesByProperty		Yes	Yes
UpdateProperties		Yes	Yes
RemoveVertices	Yes	Yes	Yes

Table 5.1: Operations performed in the benchmarks.

Labeled Graph Benchmark

As the name of the benchmark suggests, an input graph with only labels (i.e. without properties) is accepted. The performance of basic traversals on the graph where elements have no payload is measured and analysis of speed of loading and deleting the vertices is included. Labels are taken into account during the traversing.

Property Graph Benchmark

Property graph benchmark accepts the same format of input data; however, it expects that the elements have certain properties to be able to run tests based on them. These tests include traversing queries working with properties (e.g. *Dijkstra*) and also non-traversing queries where the relationships between vertices could be ignored (e.g. *FindEdgesByProperty*). The *UpdateProperties* operation is also included not only to measure how quickly the DB systems can read properties, but also write.

All the traversing tests from the Labeled graph benchmark are left in place to be able to observe whether the presence of properties in the graph will have any effect on the performance of operations that do not work with properties at all.

Indexed Graph Benchmark

This benchmark will be run on the precisely same data as the Property graph benchmark. The only noticeable, although substantial, difference is the inclusion of the *CreateIndexes* operation which sets the database engines to start indexing values on some of the properties. This step should presumably influence the performance of most of the successive operations, either positively or negatively. For example, tests working directly with properties (e.g. *FindEdgesByProperty*) should be completed much faster; on the other hand, tests directly changing the indexed data (e.g. *LoadGraphML*) will have to take the indexes into consideration and thus be noticeably slowed down. When the execution of the benchmark is finished, it will be possible to compare the results to those of Property Graph Benchmark in order to monitor how well the indexes are handled in various DB systems.

Blueprints supports two ways of indexing elements' properties in the database through *IndexableGraph* and *KeyIndexableGraph* interfaces [16]. The former one is less automatic and supports specific querying techniques with added parameters, e.g. to achieve case insensitive searching. Nonetheless, it is implemented by fewer database systems than *KeyIndexableGraph*, which is why we had to select *KeyIndexableGraph* interface to facilitate indexing of the elements in the graph database.

5.4 BlueBench Software Architecture

BlueBench is directly patterned on GraphDB-Bench and adopts all main architectural features of the framework. The concept of Operations, Operation Factories and Benchmarks was not changed because the core of the work on BlueBench was adding new functionality, support for more database engines and various graph operations. Although basically all source files had to be modified to accommodate the new functionality, original GraphDB-Bench classes were left in the *com.tinkerpop.bench* package and any newly created classes were saved under package *cz.cuni.mff.bluebench*. The most important design decisions and features are described in this section, accompanied with slightly concise UML diagrams.

5.4.1 Operation

Operation is an abstract ancestor for all measurable actions that are executed on the currently assessed database. Before executing, each *Operation* is set with its arguments which is an array of strings. The actual run of the *Operation* consists of three phases: initialization, execution and conclusion. These three phases are represented by corresponding methods, always call their counterpart which should be overridden by child classes, and are executed one by one by the class running the benchmark. While initialization or conclusion phases only prepare the object with the arguments and do the cleanup respectively, the performance of the execution phase is measured.

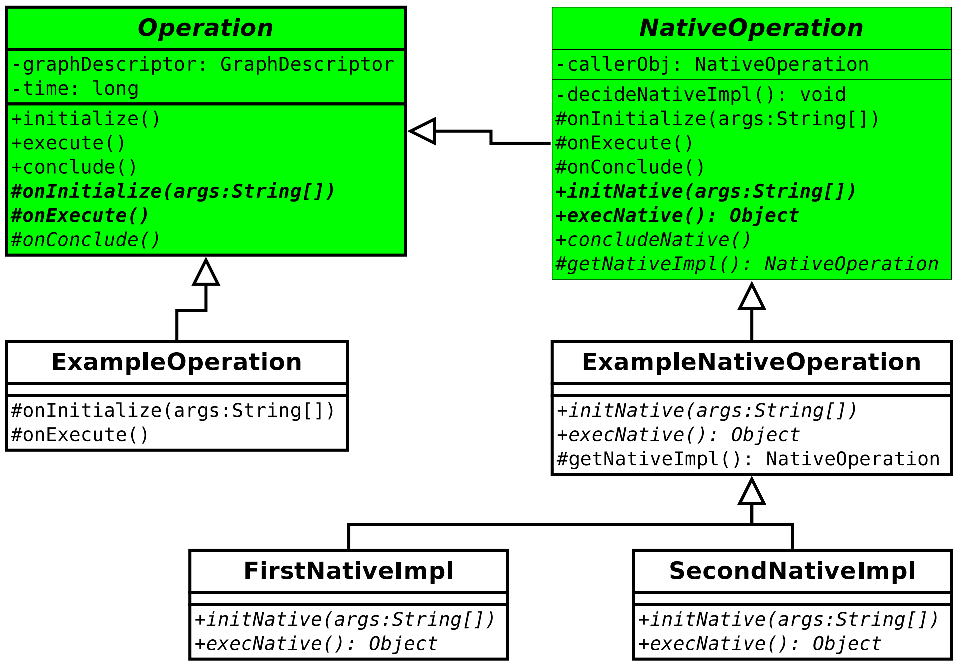

Operation
-graphDescriptor: GraphDescriptor
-time: long
+initialize()
+execute()
+conclude()
#onInitialize(args:String[])
#onExecute()
#onConclude()
NativeOperation
-callerObj: NativeOperation
-decideNativeImpl(): void
#onInitialize(args:String[])
#onExecute()
#onConclude()
+initNative(args:String[])
+execNative(): Object
+concludeNative()
#getNativeImpl(): NativeOperation
ExampleOperation
#onInitialize(args:String[])
#onExecute()
ExampleNativeOperation
+initNative(args:String[])
+execNative(): Object
#getNativeImpl(): NativeOperation
FirstNativeImpl
+initNative(args:String[])
+execNative(): Object
SecondNativeImpl
+initNative(args:String[])
+execNative(): Object

Figure 5.1: UML diagram showing *Operation* class hierarchy.

For the reason that sometimes it is necessary to approach particular DB engines differently, the *NativeOperation* class was added. Any operation directly extending this class serves as a wrapper for any specific implementations of the initialization, execution and conclusion phases, while it still could provide the phases' default behavior. This is achieved by the *decideNativeImpl()* method which checks the class type of the graph that is currently worked upon, and initializes the *callerObj* object with the result of the appropriate version of *getNativeImpl()* method. This is performed only once during the operation's *onInitialize()* phase, and from then on the methods for the three phases are supplemented by *initNative()*, *execNative()* and *conludeNative()* respectively. It is convenient to override the methods in a designated inner class such as in this example:

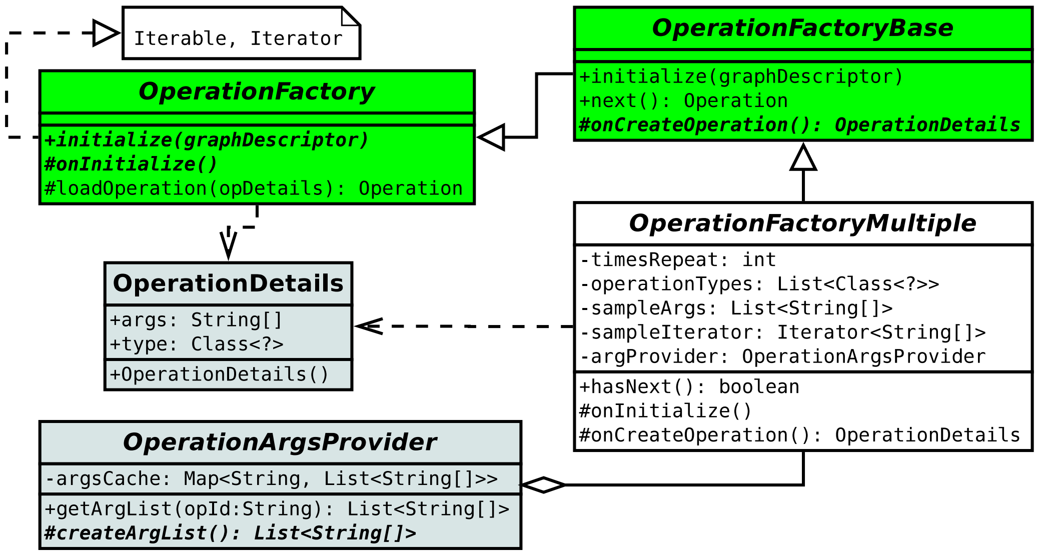
```
...
@Override
protected NativeOperation getNativeNeo4jImpl() {
        return new Neo4j();
}
```

```
private class Neo4j extends TraversalNative {
        @Override
        public void initNative(String[] args) {
 ...
```

There are two scenarios when *NativeOperation* can be used with an advantage; either in the case of any implementation specifics imposed by the tested database engine, or for assessing some special functionality of the underlying database system that is not covered by the common interface. The class hierarchy of *Operation* is shown in Figure 5.1.

5.4.2 Operation Factory

OperationFactory is a class implementing *Iterable* and *Iterator* interfaces so that it can by conveniently iterated in a "foreach" statement to successively return particular Operations that are contained in the factory. A pair of methods *initialize()* and *onInitialize()* is used to achieve the same effect as it is with Operations. Method *loadOperation()* constructs a new Operation with the given type and arguments; these are acquired by calling *onCreateOperation()* method. To help with carrying these details between various classes the class *OperationDetails* is provided.



