Karlsruhe Institute of Technology

Institut für Telematik
Pervasive Computing Systems / TECO
Leiter: Prof. Dr.-Ing. Michael Beigl

Benchmarking of Graph Databases - Suitability for the Industrial Internet of Things

Bachelorarbeit
von

Christian Navolskyi

an der Fakultät für Informatik

Verantwortlicher Betreuer:	Prof. Dr. Michael Beigl
Betreuender Mitarbeiter:	Andrei Miclaus, M.Sc.
Bearbeitungszeit:	15.01.2018 – 14.05.2018

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und weiterhin die Richtlinien des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, den 14.05.2018

Zusammenfassung

In dem derzeitigen Wandel der Industrie in Richtung Industrie 4.0 kommt es auch dazu, dass viele Daten produziert werden, welche wertvoll sind, da sie für Verbesserungen in der Produktion verwendet werden können. Die anfallenden Daten können als Graph repräsentiert werden, deswegen macht es Sinn Graph-Datenbanken zu untersuchen.
In dieser Arbeit werden wir die Performanz von Graph-Datenbanken in einem industriellen Umfeld untersuchen. Unser Ziel ist es, herauszufinden ob die Resultate aus bestehender Forschung in diesem Gebiet herangezogen werden können um die Tauglichkeit der Graph-Datenbanken in der Industrie zu prüfen. Anstatt Graphen ähnlich zu denen in sozialen Netzen für die Untersuchungen zu nutzen, wie es die meisten Studien in diesem Gebiet tun, werden wir eine Daten-Struktur entwerfen die den industriellen Datenraum repräsentiert und schauen uns ebenso die Besonderheiten eines industriellen Einsatzes einer Graph-Datenbank an, um unsere Arbeitslasten entsprechend zu entwerfen.

Der Yahoo! Cloud Service Benchmark (YCSB) wird erweitert um Datensätze mit der von uns entworfenen Datenstruktur zu generieren. Für die Auswertung haben wir folgende vier allgemein bekannte Graph-Datenbanken ausgewählt, Apache Jena, Neo4j, OrientDB und Sparksee (früher bekannt unter dem Namen DEX). Deren Java APIs wurden genutzt um sie in YCSB einzubinden.

Wir haben die Datenbanken mit unserer Datenstruktur auf einem einzelnen Rechner mit einem i7-3770K Prozessor und 16GB RAM ausgeführt und kamen zu dem Fazit, dass aktuelle Graph-Datenbanken nicht für den industriellen Einsatz geeignet sind. Sparksee konnte nicht mit dem Datensatz in voller Größe getestet werden, da die kostenlose Lizenz diese Datenmenge nicht unterstützt. Wenn es den erreichten Durchsatz halten könnte, würde es auch mit dem Datenaufkommen aus der Industrie zurechtkommen. Da wir das nicht direkt testen konnten, können wir keine fundierte Entscheidung diesbezüglich treffen. OrientDB verfehlte unser gesetztes Ziel für den Durchsatz nur knapp, wohingegen Jena und Neo4j weit davon entfernt waren.
Nachdem wir die Generalisierbarkeit von Resultaten aus Graph-Datenbank Benchmarks ausgewertet haben, können wir auch hier keine eindeutige Entscheidung treffen, da Vergleiche mit unterschiedlichen Studien zu verschiedenen Schlussfolgerungen führen. Wir konnten allerdings feststellen, dass die Performanz beim hinzufügen von Knoten und Kanten unter anderem auch vom Lese-Durchsatz abhängt, da diese Operation gebraucht wird, um Kanten zum Graphen hinzuzufügen. Schließlich scheinen doch mehr Argumente dafür zu sprechen, dass Graph-Datenbanken schlechter im industriellen Einsatz abschneiden. Das führt dazu, dass die Resultate aus anderen Studien nicht direkt übernommen werden können.

Abstract

In the current transition happening in the industry towards Industry 4.0, a lot of data is produced. This data is valuable as it can be used for all kinds of improvements in the production process. The accumulating data can be represented in a graph and therefore it is worth examining graph databases.
In this thesis, we will investigate the performance of graph databases in an industrial environment. Our goal is to examine whether the results of other research in the field of graph databases can be used to determine the performance of a graph database in the industrial internet of things. Instead of using social network graphs as other research in this field does, we will design a data structure that represents the industrial data space and also look at the peculiarities of an industrial use to design our workloads accordingly.

The YCSB benchmark will be extended to create datasets with our desired data structure. For the benchmark, we chose the following four commonly known graph databases, Apache Jena, Neo4j, OrientDB and Sparksee (also known as former DEX). Their Java APIs were used to integrate them into YCSB.

We evaluated the databases with our data structure on a single machine with an i7-3770K processor and 16GB of RAM and came to the conclusion, that graph databases are not suitable for an industrial application. Sparksee could not be tested with the large dataset, due to a missing license. If it could hold its throughput it would be suitable, since we could not investigate that, no solid conclusion can be drawn. OrientDB missed the target throughput slightly, whereas Jena and Neo4j were far away from the target throughput.
After evaluating the generalisability of graph database benchmark results we came to no clear conclusion, as the comparison with other research points into two different directions. However, we can say that the insert throughput also depends on the read performance of the database as inserting edges requires read operations. Besides that, there were more arguments indicating that graph databases perform worse in an industrial environment. This leads to the conclusion, that the results of other studies cannot directly be used to determine the performance in an industrial use case.

Contents

1. Introduction

1.1 Problem Statement

With the growing digitalisation of the industry, more data is available and can be used to improve production processes. The amount of data created depends on the individual use case but still, it needs to be stored to be useful. Since there are multiple databases available it can be difficult to choose the right one for an individual scenario.

Current graph database benchmarks only cover social network graphs, which differ from the data structure present in the industry, for example, in their edge to node ratio.