Iterable, Iterator
OperationFactory
+initialize(graphDescriptor)
#onInitialize()
#loadOperation(opDetails): Operation
OperationFactoryBase
+initialize(graphDescriptor)
+next(): Operation
#onCreateOperation(): OperationDetails
OperationDetails
+args: String[]
+type: Class<?>
+OperationDetails()
OperationFactoryMultiple
-timesRepeat: int
-operationTypes: List<Class<?>>
-sampleArgs: List<String[]>
-sampleIterator: Iterator<String[]>
-argProvider: OperationArgsProvider
+hasNext(): boolean
#onInitialize()
#onCreateOperation(): OperationDetails
OperationArgsProvider
-argsCache: Map<String, List<String[]>>
+getArgList(opId:String): List<String[]>
#createArgList(): List<String[]>

Figure 5.2: *OperationFactory* and related classes.

The primary task of a descendant of the *OperationFactoryBase* class is gradually providing arguments and a single *Operation* which will use them. For example, this behavior can be used to execute the same test multiple times, although with different random parameters. However, a more advanced factory was needed to implement some more complex comparisons of the DB systems. To assure that two distinct operations can be run with identical arguments, the *OperationFactoryMultiple* was introduced. This factory works similarly to the basic one; the difference is that it is initialized with an array of *Operation* classes and a list of arguments. Then, when *next()* is called, the current operation and the following argument are returned unless this was the last argument of the list. If that is that case, the list of arguments is reset and the operation array pointer advanced by one. Another capability of *OperationFactoryMultiple* is giving the exactly same set of parameters for each consecutive run of a particular *Operation*. This is achieved by *OperationArgsProvider* class which caches each list of arguments once it has been required by a factory. Without such functionality it would not be possible to compare performance results of the same tests in different benchmarks (e.g. when the DB has set indexes and when it does not) because the parameters of the tests would not be corresponding.

OperationFactory and the other mentioned classes are depicted in Figure 5.2.

5.4.3 Core Benchmark Classes

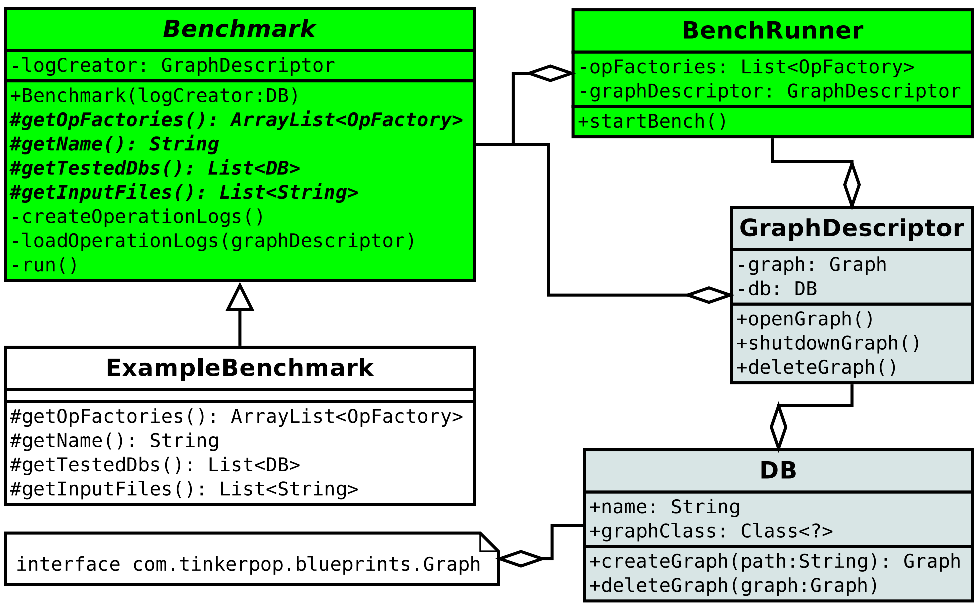
As can be seen in Figure 5.3, each concrete benchmark implements methods to provide sufficient information about the testing; namely what operations and operation factories will be used, what DB systems will be tested and what initial datasets will the systems be filled with. So, when a *Benchmark* object is constructed, the *run()* method is automatically executed on the given set of tested DB systems, and inside that method a new instance of *BenchRunner* class is created to perform all provided operations on the DBs one by one. There are three special operations used to ensure that the execution of every factory and its operations has equal starting points. Specifically, the *Graph* is always closed and execution of a garbage collector attempted after a factory has finished, and right before the start of another factory, the *Graph* is opened again. This is taken care of inside *BenchRunner*.

To alleviate the complexity of *BenchRunner*, the *getOperationFactories()* method already combines all operations with all particular types of input data and returns a joined list of factories ready to process. Therefore, it is not necessary for the list of initial datasets to leave the scope of the concrete implementation of a benchmark and *BenchRunner* can be used only to iterate through the provided list of operation factories.

There are two helper classes to simplify work with different DB engines and hide their implementation. *GraphDescriptor* provides methods to safely open, close and delete a Blueprints graph. By contrast, *DB* is really an "enum" which describes particular DB systems and contains core methods for manipulation with the underlying engines.

5.4.4 Logging of Operations

Execution time and result of every *Operation* is stored in "csv" files, one file per DB system and benchmark. However, operation logging is also used to record the accurate sequence of operations and their input arguments. This is performed at the beginning of each benchmark by the *createOperationLogs()* method; after that, every successive execution gathers the data from the logs using method *loadOperationLogs*. Thanks to that, it is evident that the operations order and the arguments are identical for every tested DB system; in addition, the benchmarks can be either entirely or partially re-run as long as the logs are kept. It also means that it is not necessary to process the operation factories obtained by



Benchmark
-logCreator: GraphDescriptor
+Benchmark(logCreator:DB)
#getOpFactories(): ArrayList<OpFactory>
#getName(): String
#getTestedDbs(): List<DB>
#getInputFiles(): List<String>
-createOperationLogs()
-loadOperationLogs(graphDescriptor)
-run()
BenchRunner
-opFactories: List<OpFactory>
-graphDescriptor: GraphDescriptor
+startBench()
GraphDescriptor
-graph: Graph
-db: DB
+openGraph()
+shutdownGraph()
+deleteGraph()
ExampleBenchmark
#getOpFactories(): ArrayList<OpFactory>
#getName(): String
#getTestedDbs(): List<DB>
#getInputFiles(): List<String>
DB
+name: String
+graphClass: Class<?>
+createGraph(path:String): Graph
+deleteGraph(graph:Graph)
interface com.tinkerpop.blueprints.Graph

Figure 5.3: UML digram of *Benchmark* and related classes.

getOperationFactories with every run of *BenchRunner*, but only during the execution of *createOperationLogs()* instead. Each regular iteration of the benchmark is served by a special factory *OperationFactoryLog* which is employed to read the operations from the log and pass them to the *BenchRunner*.

Some DB system must be provided for the creation of the operation logs because *BenchRunner* works equivalently during the logs creating and logs loading phases. The chosen system is passed to *Benchmark* in the constructor; nonetheless, it is not important what system is selected since the performance is not taken into account during the logs creation. As a result, we have decided to create a trivial implementation of the *Graph* Blueprints interface, called *DummyGraph*, which could be used for very fast logs creation, because no results are computed during the execution of the implemented methods.

5.4.5 Loading the Graph

Populating the assessed graph database with graph data can be taken as an example of *NativeOperation* utilization. Some differences are presented by several DB engines and it was not possible to load the data in any unified way. Therefore we have slightly rewritten the original Blueprints *GraphMLReader* and modified it to accept an implementation of *GraphFiller* interface as a parameter. Because the graph database is filled with data in a separate operation, it was convenient to turn the operation into a native one and make it implement the *GraphFiller* interface. As a result, methods for adding vertices or edges, committing transactions and so on can be overridden to match the particular requirements of the GDB engines.

5.4.6 Addressing Vertices across Scopes

Because BlueBench is a graph database benchmark suite, many operations deal with the graph structure and begin on a preset vertex or edge. This starting element should most often be decided at random and be identical during all the test runs. It was important to decide how and in what form to keep identifiers to those elements between runs of different benchmarks on different DB systems. To slightly alleviate the problem, it has been decided to use only vertices for this kind of task.

The first trivial solution would be identifying the vertices by the IDs they were given upon the insertion into the database. However, according to the Blueprints specification, the ID can be selected by the database engine itself so the provided one could be completely ignored. A partial solution to this problem is the *IdGraph* template class[1] which could be used as a wrapper around those *Graph* implementations that ignore the given custom ID. The solution is said to be partial because for the functionality to work, the underlying *Graph* must be an instance of *KeyIndexableGraph* so that the vertices can be indexed by the wrapper. Unfortunately, not all of the DB systems we want to test implement the mentioned *KeyIndexableGraph*.

Another plausible workaround might be using vertex properties to hold their unique identifiers. It would then be effortless to select the right vertex by calling the Blueprints *getVertices()* API. Nonetheless, since we were considering including RDF data stores into BlueBench, it was impossible to utilize properties for any task. For the same reason it is not possible to obtain a set of all vertices in a predefined order and then iterate through, not to mention the performance of such a query.

At last, we have decided to use our own helper "Map" data structure which would contain pairs of vertex IDs, one of them being a simple number and the other one the real identifier that the vertex was given at the time of inserting the vertex into the database. This map is always filled at the beginning of the DB benchmarking when the graph data are loaded into the DB and kept for all successive operations. This approach, however, poses a minor limitation; it will not be possible to use any *BatchGraph*[2] implementations because in that case the vertex ID does not necessarily have to be determined during the insertion.

5.5 Summary

In this chapter we discussed the most important design decisions that led to the actual implementation of BlueBench. In summary, BlueBench is a benchmarking suite consisting of three standalone benchmarks, focusing on traversing, element properties and element indexing respectively. BlueBench is a direct extension of GraphDB-Bench framework and therefore it adopts its main features, such as using Java as the programming language and Blueprints as the common interface for accessing the particular underlying DB systems.

[1]Blueprints IdGraph,
`https://github.com/tinkerpop/blueprints/wiki/Id-Implementation`

[2]Blueprints Batch,
`https://github.com/tinkerpop/blueprints/wiki/Batch-Implementation`

6. BlueBench Results

In this chapter we begin with an explanation in what environment and on what data BlueBench was executed. This is followed by a list of all the GDB systems that were assessed, which includes any of their implementation specifics and necessary workarounds coming from them. Lastly, the results from the benchmark are described and plotted.

6.1 Testing Environment

The tests were executed on a computer equipped with a single core Intel Xeon E5450 running at 3.00 GHz and 16GB of RAM, with Ubuntu Server 12.10[1]. The Java Virtual Machine of version 1.7 was started using the default parameterization except for the starting and maximum heap size, set to 12GB. The remaining 4GB of memory were left for other running system processes and, most importantly, for two of the tested DB systems which could not be run inside the JVM. Further description of this issue follows in Section 6.3.

6.2 Input Data

In BlueBench both artificially generated data and real graph datasets were used to fill up the tested databases.

Specifically, the first benchmark which completely ignores how the databases handle properties was run twice. Once on synthesized data and then on data collected from Amazon's co-purchasing network[2] as it looked like in the year 2003. This network forms a directed graph consisting of more than 260.000 vertices and about 1.234.000 edges, where vertices represent products and a directed edge from product *a* to product *b* means that product *a* was frequently purchased together with the product *b*. This dataset was selected for the benchmark because of its appropriate size and also because the co-purchasing graph posseses typical real-world network properties (e.g. power law distribution, high clustering coefficient)[9].

For the rest of the benchmarks, which also work with vertex and edge properties, no suitable real datasets were found, so only artificially generated graphs were used. The generation was performed with the igraph[3] library, namely the synthesizer implementing the Barabasi-Albert model. String labels were added to all edges once the graph had been constructed. Afterwards, for all the benchmarks with the exception of the first one, several properties were appended to the edges and vertices of the graph so that all elements had one string and one integer property.

All the benchmarks were executed on artificial directed graphs having 1, 50, 100 and 200 thousand vertices and an approximate mean vertex degree equal to

[1]Ubuntu Server Quantal Quetzal, `http://www.ubuntu.com/`

[2]Amazon product co-purchasing network, collected on 2nd of March 2003, `http://snap.stanford.edu/data/amazon0302.html`

[3]The igraph library, `http://igraph.sourceforge.net/`

5 and 10. This gives the total of eight different graphs with sizes ranging from only about 6.000 elements to as many as 2.200.000 elements.

The input data were stored in the GraphML file format for it being one of the standards for handling graph data. The format is supported by the igraph library and it is easy to be read in the application using only XML parsing standard functions. GraphML might be a little slower to parse for its slightly smaller storage efficiency; however, since this is the only file format used for input graph data in the benchmark, at least in this aspect the fairness could be guaranteed.

6.3 Assessed DB Systems

During the first benchmark, where properties are ignored, the following DB systems were assessed: DEX 4.7.0[4], InfiniteGraph 3.0.0[5], MongoDB 2.2.3[6], our own prototype MysqlGraph, NativeSail 2.6.4[7], Neo4j 1.8.1[8], OrientDB 1.3.0[9], TinkerGraph 2.2.0[10] and Titan 0.2.0[11]. Thanks to the size of this set none of the best known DB systems are left out, and several different technologies are represented (e.g. typical GDB, RDF store, Document, ...). In the rest of BlueBench all of these databases except NativeSail were benchmarked as it is not possible to naturally work with element properties in NativeSail DB.

Specific configuration can obviously be set for all of the systems to achieve the best performance. However, the benchmark would not be fair if some of the systems suffered from any possible lack of proper configuration and some did not. Therefore, to avoid any fairness issues caused by insufficient knowledge of the right configuration of the systems, we have decided to strictly use the initial configuration only. Finally, all the GDBs were run with their transaction mode enabled in order to be able to observe performance of their consistency ensuring methods.

Although the DB systems share a common Blueprints interface, there still are many differences in the way of how it is necessary to work with the engines. More detailed commentary on why particular database systems were chosen and description of any of their implementation specifics are laid out in this section. General description of the systems is in Chapter 2.

DEX

This system had to be included in BlueBench for it being a well-established piece of software in the graph database world. Also, the results of DEX in all of the described existing benchmarks were very promising. In addition, there are implementations of the most common graph algorithms offered by DEX; therefore, performance of their graph traversal method was measured in the benchmarks, too.

[4]`http://sparsity-technologies.com/dex.php`
[5]`http://www.objectivity.com/products/infinitegraph/`
[6]`http://www.mongodb.org/`
[7]`http://www.openrdf.org`
[8]`http://www.neo4j.org/`
[9]`http://www.orientdb.org/`
[10]`https://github.com/tinkerpop/blueprints/wiki/TinkerGraph`
[11]`http://thinkaurelius.github.com/titan/`

On the other hand, Blueprints interface is not absolutely respected by DEX and there are some peculiarities that had to be taken into account during the implementation. Firstly, any collections of returned vertices or edges implement *CloseableIterable* and must be explicitly closed to free native resources of the engine when the work with the collection is finished. Furthermore, vertices have labels in DEX similarly to edges. This had to be kept in mind as preceding setting of the vertex labels is required for some important features to work, such as creating custom indexes in the graph.

InfiniteGraph

Although InfiniteGraph has been around for quite a long time, we could not find a single benchmark comparing it to other GDB systems. This might be the reason of a quite specific interface this DB system provides. Despite the fact that the Blueprints interface is also supported by InfiniteGraph, many operations with the DB have to implemented completely differently, most often using native methods of the engine. This includes working with transactions, where, as opposed to Blueprints, a transaction has to be both started and stopped manually in order to enclose multiple operations in it. In addition, handling with custom element indexes must be done entirely through the native API because InfiniteGraph does not conform to the Blueprints interfaces that are designed to make the work with indexes easier.

Most importantly, the database does not naturally support labels; custom Java classes for all vertices and edges must be created instead. This implies that any operations adding vertices or edges to the graph or setting element properties have to be done uniquely to the current implementation and Blueprints API obviously cannot be used. Moreover, the native InfiniteGraph API had to be used to execute a lot of other basic operations (e.g. finding elements with a certain property). Therefore, quite extensive workarounds had to be implemented to be able to incorporate InfiniteGraph into BlueBench.

A conceptually different interface of finding paths in the graph is contained within InfiniteGraph. It could be elegantly used to perform graph traversals as well; thus we added assessing of this operation's efficiency to BlueBench.

MongoDB

Although it might be interesting to see how a non-graph NoSQL database would perform in graph related tasks, such comparison has not been conducted in any existing work. We have decided to include a NoSQL system in BlueBench to reveal how big the difference in performance could be.

MongoDB was chosen as a typical example of a Document database system employed for working with graph data. There currently is one more non-graph NoSQL engine with an accessible Blueprints wrapper, which is ArangoDB[12]. However, at the time of implementation of BlueBench it was discovered that some of the wrapper methods were performing very poorly or even returning wrong results; and that includes the most important functionality, such as returning

[12]ArangoDB, `http://www.arangodb.org/`

neighbors of a vertex or returning all edges in the graph having a certain property. Last but not least, MongoDB is currently the best ranked software of all document-stores [19].

This is the reason why MongoDB was chosen over ArangoDB as a representative of the non-graph NoSQL world in this benchmark. There is only a slight disadvantage for this engine in terms of competitiveness with other DB systems coming from the fact that MongoDB cannot be run inside the Java Virtual Machine to avoid latency caused by inter-process communication. Nonetheless, this latency should not be extensive as all the processes will be run on the same computer.

MysqlGraph

In order to be able to compare the performance of a relational database system with graph database systems, we implemented the Blueprints interface with MySQL 5.5.29[13] and included the newly created GDB, simply called MysqlGraph, in the benchmarks. Having MysqlGraph in BlueBench can further support the claims that relational database systems are not suitable for graph related queries; on the other hand, it would be interesting to observe that some of the GDB systems perform worse in some scenarios, should such situation occur. MySQL was chosen because of its widespread use in both academic and commercial fields. Other reasons for the selection include the fact that it is not an object-relational database like Oracle[14] but a pure relational database [11]. Also, an extensive support is provided to MySQL users which could greatly simplify the process of implementing the application.

The MySQL server can be run in a standalone process and connected to the rest of the program via JDBC[15]. Similarly to MongoDB, there could be some delays coming from the communication between processes; nonetheless, such setup of the access to a MySQL database is very usual across application fields and technologies.

The architecture of MysqlGraph database is depicted in Figure 6.1. This arrangement allows for a very straightforward implementation of the Blueprints methods while keeping the queries efficient. It should be noted that the indexes in the properties tables are required by the architecture to ensure the uniqueness of property names for particular elements. Therefore, any calls to Blueprints API for adding or removing indexes on properties are completely ignored and the database system should perform identically regardless of the indexes being used by the application or not. Another thing that could catch the attention is the type of properties' values; it is only a string with a set maximum length. In reality, the format should be *TEXT* or similar to allow for unlimited size of properties, yet such a decision would definitely degrade the performance. We have decided to keep the *VARCHAR* format and leave open the possibility of extending the schema by adding another table for very long strings having *TEXT* type.