1.1.1 Use Case - Industry 4.0

There are multiple analytic algorithms to run on data to extract certain features. In the industry, those algorithms play an important role too but in this thesis we are looking at different aspects of the industrial use case, mainly inserting and reading data. As far as we know an industrial data structure wasn't used to examine the performance of graph databases before.

In section 2.2 we will show an example given by the industry. There is no industrial data available publicly so we have to base our design on that given example which is visualised in figure 2.2.

1.1.1.1 Inserting Data

To digitalise the production processes the data produced by every machine in the production line should be stored for future analysis. And to store that data it needs to be written into a database. Since most factories are running 24 hours a day, the machines are producing a lot of data. Storing the data from production will be the base load for the underlying databases.

1.1.1.2 Reading Data

Besides using the stored data for analysis algorithms, simply reading data from the database is another common task. An example would be to get the time at which a specific product was processed by a specific machine to check if all parameters were set correctly.

1.2 Question

The question of our research is, how well current graph databases are capable of handling the amount of data created in an industrial environment. Furthermore, we will look at how the structure effects performance to conclude whether other research investigating the performance of graph databases can be used to determine the performance these databases would have in an industrial environment.

1.3 Methodology

We will choose the databases to use for our testing from other studies covering benchmarking graph databases to be able to compare the results and look at similarities in behaviour. To evaluate different databases we first will look up existing benchmarks and choose the best one for our research. In the benchmarking program we need to look at the creation of data and how it can be stored and retrieved. The same exact dataset should be used for all databases equally to eliminate the variation that comes with generating data during each benchmark run. Workloads will be designed to investigate the performance of graph databases with industrial data and the production environment will be simulated. With the databases and benchmark set up, we will run the workloads and evaluate the results to conclude whether current databases are suitable for the industrial internet of things.

1.4 Goal of this Thesis

With this thesis, we want to examine whether and if so, how well graph databases are able to stand the load of a production line. Because every manufacturer is different and we cannot cover all scenarios we try to cover the most important parameters so that the suitability for the individual case can be estimated. Besides this specific investigation we will try to conclude whether the results of research performed on graph databases with social network graphs can be applied to the industrial use case.

1.5 Structure

In chapter 2 we are motivating graph and the use of graph databases. The different kinds of graph databases are explained and an example database which we are testing is mentioned and shortly described. Also in this chapter we are comparing the different available benchmarking programs and their features and take a look at research done by others in the field of graph database benchmarking.

In chapter 3 the industrial data is modelled, and its structure is analysed as well as a reasonable amount of data is determined. Then we are figuring out how a workload

could look like in an industrial environment. At last, we further analyse our chosen benchmarking program and give an overview of its procedure.

Chapter 4 is focused on the design of the different extensions for the benchmark and also the concrete data structure. For the extensions we cover the design of the specific workloads, the design of classes to create and recreate the dataset, the graph workload class managing the graph databases and the graph data and finally the database bindings which are responsible for connecting the database to the benchmarking program.

In chapter 5 the implementation of the single components is described. First we cover the graph data generator which includes the class for creating the graph data as well as the class for recreating it from files. Next, the bindings are implemented and their individual adaptions to the benchmark are highlighted. And lastly, we explain the graph workload class that is the mediator between the created graph data and the database bindings.

Chapter 6 focuses on running the benchmark and evaluating the results. First, we define our objective during evaluation. Then the configuration of our system is stated, as well as the hardware and the software side. Next the procedure of running the benchmarks sequentially is explained following the different aspects we are testing. These are grouped into ”throughput” in section 6.4, ”production simulation” in section 6.5 and ”retrieving under load” in section 6.6. Each group includes multiple benchmarks in which we changed one variable at a time. The results are presented directly after each benchmark followed by a discussion to interpret the results.

In chapter 7 we draw a conclusion over our work and give the answer to our research question from above. Also ideas for future research and development in this field are presented. Finally a summary is given at the end of this chapter.

2. Background & Related Work

In this chapter, we will give an introduction to the different fields touched by our research. Related work is mentioned primarily in section 2.4 as it covers the different benchmarks and their findings.

2.1 Graphs

A graph as the literature tells us [1, p. 89] is a tuple $G = (V, E)$ where $E \subseteq V \times V$. Elements of V are called vertices and elements of E are called edges. The set of vertices has to be non-empty, but the edge set can be. In this thesis we are focusing on directed graphs only, although some graph databases are capable of handling undirected graphs too. Also there would be no benefit in using undirected edges since our model also uses directed edges. In general graphs can have labels or weights on their edges as stated in [1, p. 99]. For our purposes we will use labels on the vertices and edges to ease the understanding of our data structure. In section 2.3 we will give reasons why having labels on the graph components is useful.

Figure 2.1 shows an example of a directed graph with labels on its vertices. An equivalent representation of that graph would be

$$\begin{aligned} V &= \{1, 2, 3\} \\ E &= \{(1, 2), (1, 3), (2, 1), (3, 2)\}. \end{aligned} \tag{2.1}$$

2.2 Industrial Data

Under the term ”industrial data” we understand data which is produced by machines during the production. That could be the current settings of the machine, temperatures or tolerances measured during processing or what product is currently worked on. In chapter 3.1 the possible structure of this data is analysed.

As there is no publicly available information about how industrial data should look like we will use the example given by our partners at SICK AG [2] as an inspiration for our test data.

Listing 2.1 shows the graph excerpt of our given example.