[13] `http://www.mysql.com/`

[14] Oracle database, `http://www.oracle.com/us/products/database/overview/index.html`

[15] The Java Database Connectivity, `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`

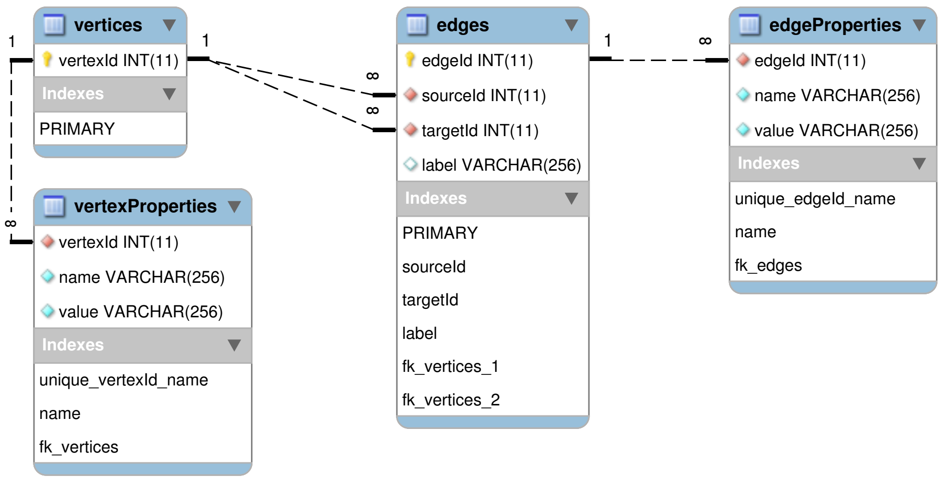


Figure 6.1: EER model showing the design of MysqlGraph database.

As it is with true graph database engines, native implementations of some algorithms are usually provided. We have considered writing the basic traversal algorithm using either *joins* or *inner selects*; unfortunately, such implementation would be a little too complicated and very probably inefficient in comparison to the original way the algorithm is conducted in BlueBench. Thus, we have decided only to stick to the Blueprints interface.

NativeSail

To represent RDF stores in the benchmarks, assessing of NativeSail was incorporated into BlueBench. NativeSail was selected because Sail is the only RDF product currently supporting the Blueprints interface and NativeSail is the only alternative of Sail implementations that uses a disk to store the data.

There are some restrictions coming from NativeSail being an RDF store. Firstly, there are syntactic constraints on the format of vertex identifiers and edge labels; these must respect the *URI* format. This is the reason why all graph datasets used in BlueBench follow this convention. Another peculiarity of RDF stores is that element properties are not natively supported. In order to achieve having properties, they would have to be simulated by adding a vertex per property and connecting this vertex by an edge with the original parameterized vertex. Then all Blueprints methods working with properties would have to be overridden. We have decided that applying such workaround usable for only one DB system should not be in the scope of this thesis and removed NativeSail from the second (properties-testing) benchmark instead. In addition, no custom indexes can be used while working with Sail; this in combination with the properties restriction automatically disqualifies NativeSail from the third benchmark also.

In summary, it was decided that NativeSail performance would be assessed only on the tests of the first benchmark.

Neo4j

Neo4j is one of the best known and most popular graph database systems; on top of that, its results came out considerably good when assessed in any of the existing benchmarks. Therefore, there was no question whether to include Neo4j in BlueBench or not. Neo4j even provides several algorithm implementations out of the box, graph traversal being one of them, so we have added Neo4j's traversal implementation test into BlueBench to measure its performance.

Including Neo4j in the benchmarks was exceptionally straightforward as there is no significant deviation from the Blueprints interface presented by this graph database system.

OrientDB

OrientDB is another well-established software in today's graph database market. It provides pretty flawless conformity with the Blueprints interface so incorporating OrientDB into BlueBench was easy. However, there is one exception; namely that at the time of running OrientDB we could not achieve to maintain the transaction buffer as large as for the rest of the DB systems. The bigger the buffer is, the less often fresh data need to be committed and thus the performance is higher. Therefore, we lowered the size of the buffer for OrientDB only not to have the rest of the systems suffer from the issue.

There is a native implementation of the traversal algorithm included in OrientDB and exposed through the provided Java API; nevertheless, we could not add the assessment of the method's performance into BlueBench because the results that were being returned from the method were evidently incorrect.

TinkerGraph

TinkerGraph is an in-memory direct implementation of the Blueprints interface and is a part of the TinkerPop stack. Not having to work with the secondary storage obviously gives big advantage in performance; however, it limits the size of the database to the memory size. Therefore, inclusion of TinkerGraph in BlueBench should be taken only as a proof of concept and to show what a supposedly ideal performance would be like.

Titan

Titan is a quite new graph database system, taking Blueprints as the primary interface to work with the graph database. This fact promises that this DB system has been optimized for the Blueprints methods and so the performance in BlueBench would be high. It is possible to choose from various storage back-ends when working with Titan; we have selected Oracle Berkeley DB 5.0.58[16]. This way the best possible performance should be guaranteed [12] because the database will be used on a single server inside the JVM. More detailed description of allowed back-ends and Titan in general is in Chapter 2.

There are some limitations imposed by Titan. To begin with, all elements obtained from the database are automatically invalidated every time a new transaction is started. Therefore it is required to re-get any element which is needed to be used again if the application is not in the same transaction context anymore. This influences the beginning of every benchmark when all the vertices and edges

[16]Oracle Berkeley DB, `http://www.oracle.com/technetwork/products/berkeleydb`

are inserted into the database, and also the tests where lots of elements are updated or deleted. As a solution, every cached vertex or edge is always refreshed before it is used again. It would be a little too complicated and with a small benefit to keep a set of elements that are valid in the current transaction because such approach would only be a substitute for a similar structure used by Titan itself [13].

Other limitations are related to index handling. Firstly, as of Titan 0.2.0, indexing edge properties is not supported and thus only vertex index is used by the indexing benchmark while creating indexes in Titan database. This workaround has two consequences, namely inserting and deleting edges from the database will be a little faster because the index can be ignored; on the other hand, looking up edges by their property values will be slower because there is not any index that could speed up the query. This will have to be taken into account at the time of analyzing the benchmark results. The second index-related restriction, which imposed some changes in BlueBench architecture, requires that an index for any property name must be created prior to the first usage of that property. Which specifically means that all the indexes must be created before the graph database is filled with data; therefore it will be pointless to measure the time of the indexes creation and we will instead try to compare the graph loading times with indexes enabled and with indexes disabled. Nonetheless, these two limitations accompanying indexes are only temporary [14]; as a consequence, the workarounds can be removed from BlueBench in the future.

6.4 Results of the Tests

We examined the performance of the graph database systems on the operations listed in Subsection 5.3.1, with the exception of *DeleteGraph* and *CreateIndexes*. These two operations were excluded from BlueBench assessments for a number of reasons. First, it would be complicated to run the tests multiple times, which is necessary to guarantee a better precision. Second, such functionality is not needed to be run very often in any kind of application; and when it is run, the execution speed is not a factor. Third, *DeleteGraph* had to be implemented individually for each system; and, thus, the performance would be incomparable. Lastly, due to restrictions coming from the limitations of Titan, *CreateIndexes* must be run when the database is clean of any objects, so it would not make much sense analyzing it.

Each operation except *LoadGraphML* was executed ten times, two fastest and two slowest times were discarded, and then the rest was averaged. This approach should mitigate the impact on accuracy caused by temporary processes suddenly run on the machine, and also dismisses the result of the first slower operation when the system caches have not yet been loaded. Although assessing performance with techniques not allowing the use of cache is also valuable, it would be misleading to combine the raw non-cached result with the remainder because the cache was not manually emptied. Using only six median values should ensure much more accurate results; nonetheless, there still are other factors which can influence the experiments – the testing machine was connected to the Internet and normal system processes were running during the benchmark execution. Given that the run time of the whole BlueBench suite was roughly a week on the testing machine, it is very probable that some of the tests were slightly influenced by system updates and so on.

6.4.1 Graph Loading

The database must be inevitably loaded for the rest of the operations to have data to work with, and that can simultaneously be used to monitor how fast the objects are inserted. The insertion speed was measured as Loaded objects per second (LOPS) so that results from datasets of different sizes can be directly compared.

Figure 6.2 depicts the LOPS value as it was measured in the *Labeled Graph Benchmark*. DEX was the fastest system, closely followed by Neo4j, both steadying at about 35.000 LOPS. InfiniteGraph and OrientDB placed on the other side of the spectrum, with performance an order of magnitude lower. This shows quite a difference in insertion speed of the assessed systems; however, the results also show that most of the GDBs scale regularly with growing size of the network – the decrease of LOPS is only sub-linear.

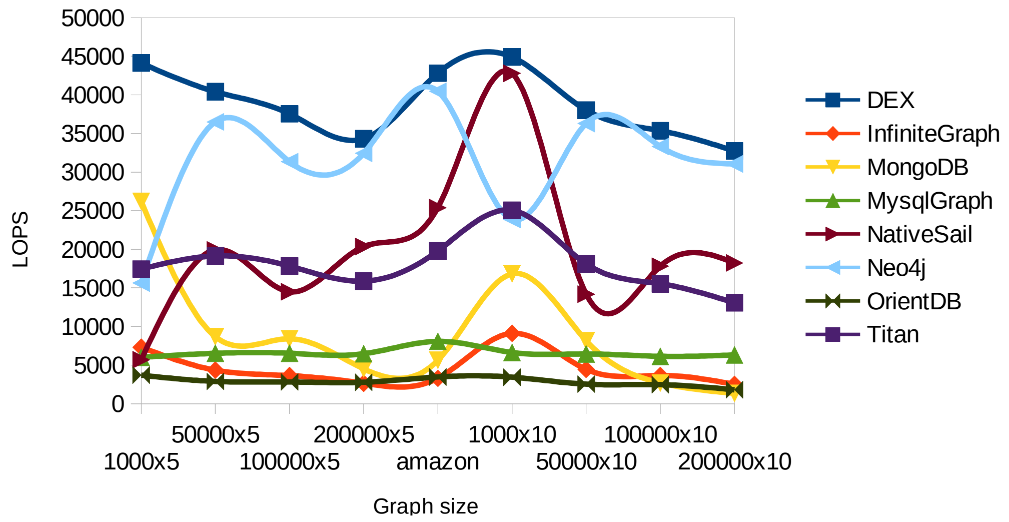


Figure 6.2: Objects loaded per second in *Labeled Graph Benchmark*.

The situation changes when element properties are involved in the second benchmark as displayed in Figure 6.3. The overall speed understandably diminishes by nearly half and Neo4j surpasses the rest of the systems by a big margin. There is no noticeable slowdown of Titan, and thus it gets on par with DEX. The rest of the GDBs remain much slower. Another change occurs when the graph elements are set to be indexed before the start of the loading (*Indexing Benchmark*). Specifically, Neo4j gets behind Titan and DEX which both do not seem influenced by the indexes at all. The reason of Titan's result is trivial, it does not support indexing over properties of edges; therefore the insertion is constrained with much less burden. By contrast, it will be seen in the forthcoming test results that DEX does not take advantage of the indexes at all – and they evidently are ignored during the insertion phase as well.

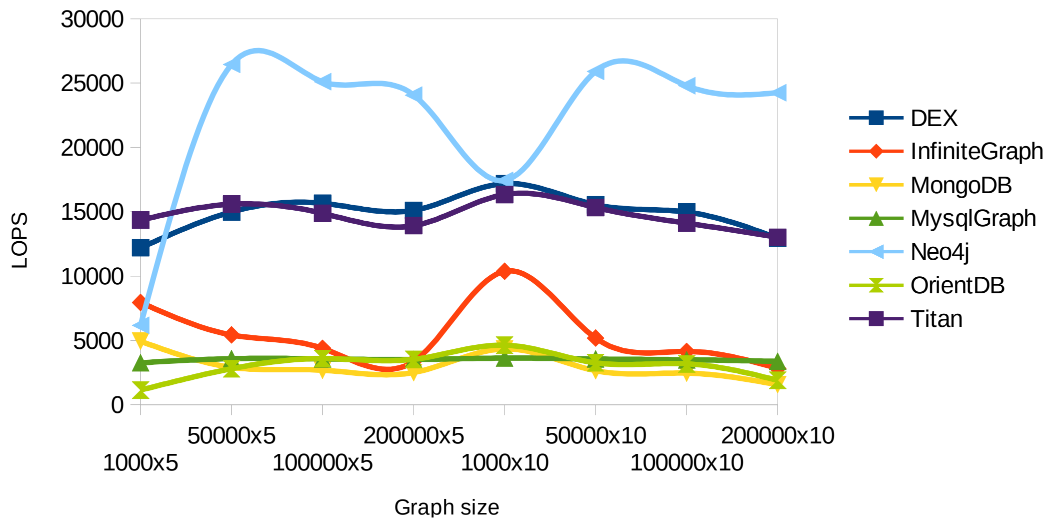


Figure 6.3: Objects loaded per second in *Property Graph Benchmark*.

6.4.2 Traversal

The performance of a breadth-first search traversal which follows the edge direction was monitored. To better estimate how the GDBs scale with growing data size, the results were expressed as the Traversed edges per second (TEPS). Figure 6.4 shows an absolute dominance of Neo4j on sparser graphs with TEPS getting as high as 300.000. It is followed by DEX, Titan and NativeSail. On the other hand, MongoDB and InfiniteGraph do not seem to be optimized for this kind of query at all, giving out TEPS of only around 1.000 which is further decreasing with the growing graph size.

The set of denser graphs brought Neo4j much closer to the rest of the systems, mainly to the benefit of DEX. As it can be seen in the plot, Neo4j's TEPS values have a decreasing tendency as the data get larger, whereas DEX's performance is gradually growing. We would probably see DEX surpassing Neo4j if the operation was tested on even bigger data. Meanwhile, Titan and NativeSail became roughly twice slower than DEX, but still clearly separated from the remaining GDBs.

The native implementation of the traversal algorithm provided by a subset of the tested engines was executed with the same parameters within the *Traversal-Native* operation. The performance is compared to the results of the conventional methods in Figure 6.5. Apart from Neo4j, the GDBs showed some improvement after using the native method; especially InfiniteGraph, TEPS values of which are up to four times higher. A healthy margin in favor of the native method is noticeable in DEX's results also. However, we still conclude that once an application uses Blueprints to work with the database, it is not worth making the effort to accommodate the native interface, because the differences in performance are not extensive.

The execution times of the *Traversal* operation when run in the three benchmarks were very similar; in other words, it was shown that the presence of element properties or custom indexes on these properties does not have any noteworthy effect on the efficiency of traversal queries.

A very similar situation occurred in the *FindNeighbors* operation; after all, the two operations are implemented using the same *getVertices()* method from

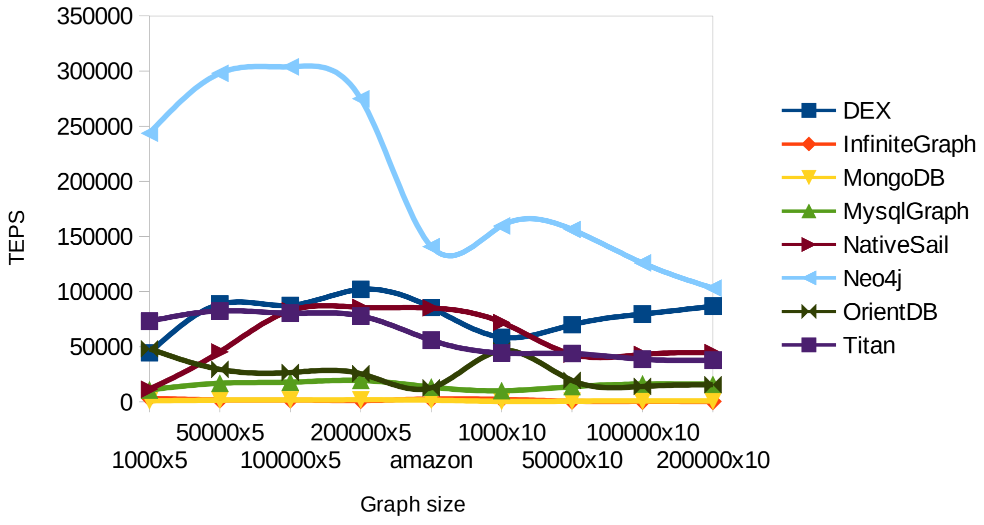


Figure 6.4: Traversed edges per second in *Labeled Graph Benchmark*.

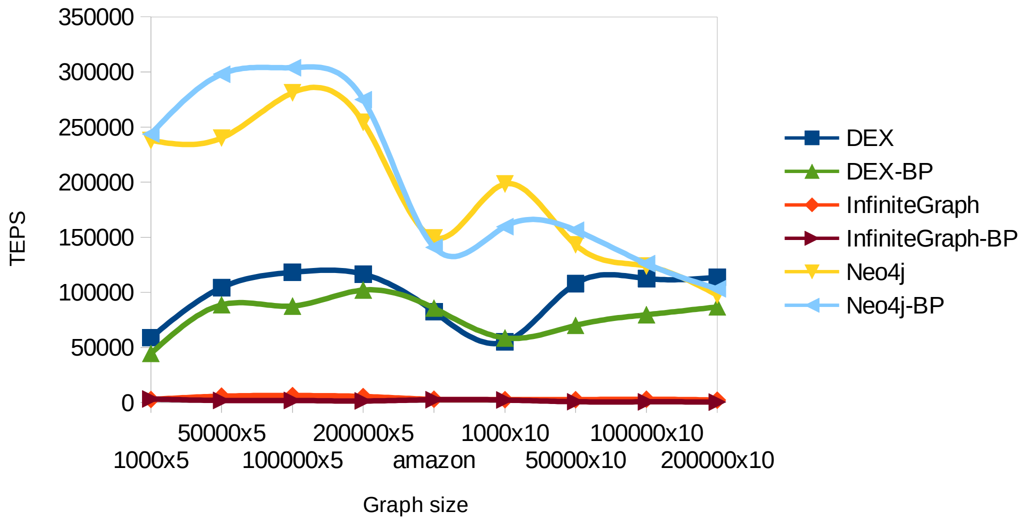


Figure 6.5: Comparison of native traversal implementations and their counterparts written in Blueprints (*Labeled Graph Benchmark*).

the Blueprints interface. Only NativeSail struggled and yielded surprisingly high results reaching hundreds of milliseconds for the largest graphs. This behavior was most probably caused by the *FindNeighbors* operation fetching all neighbors, even in the opposite direction, in contrast to *Traversal* operation where edge direction is respected. Besides that, the rest of the engines showed little or none growth of the response time with the increasing graph data size.