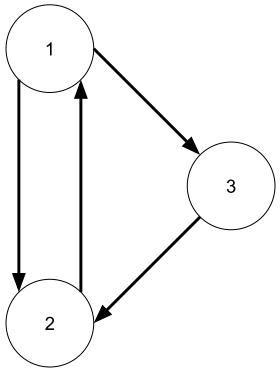

Figure 2.1: A directed graph with three labelled vertices and four edges.


```
"@graph": [
  {
    "@id": "http://localhost:3000/observations/185",
    "@type": "ssn:Observation",
    "featureOfInterest": "aoi:Feature",
    "observationSamplingTime": "2016-05-18T12:55:27.954Z",
    "observedProperty": [
      "aoi:twisting",
      "aoi:y-shift",
      "aoi:x-shift"
    ],
    "observationResult": "http://localhost:3000/observations/185/sensor-output",
    "observationResultTime": "2016-05-18T12:55:27.954Z",
    "observedBy": "http://localhost:3001/AOI_SMD407",
    "dataClass": "Testdata"
  },
  {
    "@id": "http://localhost:3000/observations/185/sensor-output",
    "@type": "ssn:SensorOutput",
    "isProducedBy": "http://localhost:3001/equipment/AOI_SMD407",
    "hasValue": "http://localhost:3000/observations/185/result"
  },
  {
    "@id": "http://localhost:3000/observations/185/result",
    "@type": "ssn:ObservationValue, shopfloor:Panel",
    "orderNo":"http://localhost:3000/order#0",
    "partNr": "http://localhost:3000/part#2060817",
    "hasPart": "http://localhost:3000/observations/185/board#3827581",
    "startTime": "2016-05-18T12:55:27.954Z",
    "endTime": "2016-05-18T12:56:27.954Z"
  },
  {
    "@id": "http://localhost:3000/observations/185/board#3827581",
    "@type": "shopfloor:Board",
    "hasPart": ["http://localhost:3000/observations/185/component#C1-1","http://
        localhost:3000/observations/185/component#C2-1"],
    "boardUID":"3827581",
    "isBadBoard": false
  },
  {
    "@id": "http://localhost:3000/observations/185/component#C1-1",
    "@type": "shopfloor:Component",
    "componentType": "C0603",
    "position":0,
    "testFeature":[
      {
        "@id": "http://localhost:3000/observations/185/component#C0603-MENI-901-
            TWISTING",
        "feature": "aoi:twisting1",
        "analysisMode": [
          {"@id": "http://localhost:3000/observations/185/AnalysisMode#C0603-MENI
              -901-TWISTING",
            "windowNumber": "901",
```

```
            "featureFlag": "0",
            "mode":"MENI"
          }
        ],
        "hasValue": {
          "@type": "xsd:integer",
          "@value": "10"
        }

      },
      {
        "@id": "http://localhost:3000/observations/185/component#C0603-MENI-901-Y-
            Shift",
        "feature": "aoi:y-shift1",
        "analysisMode": [
          {"@id": "http://localhost:3000/observations/185/AnalysisMode#C0603-MENI
              -901-Y-Shift",
            "windowNumber": "901",
            "featureFlag": "0",
            "mode":"MENI"
          }
        ],
        "hasValue": {
          "@type": "xsd:integer",
          "@value": "-17"
        }
      },
      {
        "@id": "http://localhost:3000/observations/185/component#C0603-MENI-901-X-
            Shift",
        "feature": "aoi:x-shift1",
        "analysisMode": [
          {"@id": "http://localhost:3000/observations/185/AnalysisMode#C0603-MENI
              -901-X-Shift",
            "windowNumber": "901",
            "featureFlag": "0",
            "mode":"MENI"
          }
        ],
        "hasValue": {
          "@type": "xsd:integer",
          "@value": "20"
        }
      }
    ]
  },
  {
    "@id": "http://localhost:3000/observations/185/component#C2-1",
    "@type": "aoi:Component",
    "componentType": "C0603",
    "position":0,
    "testFeature":[
      {
        "@id": "http://localhost:3000/observations/185/component#C0603-MENI-901-
            TWISTING",
        "feature": "aoi:twisting1",
        "analysisMode": [
          {"@id": "http://localhost:3000/observations/185/AnalysisMode#C0603-MENI
              -901-TWISTING",
            "windowNumber": "901",
            "featureFlag": "0",
            "mode":"MENI"
          }
        ],
        "hasValue": {
          "@type": "xsd:integer",
          "@value": "12"
        }
      },
      {
        "@id": "http://localhost:3000/observations/185/component#C0603-MENI-901-Y-
            Shift",
        "feature": "aoi:y-shift1",
        "analysisMode": [
```

```
          {"@id": "http://localhost:3000/observations/185/AnalysisMode#C0603-MENI
              -901-Y-Shift",
            "windowNumber": "901",
            "featureFlag": "0",
            "mode":"MENI"
          }
        ],
        "hasValue": {
          "@type": "xsd:integer",
          "@value": "14"
        }
      },
      {
        "@id": "http://localhost:3000/observations/185/component#C0603-MENI-901-X-
            Shift",
        "feature": "aoi:x-shift1",
        "analysisMode": [
          {"@id": "http://localhost:3000/observations/185/AnalysisMode#C0603-MENI
              -901-X-Shift",
            "windowNumber": "901",
            "featureFlag": "0",
            "mode":"MENI"
          }
        ],
        "hasValue": {
          "@type": "xsd:integer",
          "@value": "11"
        }
      }
    ]
  }
]
```

Listing 2.1: An excerpt showing the observation of components.

In figure 2.2 the provided example is visualised partially. It shows the observation of a product.

2.3 Graph Databases

There is a variety of database types available and the main categories are SQL and NoSQL databases. A short description of SQL databases would be "A relational database organizes data in tables (or relations). A table is made up of rows and columns. A row is also called a record (or tuple). A column is also called a field (or attribute). A database table is similar to a spreadsheet." ([3])

NoSQL databases, on the other hand, are able to store any kind of data in any record. They don't rely on a specified schema and they are able to scale horizontally at the expense of consistency. [4]

Graph databases are a type of NoSQL databases. They use graph theory to store their data as described in section 2.1. Every vertex and every edge has an unique identifier (id) in the database. Properties can be assigned to edges and nodes as key/value pairs. Additionally edges store a reference to the start and end node they are attached to. [5]

The labels mentioned at the end of section 2.1 can be seen as the properties assigned to a vertex or edge. To map a production line in which the elements like machines and products have their own real-world ids, a property can be used to store these ids as key/value pairs. Later a particular machine, for example, can be looked up by its id. That is crucial to find the data stored in the database.