6.4.3 Shortest Path

The breadth-first search and the shortest path algorithms both count as traversals and so they could be expected to exhibit a similar performance diversity. However, *getVertices()* method was used for *Traversal* operation implementation, whereas *ShortestPath* employs *getEdges()* Blueprints API. The mechanism underlying these two methods can be separate; in addition, the different algorithms imply different query patterns, and thus dissimilar caching techniques can be used. Two versions of the shortest path operation were executed, the first without considering edge labels and the second following only edges having a label which was previously selected at random. We again cannot confirm any degradation of performance caused by properties or indexes attached to the elements; consequently, only results from the first benchmark are discussed.

Figure 6.6 depicts the results of *ShortestPathLabeled* operation. Their difference from those of *ShortestPath* is negligible, with the exception of Titan, which performed much better in the labeled version due to its vertex-centric indexes optimization (the index could be used to quickly obtain only the edges having the right label). In general, the results are distinct from those of *Traversal* operation; namely, DEX is not the nearly best performing system as it was with search traversals, as opposed to Titan which clearly improved. On the other hand, InfiniteGraph, MongoDB and MysqlGraph remained to be very inefficient, like they were in the *Traversal* operation.

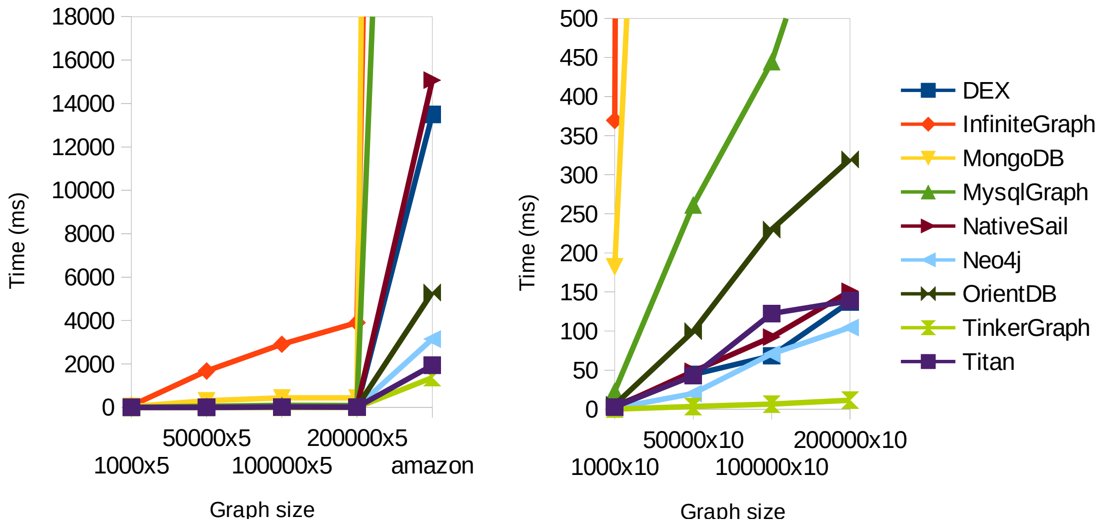


Figure 6.6: Run times of *ShortestPathLabeled* in *Labeled Graph Benchmark*.

The *ShortestPath* test results yield another interesting observation. As it is clearly noticeable from the plotted run times, the GDBs took much longer to execute the operation on the Amazon dataset than on the synthesized graph data, in spite of the graph size being of similar magnitude. This was caused by the fact that there mostly is not an existing oriented path between any two vertices randomly selected from a graph generated by the Barabasi-Albert model. Therefore, the executed algorithms usually finished prematurely and rather quickly. By contrast, the network obtained from Amazon is well connected, and thus the operations often ran until the path was completed, taking much more time to return. This suggests that the graphs created by the selected generator do not necessarily resemble real networks in every possible aspect.

As opposed to *ShortestPath*, *Dijkstra* operation represents a complex query requiring the traversal of the complete graph, and making use of both element properties and edge labels. Therefore, it is more complicated for the GDBs to take advantage of their caches as they would normally do during simpler traversals. This is likely to be the reason of MysqlGraph's acceptable performance in relation to the other systems and, equally importantly, in relation to its performance in *ShortestPath*.

Neo4j is the most efficient engine for this task; it even outperforms TinkerGraph by a recognizable margin (by almost 30%). Given that TinkerGraph works only in memory and can avoid any delays caused by persistent storages, an explanation for this could be that Neo4j managed to load the entire graph into its caches and calculated the algorithm there. The results of *Dijkstra* operation are plotted in Figure 6.7.

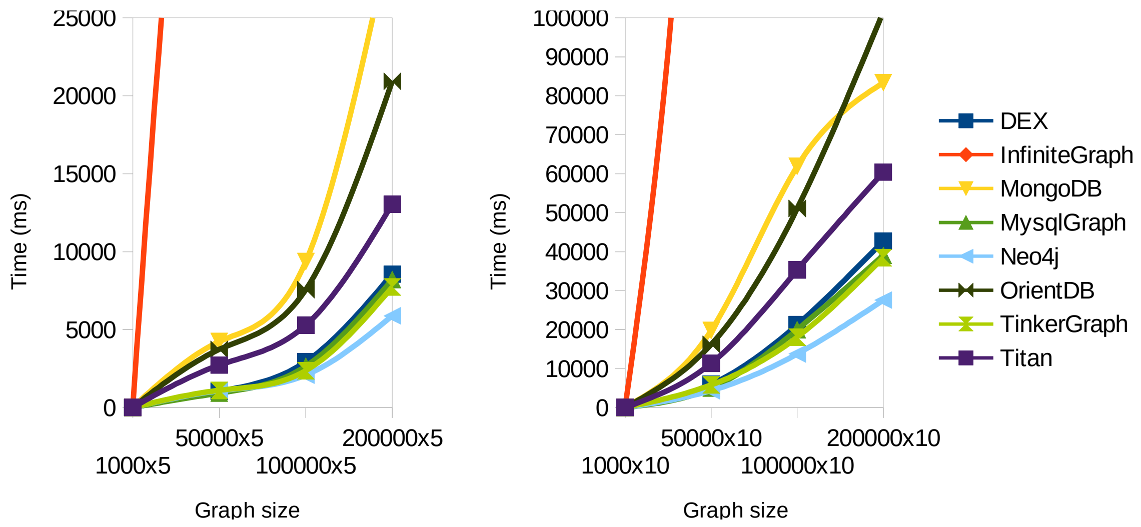


Figure 6.7: Performance of Dijkstra's algorithm in *Property Graph Benchmark*.

Only TinkerGraph's efficiency was hindered by the presence of element indexes; otherwise the GDBs were practically unaffected. Besides, with respect to the understandable sovereignty of TinkerGraph in all above operations, its performance in *Dijkstra* is surprising. Apparently, the algorithms and data structures in TinkerGraph engine are optimized for less demanding tasks. However, the Dijkstra's algorithm appeared not to be the best possible choice of a complex operation for GDB assessment. Considerable portions of the run time were spent inside the algorithm itself, leaving only limited room for the differences in the underlying systems to be fully revealed.

6.4.4 Non-traversing Queries

The performance of two operations which ignore relations between vertices was analyzed; namely, *FindEdgesByProperty* and *FindVerticesByProperty*. Both the operations filter the graph's elements according to a string randomly chosen from a set of strings which are stored in the elements' attribute. Therefore, the operation requires the database systems to iterate through all the objects in the graph and perform string comparisons on the specified attribute of the objects. In *Indexed Graph Benchmark* the GDBs are encouraged to use their indexing mechanisms.

Analysis of *FindEdgesByProperty* operation's execution times and the comparison between plain and indexed benchmark versions is depicted in Figure 6.8. Once the indexes are used, an apparent performance improvement can be seen for almost all the systems except for Titan, which does not support indexing edges, MysqlGraph, where indexes are used permanently, and DEX – which is the only surprising case. The experiments clearly show that DEX completely ignores the assigned index. This problem, in addition to DEX's already slow one-by-one filtering, renders the database system very impractical for this type of queries. In fact, the indexing problem can be caused only by the implementation of the Blueprints interface and not by the DEX engine itself, accuracy of which we could not verify.

The most efficient persistent GDB for this operation is Neo4j, being at least twice as fast as the other systems. Such a big difference is astonishing since this purely non-traversing query should definitely be handled at least equally efficiently by MongoDB, OrientDB, InfiniteGraph and Titan, which have their backends based on standalone NoSQL databases (as described in Chapter 2). MysqlGraph's rapid growth of response time can be explained by the need of an execution of a *join* to merge the tables where the edges and their properties are stored.

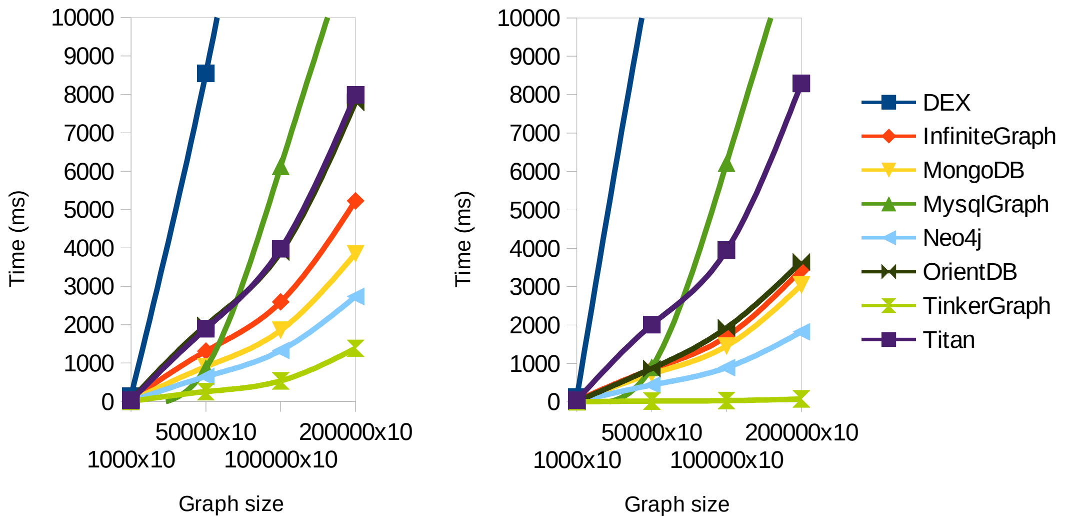


Figure 6.8: *FindEdgesByProperty* in *Property Graph Benchmark* (left) and *Indexed Graph Benchmark* (right).

FindVerticesByProperty operation provided similar results in most cases. However, MysqlGraph does not need to *join* tables to return vertices; thus, it is slightly faster than most of the other engines in the *Indexed Graph Benchmark*. Moreover, Titan could use the index this time which made its performance considerably better. Finally, Neo4j again excelled amongst the persistent GDBs, being about twice faster than the second best performing engine.

6.4.5 Manipulation Queries

Extensive data manipulation queries were executed on the databases through *UpdateProperties* and *RemoveVertices* operations in order to observe how the systems cope with situations when data need to be changed, or deleted respectively.

Performance of modifying properties in the second and third benchmark is compared in Figure 6.9. The presence of indexes evidently has very negative influence on the efficiency of updating data in all the systems, especially in InfiniteGraph, MongoDB, Neo4j and also slightly on OrientDB. The other systems, however, did not have to consider indexes for various reason discussed above. In conclusion, Neo4j showed very good performance in updating data for being more than three times faster than the second best performing system; and it also handled indexes quite comfortably. OrientDB and InfiniteGraph ended up on the other end of the scale because the former was more than an order of magnitude slower when the data was indexed and the latter was very slow throughout the entire test.

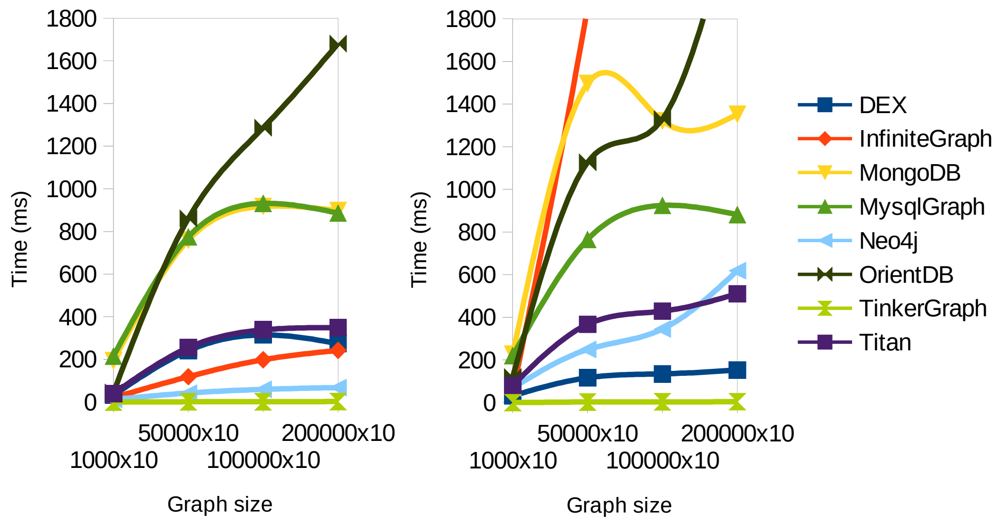


Figure 6.9: *UpdateProperties* in *Property Graph Benchmark* (left) and *Indexed Graph Benchmark* (right).

Deleting a vertex is a rather expensive operation because it requires any edges incident with the vertex to be deleted also. Since *RemoveVertices* was always removing a significant part of the graph in one go, the execution times reached tens of seconds for the slower systems. Therefore, as it can be seen in Figure 6.10, the density of the graph substantially influenced the performance. Neo4j, DEX and Titan could cope with the situation much more efficiently than the other GDBs. When the same operation was run in *Indexed Graph Benchmark*, all engines apart from MongoDB and TinkerGraph ended up suspiciously unaffected.

This observation suggests that the indexes must have been updated in another thread after the method had returned, as opposed to *UpdateProperties* operation effect, where the deceleration was clearly noticeable.

RemoveVertices was also executed in the first benchmark, where InfiniteGraph and MysqlGraph did not exhibit such big problems with graph density as they did in *Propery Graph Benchmark*; therefore, it is apparent that the process of deleting too many edge attributes was the true cause of their slowdown. Finally, NativeSail is part of the *Labeled Graph Benchmark* and was tested on this operation, too; only to expose a surprisingly inefficient implementation of the vertex removal method. To sum up, Neo4j was the fastest engine for deleting vertices from the sparser version of the graph, while DEX performed slightly better on the denser graph.

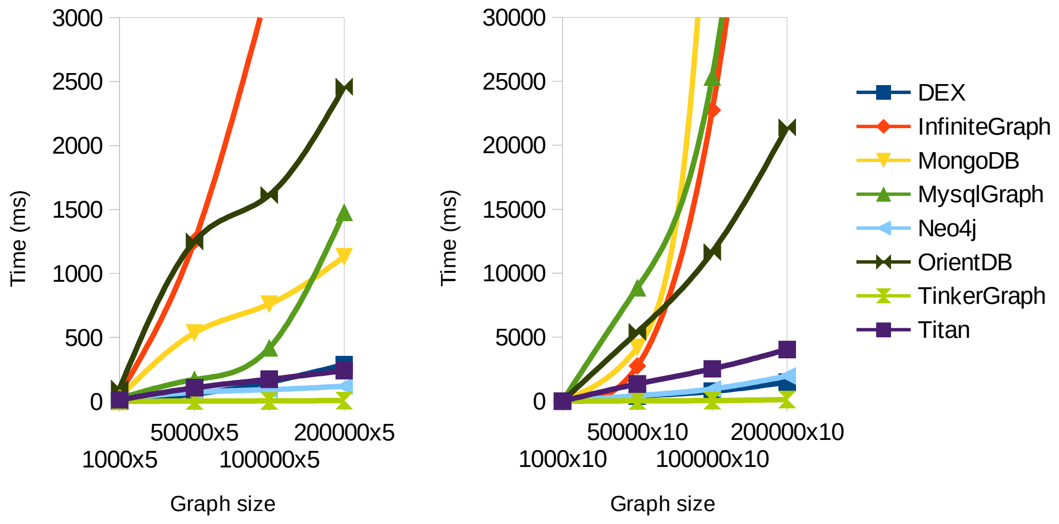


Figure 6.10: Run times of *RemoveVertices* in *Property Graph Benchmark*.

6.4.6 Summary of the Results

The results of a number of tests incorporated in BlueBench are depicted and discussed within this section. BlueBench is divided into three particular benchmarks which differ in the amount of data stored in the database, but are identical in the structure of the graph data and very similar in what operations get executed in them. Therefore, it is possible to compare the efficiency of the operations on the set of the selected GDBs, while conducting the same comparison across benchmarks also. This leaves the evidence of how well the databases scale with growing size of the network and how much they are influenced by the presence of attributes or indexes in the graph.

We divided the operations into several groups to make the analysis more comprehensible: graph loading, traversal queries, non-traversal queries and data manipulation. It is surprising to observe that the groups of experiments provided results that are quite alike; in other words, the GDBs' relative performance was similar although it was measured in entirely different scenarios.

Arguably the strongest reason for GDBs to exist is the need of efficient implementation of traversal operations on persistent graph data. In this area Neo4j and DEX clearly outperform the rest of the systems, mainly because of the specialization of their backends for exactly this type of queries. Neo4j was constantly achieving the best performance even in the other tests, followed by DEX and Titan. These results are in contrast with [4] which was written four years ago and favored DEX over Neo4j. Clearly, Neo4j has made a lot of progress over the years.

On the other hand, it was shown that directly using a document-oriented database or even relational database for graph operations is not very efficient. MysqlGraph and especially MongoDB performed well only in tests which were either lightly or not graph related at all. Finally, InfiniteGraph was revealed to be the least performing implementation with notoriously slow traversals; often struggling with scenarios where the final execution time was barely acceptable. However, it must be stated that Infinitegraph is focused on distributed solutions with horizontal scaling, not primarily addressing performance on a single machine.

The experiments also helped to discover an important difference between the synthesized data and the data obtained from a real network. The graphs generated by the Barabasi-Albert model exhibited low oriented interconnection of vertices and higher probability of nodes with an extensive degree; although the latter feature could be altered by setting an appropriate parameter before running the graph generation.

6.5 Summary

The process of executing BlueBench was depicted and its results were interpreted in this chapter. We began with describing the machine which was used for the benchmark runs and explaining what datasets were used to fill the databases with data. We then proceeded to list the GDBs that were chosen for the experiments, including the most important specifics that had to be respected during the implementation of BlueBench. In addition, the design of our own representative of the relational databases world, MysqlGraph, was portrayed. Finally, the results gathered from the execution of BlueBench were plotted and analyzed.