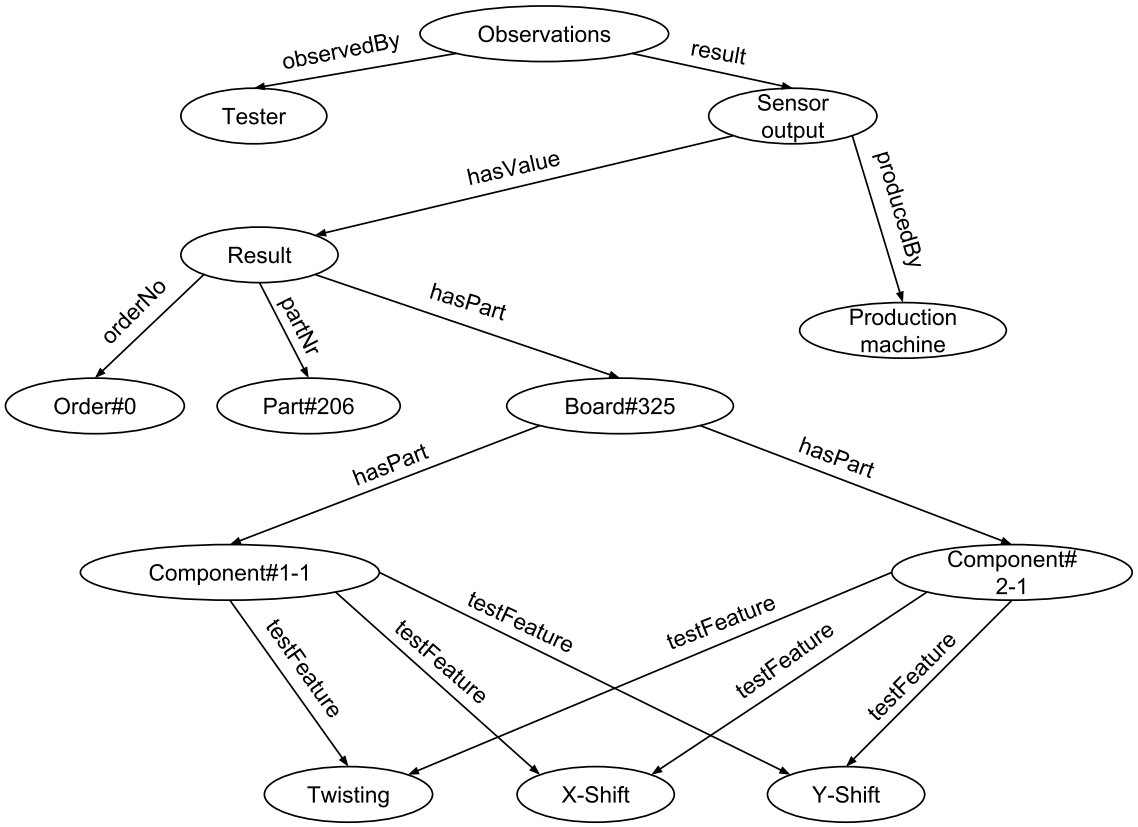

Figure 2.2: An example graph representing the observation of a board created after the example shown in listing 2.1.

In the following subsections 2.3.1 through 2.3.3 we will discuss the different types of graph databases and give examples of real databases which operate by that type. All databases used in this thesis support the ACID[1] principle and transactions.

2.3.1 RDF/Triplestores

First on our list are RDF stores also known as triple stores.

RDF (Resource Description Framework) is a model for data interchange on the web. It is able to merge data even with different schemas, it also supports the evolution of a schema over time. The linking structure of the web is extended by RDF by it using URIs[2] to name relationships and resources connected by those relationships. [6, p. 4]

> This linking structure forms a directed, labelled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations. ([7])

Triplestores store semantic facts as subject - predicate - object triples, also referred to as statements using RDF. These statements form a network of data, which can also be seen as a graph. [6, p. 4]

[1] short for atomicity, consistency, isolation, durability. It should guarantee data validity.

[2] abbreviation of Universal Resource Identifier, used to identify abstract of physical resources. https://tools.ietf.org/html/rfc3986

2.3.1.1 Apache Jena TDB

"Apache Jena (or Jena in short) is a free and open source Java framework for building semantic web and Linked Data applications. The framework is composed of different APIs interacting together to process RDF data." ([8])

Jena stores its information in statements as triples of subject, predicate and object. This structure can be seen as a graph, with the subject and the object being vertices and the predicate as an edge between them.

Jena TDBs dataset consists of the node table, triple and quad indices and the prefix table. The node table contains the representation of RDF terms and provides a mapping from Node to NodeId and the other way around. Triple and quad indices are indices for the default graph and named graphs respectively. The triple indices contain three indices for the three parts of a statement. Each index has all information about the triple, there is no secondary index. Prefixes table are mainly used in presentation and serialisation of the triples in RDF/XML or Turtle. [9]

2.3.2 Document Stores

As the name suggests the data model of document stores consist of documents which can have fields without depending on a defined schema. [10] It aggregates data in those documents and transforms them internally into a searchable form. [11]

2.3.2.1 OrientDB

OrientDB is a mix of a document store and a graph store, as stated in their manual "OrientDB is a document-graph database, meaning it has full native graph capabilities coupled with features normally only found in document databases." ([10]) It's designed as a robust, highly scalable database with a wide possible set of use cases. [10] OrientDB does not require a fixed schema and therefore supports schemaless and schema-mixed models. It uses an indexing algorithm called MVRB-Tree, which derived from the Red-Black Tree and the B+ Tree and therefore it supports fast insertions as well as fast lookups. [12]

2.3.3 Graph Stores

Graph stores organise their data as graphs. References with foreign keys known from relational databases are mapped as relationships in graph databases. Each node in the database model contains a list of relationship-records to represent their connection to other nodes. [13]

2.3.3.1 Neo4j

Neo4j is a native graph database and was built as such from the ground up. It organises its data in a graph structure and has nodes, relationships and attributes as directly accessible data structures. It can assign attributes to both nodes and edges. Neo4j is transactional and fulfils the ACID properties. [14]

2.3.3.2 Sparksee

The user manual describes Sparksee as follows, “Sparksee is an embedded graph database management system tightly integrated with the application at code level.” ([15]) Sparksee is implemented in C++ and provides a Java API.