Conclusion

This thesis addresses the need of creating a complex and fair benchmark for an experimental assessment of various GDB implementations. We have elaborated on the specific requirements of such a benchmark, and analyzed all possible ways of accessing the databases and conducting the experiments, so that the benchmark can cover all functionality expected from the GDBs and yet remain fair. We have also examined the existing work in the field of graph databases benchmarking; and found that some valuable results have been produced, yet the testing scenarios were always limited in terms of the variety of assessed systems, input data or tested operations.

The result of our efforts, BlueBench, can be run both on real and generated graph datasets and is composed of a number of testing scenarios which reflect the wide range of use-cases that GDBs have to face today. BlueBench is based on the already existing GraphDB-Bench tool for its great generality and extensibility; such an approach helped to guarantee the further extensibility of BlueBench itself.

We executed BlueBench on datasets of both varying sizes and graph models, and evaluated an extensive set of particular database systems of different types, which includes native GDB engines, an RDF store, a document-oriented database and a relational database. Therefore, not only the performance of the best known GDBs was compared, also the appropriateness of putting non-graph databases into graph-related tasks was verified. However, it was considerably challenging to include such different databases in the benchmarks as many peculiar implementation specifics had to respected.

The final results of BlueBench are engrossing; they confirm the assumption that GDBs are much more suitable for graph-related queries than any other database systems. Among the GDBs themselves, we have managed to separate the really well performing systems from those that only claim to be the most efficient.

Future Work

Although lots of effort was put into selecting the set of aspects to be tested by BlueBench, there is much more work still to be done. For example, the performance of the database systems was measured only on a single machine under convenient conditions, i.e. the systems had as much memory as they needed. Many interesting results could be obtained from executing the benchmarks on a cluster of nodes where the GDBs would be replicated and a concurrent access (with intentional conflicts) performed.

Moreover, due to the complexity of some testing operations included in BlueBench, all the input graph datasets had to be rather small (no more than a few millions of objects). It would be interesting to observe how the databases would cope with data of much higher volumes if, of course, only the less demanding operations were executed. The fact is that the GDB vendors claim that their products can scale up to *billions* of objects in the graph.

Finally, a great success would be reviving the original, although never really started, GraphDB-Bench project which would not be used only be individuals, but by entire open communities with the intention of providing a universal graph database benchmarking tool, which would periodically evaluate the performance of all relevant graph database software. We hope that such a feat can still be achieved.

Bibliography

[1] Rodriguez, Neubauer. *Constructions from dots and lines.* Bulletin of the American Society for Information Science and Technology, 2010, Pages 35–41.

[2] Dominguez-Sal, Muntés-Mulero, Martínez-Bazán, Larriba-Pey. *Graph Representation.* Graph Data Management: Techniques and Applications, 2012, Pages 1–28. ISBN 978-1-61350-053-8.

[3] Dominguez-Sal, Urbón-Bayes, Giménez-Vañó, Gómez-Villamor, Martínez-Bazán, Larriba-Pey. *Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark.* Springer Berlin Heidelberg, 2010, Pages 37–48. ISBN 978-3-642-16720-1.

[4] Bader, Feo, Gilbert, Kepner, Koester, Loh, Madduri, Mann, Meuse, Robinson. *HPC Scalable Graph Analysis Benchmark.* 2009. (`http://www.graphanalysis.org/benchmark/index.html`)

[5] Chakrabarti, Zhan, Faloutsos. *R-MAT: A Recursive Model for Graph Mining.* 2004. (`http://repository.cmu.edu/compsci/541/`)

[6] Ciglan, Averbuch, Hluchy. *Benchmarking traversal operations over graph databases.* 2012. (`http://ups.savba.sk/~marek/papers/gdm12-ciglan.pdf`)

[7] Lancichinetti, Fortunato. *Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities.* Phys. Rev. E 80, 016118, 2009. (`https://sites.google.com/site/andrealancichinetti/benchmark2.pdf?attredirects=0`)

[8] Vicknair, Macias, Zhao, Nan, Chen, Wilkins. *A comparison of a graph database and a relational database: a data provenance perspective.* 2010, ACM SE '10 Proceedings of the 48th Annual Southeast Regional Conference, Article No. 42. ISBN: 978-1-4503-0064-3.

[9] Leskovec, Adamic L., Adamic B.. *The Dynamics of Viral Marketing.* 2007, ACM Transactions on the Web (ACM TWEB)

[10] Averbuch Alex. *Say hi to GraphDB-Bench.* 2010. (`http://alexaverbuch.blogspot.cz/2010/10/say-hi-to-graphdb-bench.html`)

[11] Oracle. *What is MySQL?* (`http://dev.mysql.com/doc/refman/5.0/en/what-is-mysql.html`)

[12] Aurelius Team. *Using Titan with BerkeleyDB.* (`https://github.com/thinkaurelius/titan/wiki/Using-BerkeleyDB`)

[13] Aurelius Team. *Titan - Transaction Handling.* (`https://github.com/thinkaurelius/titan/wiki/Transaction-Handling`)

[14] Aurelius Team. *Titan Limitations.* (https://github.com/thinkaurelius/titan/wiki/Titan-Limitations)

[15] Ciglan. *SGDB3 – Simple Graph Database.* (http://ups.savba.sk/~marek/sgdb.html)

[16] TinkerPop. *Blueprints – Graph Indices.* (https://github.com/tinkerpop/blueprints/wiki/Graph-Indices)

[17] sparsity technologies. *Why DEX.* (http://www.sparsity-technologies.com/dex)

[18] DB-Engines. *Ranking of Graph DBMS.* (http://db-engines.com/en/ranking/graph+dbms)

[19] DB-Engines. *Ranking of Document Stores.* (http://db-engines.com/en/ranking/document+store)

[20] sparsity technologies. *DEX - A High-Performance Graph Database Management System.* (http://www.sparsity-technologies.com/dex)

[21] sparsity technologies. *DEXHA - Architecture.* (http://www.sparsity-technologies.com/dex_tutorials4?name=Architecture)

[22] Angles Renzo. *Say hi to GraphDB-Bench.* 2012. (http://dcc.utalca.cl/~rangles/files/gdm2012.pdf)

[23] Robinson, Webber, Eifrem. *Graph Databases.* 2013, O'Reilly Media. (http://graphdatabases.com/)

[24] Objectivity inc.. *Understanding Accelerated Ingest.* (http://wiki.infinitegraph.com/3.0/w/index.php?title=Understanding_Accelerated_Ingest)

[25] Objectivity inc.. *InfiniteGraph 3.0 Technical Specifications.* (http://www.objectivity.com/products/infinitegraph/technical-specifications/)

[26] Neo Technology. *Neo4j HA Architecture.* (http://docs.neo4j.org/chunked/stable/ha-architecture.html)

[27] Redmond, Wilson. *Seven Databases in Seven Weeks.* 2012, O'Reilly Media. ISBN: 978-1-934356-92-0.(http://it-ebooks.info/book/866/)

[28] NuvolaBase Ltd. *OrientDB.* (https://github.com/nuvolabase/orientdb)

[29] NuvolaBase Ltd. *OrientDB - Caching.* (https://github.com/nuvolabase/orientdb/wiki/Caching)

[30] NuvolaBase Ltd. *OrientDB - Concepts.* (https://github.com/nuvolabase/orientdb/wiki/Concepts)

[31] NuvolaBase Ltd. *OrientDB - Concepts - Record Version.* (`https://github.com/nuvolabase/orientdb/wiki/Concepts#record-version`)

[32] NuvolaBase Ltd. *OrientDB - Transactions.* (`https://github.com/nuvolabase/orientdb/wiki/Transactions`)

[33] Aurelius. *Titan.* (`https://github.com/thinkaurelius/titan/wiki`)

[34] Aurelius. *Titan - Indexing Backend Overview.* (`https://github.com/thinkaurelius/titan/wiki/Indexing-Backend-Overview`)

[35] Aurelius. *Titan - Storage Backend Overview.* (`https://github.com/thinkaurelius/titan/wiki/Storage-Backend-Overview`)

[36] Broecheler Matthias. *Big Graph Data.* (`http://www.slideshare.net/knowfrominfo/big-graph-data`)

[37] Rodriguez Marko. *The Rise of Big Graph Data.* (`http://www.slideshare.net/slidarko/titan-the-rise-of-big-graph-data`)

[38] Rodriguez Marko. *The Rise of Big Graph Data.* (`http://www.slideshare.net/slidarko/titan-the-rise-of-big-graph-data`)

[39] Rodriguez Marko. *The Rise of Big Graph Data.* (`http://www.slideshare.net/slidarko/titan-the-rise-of-big-graph-data`)

[40] Erdos, Renyi. *On random graphs.* Mathematicae 6, 1959, Pages 290–297.

[41] Leskovec, Lang, Dasgupta, Mahoney. *Statistical properties of community structure in large social and information networks.* 2008.

[42] Barabási, Albert. *Emergence of scaling in random networks.* Science. 2008. Pages 509–512.

A. Contents of the CD

The enclosed files are organized as follows:

/kolomicenko.pdf
  This thesis.

/results/
  The genuine BlueBench results in a .csv format.

/charts/
  The charts generated from the results.

/BlueBench/doc/api/
  A very brief reference documentation to the source files.

/BlueBench/doc/install.txt
  Instructions of how to install and execute BlueBench.

/BlueBench/src/java
  Java sources.

/BlueBench/src/python
  Python sources for generating the artificial graph datasets.

/BlueBench/lib
  All necessary libraries.

/BlueBench/data/datasets
  A directory for generated graphs.

/BlueBench/data/results
  A directory for BlueBench results.

/BlueBench/data/DB
  A directory for the tested databases' persistent storages.

/BlueBench/run.sh
  A script for building and running BlueBench.