Sparksee encodes its nodes and edges as collections of objects, which all have a unique identifier. It implements two types of structures, bitmaps and maps. Adjacency matrices are converted into multiple small indices, which improves the out-of-core workloads. Sparksee uses also efficient I/O and cache policies. The bitmaps in which the adjacency list of each node is stored are typically sparse in graphs and they are therefore compressed to save space. Attributes, which are stored in a B-Tree, are supported for both nodes and edges. Two maps are used. One which maps the object id to the attribute and the other one mapping the attribute value to the object ids that have that value. [16]

2.4 Graph Database Benchmarks

As the need to compare similar programs exists benchmarks are needed to hand results over certain aspects to aid decision making. In the field of graph databases that is no different. There exist multiple benchmarks for graph databases and some are outlined shortly in the following subsections 2.4.1 to 2.4.3. In section 3.3, we choose a benchmark for our work.

2.4.1 LDBC: Graphalytics

Benchmark specifications, practices and results for graph data management systems are established by an industry council called ”The Linked Data Benchmark Council”. The Graphalytics benchmark facilitates a choke-point design to evaluate the crucial technological challenges present in system design. One example would be the ”large graph memory footprint” as mentioned in [17, p. 2].

Graphalytics uses Datagen to create social network graphs, which are easy to understand for their users. [17, p. 3]

The workloads implemented in Graphalytics represent common graph algorithms such as ”breadth-first search”, ”weakly connected components” or ”single-source shortest paths” to name just a few. [18, p. 7]

Figure 2.3 shows the architecture of the Graphalytics benchmarking software. The user can configure the available benchmarks inside Graphalytics with a benchmark configuration (2). Parameters for the algorithm included in (1) can be specified and the system under test (4) can be set up. The system on which the benchmark runs has to be provided by the user. The harness service (5) coordinates the benchmark configuration and the benchmarking process. The dataset for the benchmark has to be provided by the user or can be generated by using the available workload generators. [18, p. 11]

2.4.2 YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) was not designed specifically for graph databases, but rather for key-value and cloud stores. The client is extensible so that new workloads, databases and generators can be integrated. [19]

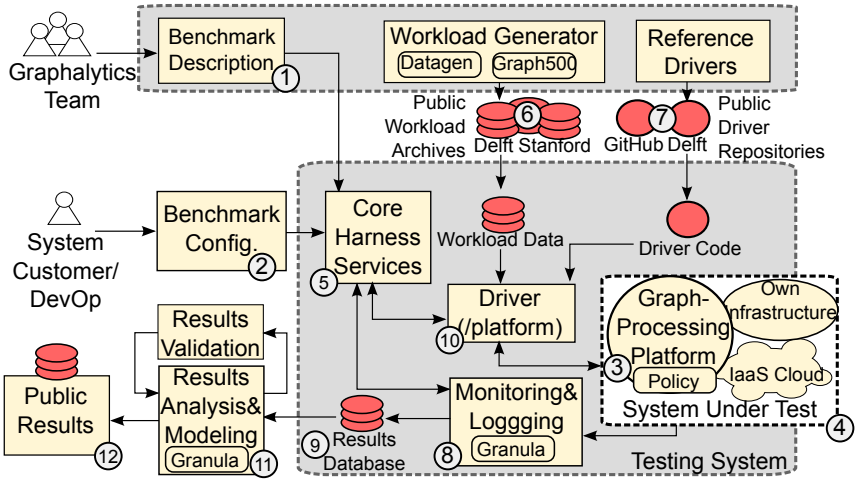

Figure 2.3: The architecture of the Graphalytics benchmark. [18, p. 11]

The core workload is designed to use simple CRUD[3] operations on any database with no special structure of the generated data.

The architecture of YCSB can be seen in figure 2.4. The client contains a `Workload executor` that uses `Generators` to create a dataset and executes operations on the database. Each `Client thread` calls the `Workload executor` to perform an operation on the database. Workload files can be specified to set the amount of data and the mix of operations to use for that workload. To tell YCSB which database should be used command-line parameters are passed to the client. The benchmark supports two phases, a load phase to fill the database with initial data and a transaction phase to execute operations on the database.

2.4.3 XGDBench

Is a graph database benchmark for cloud computing systems. It is designed to work in the cloud and in future exascale clouds. XGDBench is an extension of YCSB for graph databases. This benchmark is written in X10, a “programming language that is aimed for providing a robust programming model that can withstand the architectural challenges posed by multi-core systems, hardware accelerators, clusters, and supercomputers” ([20]).

XGDBench also focuses on social networks for their data structure. The data is generated by a procedure called ”Multiplicative Attribute Graph” (MAG), see [21] for more information.

It specifically targets read, update and graph traversal operations for its performance aspects. [20, p. 366]

In figure 2.5 the architecture of the XGDBench benchmark is illustrated. The general workflow is similar to the one from YCSB, because this benchmark is based on YCSB. The workload is executed in two phases, the load phase fills the database with data and the transaction phase executes operations on the database.

[3]CRUD stands for the basic operations on persistent storage, these are Create, Read, Update, Delete.

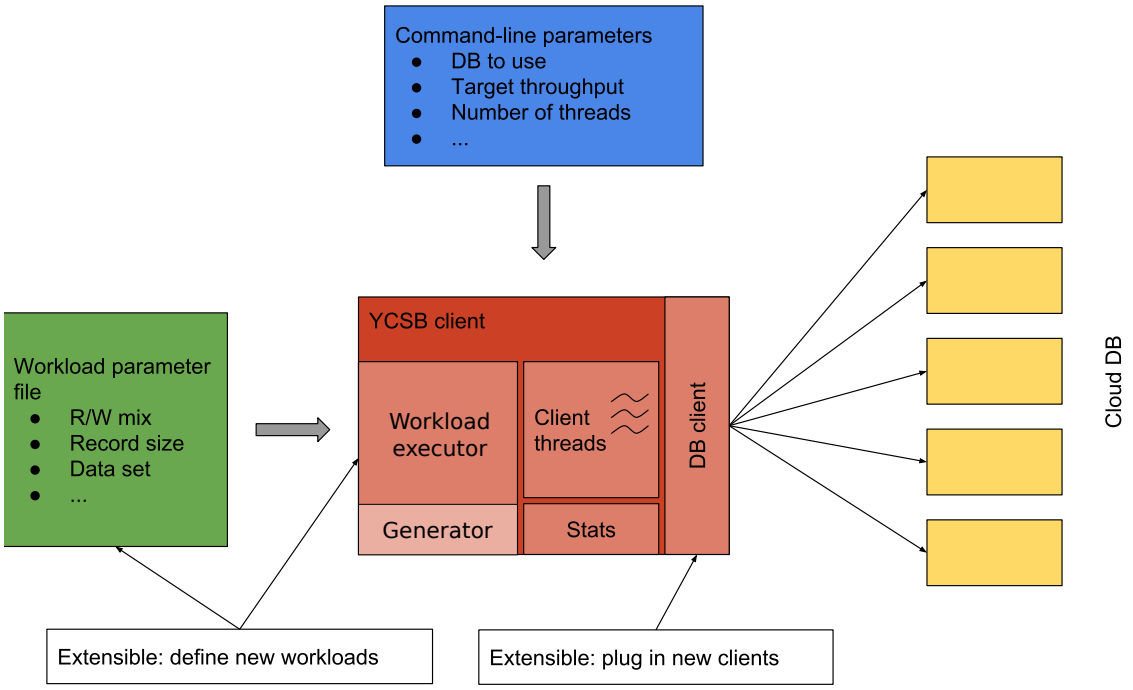

Figure 2.4: The architecture of YCSB. Recreated and modified from [12, p. 25].

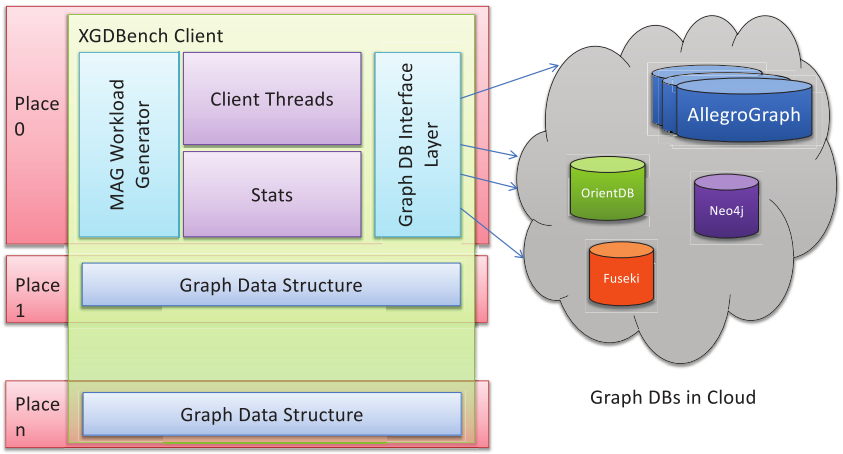

Figure 2.5: The architecture of the XGDBench benchmark. [20, p. 367]

2.5 Related Work

A number of studies have been conducted to test the performance of NoSQL databases, which we will discuss in this section. TODO: Revisit

2.5.1 Evaluation of NoSQL Systems using YCSB

Abubakar et al. have not focused their research on graph databases, but the used OrientDBs document API among others. They compared the following NoSQL databases MongoDB, ElasticSearch, OrientDB and Redis. Three different workloads were used in their study, each concentrating on one operation, which were inserts, reads and updates. The dataset size was varied between 1.000 and 100.000 records with a record size of 1KB. Their insert workload showed that OrientDB was the

slowest among the examined database across all dataset sizes whereas Redis was the fastest in this category. The read workload had no consistent order for the databases, for every dataset size another database was the best with ElasticSearch as the fastest for the largest dataset. [12]

2.5.2 HPC Scalable Graph Analysis Benchmark

Dominguez-Sal et al. implemented the HPC Scalable Graph Analysis Benchmark and tested the performance of four different graph databases. In their research they examined Neo4j, Jena (RDF), HypergraphDB and DEX (Sparksee) with four different workloads covering insert performance, looking up a set of edges, building subgraphs by utilising breadth first search and finally the traversal performance. They used a dataset generated by the R-MAT algorithm with the parameters $a = 0.55, b = 0.1, c = 0.1$ and $d = 0.25$. The final dataset had a composition of nodes to edges of $N = 2^(scale)$ and $E = 8 \times N$ with weights on the edges uniformly distributed with a maximum value of $2^(scale)$. The largest dataset they used was $2^20 = 1048576$ nodes, because most databases would not finish execution within 24 hours for larger sets. They found that DEX was over one order of magnitude faster for insert and scan then the second best database, which was Neo4j. Besides that, they found out that Neo4j had scalability problems for some operations on larger datasets. Overall DEX performed best for most operations and was close to Neo4j where it was the fastest. [16]

2.5.3 XGDBench

Dayarathna et al. introduced a benchmark for cloud computing systems called XGDBench. They used the MAG algorithm for their dataset generation, which outperforms the R-MAT in terms of creating a realistic network structure. They focus their research on the examination of online social networks and used workloads based on read, update and traversal operations. Five workloads were specified, three of which focus on read operations, one has a mix of 50% read and 50% update operations and the last one reads the neighbours of a vertex trying to mimic the loading of a friend list from a person. The evaluation of their implementation of the MAG algorithm shows that it had a high cluster prominence which means it represents the social affinities found in real social networks and the graphs created by it follow the power-law distribution which is good for realistic benchmarking scenarios. A performance evaluation was executed on these graph databases, Allegrograph, Neo4j, OrientDB and Fuseki, which is a SPARQL[4] server providing a HTTP interface to Jena. Their performance evaluation of the databases shows that the graph databases perform really badly, except OrientDB, which was at least double as fast as the next best database for any workload. The benchmark was only executed with 1024 nodes, as some databases performed very poorly which made execution with more vertices not feasible. [20]

2.5.4 Graphalytics

Capotu. a. et al. created a data generator, chose workloads based on choke-points and also conducted a benchmark on four graph databases. The choke-points are

[4]SPARQL is a language to query and manipulate RDF data. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

technological challenges the graph databases are struggling with. They designed the data generator to create datasets that can help evaluate those choke-points. The data generator supports specification of the clustering coefficient, which is important for social networks, as they indicate the presence of communities in the graph. Datagen, as the data generator they developed is called, is able to create a graph with 1.3 billion edges in 3 hours on their machine and the creation of large datasets should also be archivable with relatively cheap hardware. They implemented five algorithms, which are general statistics, breadth-first search, connected components, community detection and graph evolution. Those were executed on the following platforms, Hadoop MapReduce, Giraph, GraphX and Neo4j. MapReduce is capable of performing all workloads if given enough time, but it was up to two orders of magnitude slower than Giraph and GraphX. Neo4j performed best at breadth-first search on a dataset with many edges compared to the number of nodes but failed at all workloads using the dataset created by their date generator. [17]

3. Analysis

In this chapter we will analyse the data that could occur in an industrial use case, including its structure and amount. Further we will examine possible workloads for our graph databases in section 3.2.

At the end of this chapter in section 3.3 we will choose one benchmark for our research.

3.1 Data

As described in section 2.2 we have to work with the data coming from production machines. Figure 2.2 shows how that data could look like.

Additionally, our partners at SICK AG [2] gave us the following key data of a product example.

- A component carrier is produced every three minutes.
- A component carrier has up to 64 circuit boards.
- A circuit board has up to 128 components.
- A component is tested for up to 128 test features.

With this information we will calculate the amount of data in subsection 3.1.2

3.1.1 Data Structure

Looking at the graph in figure 2.2 and the example given by SICK we can see that the data looks much like a tree with some cross edges. A root node at the top and multiple children connected to it with multiple children each. The given excerpt from figure 2.2 shows a part of a testing procedure for a board with components. Three properties of each component were observed.

We keep this structure in mind for our design in section 4.1, where we will compose the data structure for our implementation and evaluation.

3.1.2 Data Amount

To evaluate the amount of data created during production we need to know how much is produced per time unit. With the parameters mentioned in 3.1 we can calculate the maximum number of data points produced every three minutes.

$$
\begin{aligned}
n_{nodes} &= n_{componentCarrier} \\
&\quad + n_{componentCarrier} \times n_{circuitBoard} \\
&\quad + n_{componentCarrier} \times n_{circuitBoard} \times n_{component} \\
&\quad + n_{componentCarrier} \times n_{circuitBoard} \times n_{component} \times n_{test} \\
\iff &= 1 + 1 \times 64 + 1 \times 64 \times 128 + 1 \times 64 \times 128 \times 128 \\
\iff &= 1 \times (1 + 64 + 64 \times 128 + 64 \times 128 \times 128) \\
\iff &= 1 + 64 + 64 \times 128 + 64 \times 128 \times 128 \\
\iff &= 1 + 64 \times (1 + 128 + 128 \times 128) \\
\iff &= 1 + 64 \times (1 + 128 \times (1 + 128)) \\
\iff &= 1 + 64 \times (1 + 128 \times 129) \\
\iff &= 1 + 64 \times 16.513 \\
\iff &= 1.056.833
\end{aligned}
\tag{3.1}
$$

To calculate the target throughput the databases have to archive, we need to know how many edges are between the different nodes. Therefore we need a finished data structure. In the next chapter in section 4.2.2.2 we will calculate the target throughput in $\frac{inserts}{s}$ for the workload design.

We can extract the size of each data point from our given example. Each measurement is only two to three characters long, however the other values range from 1 to around 75 characters. The size for our workload should therefore be in that range.

3.2 Workloads

Workloads should represent the mix of operations executed on a database. There are two main uses for a database in an industrial environment, the first one is described in section 3.2.1. Another one is presented in section 3.2.2. The given examples are based on what we think would represent the industrial use of databases.

In section 4.2 we will specify the workloads for our evaluation, the following subsections should only motivate the specific use cases.

3.2.1 Inserting Data into the Database

It is not rare that production runs 24h a day, therefore data is produced all around the clock. Because of that the ability to store data quickly is a decisive point in choosing a database. As the machines operate, data is continuously written to the database.

3.2.2 Retrieving Data from the Database

Besides the previous mentioned continuous writing of data into the database, retrieving data from the database would be the next common use for it. That could be in the form of looking up a certain product produced in the past, to retrieve its test parameter values or to get all products made by a specific machine to check if some are faulty.

3.3 Benchmark Comparison

To choose a benchmark for our upcoming research we will look at the following aspects of each benchmark.

- Data Structure - What is the structure of the generated data?
- Workloads - What are the workload operations?
- Programming Language - Is it written in a well-known programming language or do we have to learn it first.
- Community - Is there a community for support?

The result of our comparison is shown in the following table 3.1.

Benchmark	Data Structure	Workloads	Programming Language	Community
Graphalytics	Social Network	Algorithm based	Java	small[1]
YCSB	No specific structure	CRUD based	Java	big[2]
XGDBench	Social Network	Read, Update and Graph Traversal	X10	none[3]

[1] 8 contributors and 16 forks on GitHub https://github.com/ldbc/ldbc_graphalytics
[2] 108 contributors and 1278 forks on GitHub https://github.com/brianfrankcooper/YCSB
[3] 1 contributor and 1 fork (which is from us) on GitHub https://github.com/miyurud/XGDBench

Table 3.1: Aspects of the different databases.

Since we are not using a social network structure for our data the graph generators in Graphalytics and XGDBench don't aid us much, as the generators would be difficult to extend because of their use of complex algorithms to create that structure in the generated data ([22], [20]). YCSB on the other hand doesn't serve any particular structure presumably as they aren't designed for graph databases and therefore don't need a particular structure on their data. So YCSB should be easy to extend with our data model.

For the workload aspect Graphalytics uses common algorithms which doesn't represent out workload scenario. XGDBench and YCSB offer a good set of operations, with insert, read, and scan operations.

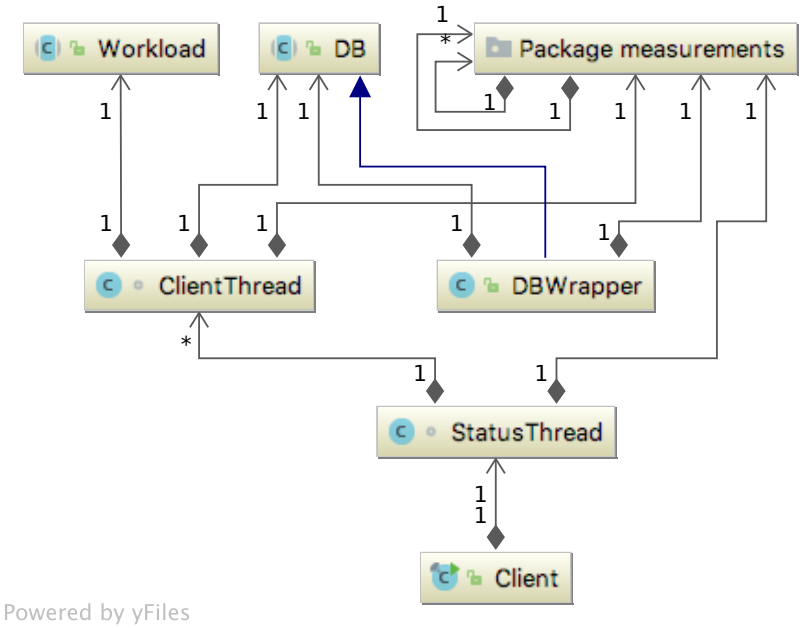

Figure 3.1: Class diagram about the main classes involved in YCSB.

Only XGDBench uses another programming language then the other benchmarks namely X10, which has to be learned from the ground up and therefore the code quality would suffer.

Lastly the community aspect. Here YCSB stands out with many contributors and an overall more active community than the other two.

All observed aspects indicate that YCSB would suit our goal the best. The generator and the workloads should be easily extendable since they have an open design[4].

3.4 YCSB

In this section we will describe the internal workflow of a benchmark run in YCSB.

YCSB separated the execution into two parts. The first part is the load phase in which the initial data is written to the database. Then comes the transaction phase where database operations are performed.

Figure 3.1 shows the classes involved in executing a benchmark run with YCSB. The `Client` takes the workload file and command-line parameters to set up the database and create `ClientThread`s. These `ClientThread`s call the `Workload` class to perform an operation on the database, which is wrapped in the `DBWrapper`. Measurements are made through the `DBWrapper` by stopping the time for every operation made on the database. To store the measurements classes from the `measurements` package are used.

The workload file specifies some parameters of the workload. These are among others, the workload class to use, how much data should be added in the load

[4]See com.yahoo.ycsb.Workload and com.yahoo.ycsb.generator.Generator in https://github.com/brianfrankcooper/YCSB

phase, how much operations should be executed in the transaction phase and what percentage of the operations should be inserts, reads, updates, scans or deletes.

The measurements can be saved as histograms each covering one particular operation. There is also a summary printed out to the console or a file depending on the parameters you set that additionally lists the overall time for the benchmark, times of the individual operations and some more meta information.

4. Design

In this chapter we will design the data structure of our test data, as well as the workloads to simulate a typical industrial use of a database.

After that we will plan our extension for YCSB in section 4.3, both for the internals of the benchmark and the bindings to connect the databases with the benchmark.

At the end in section 4.4 and 4.5 we will outline tools to support execution of the benchmark and evaluation of the results.

4.1 Data Structure

To design a schema for our data structure we had a meeting with other researchers at our institute. The result of our session can be seen in figure 4.1. In the centre left we see "Features of Interest" which could be mapped to the "testFeature" edge in the industrial example of figure 2.2 as it depicts an observation of a product. At the bottom we see a "M" which stands for "Machine", its connection to "P. Schritte"[1] shows that this machine does `1` to `n` production steps. Every production step is associated with a component which consists of a PCB[2] that has different parts, a version and a file after which it was created.

As the model shows too much detail in some areas without giving a good overview of an industrial data schema, we had to reiterate over it and get rid of some complexity where we don't need it for our purposes.

The meeting gave us a better understanding of how a production facility could handle its data and with that in mind and the objective to design a simpler schema that includes to most necessary parts of production the model shown in figure 4.2 was created.
At the top is the `Factory`, which has an `Orders` node that represents the folder for all `Order`s received by the `Factory`. A `Machine` and a `Design` are linked to the `Factory`, these represent the production machine and the design template for

[1] german for production steps
[2] short for printed circuit board

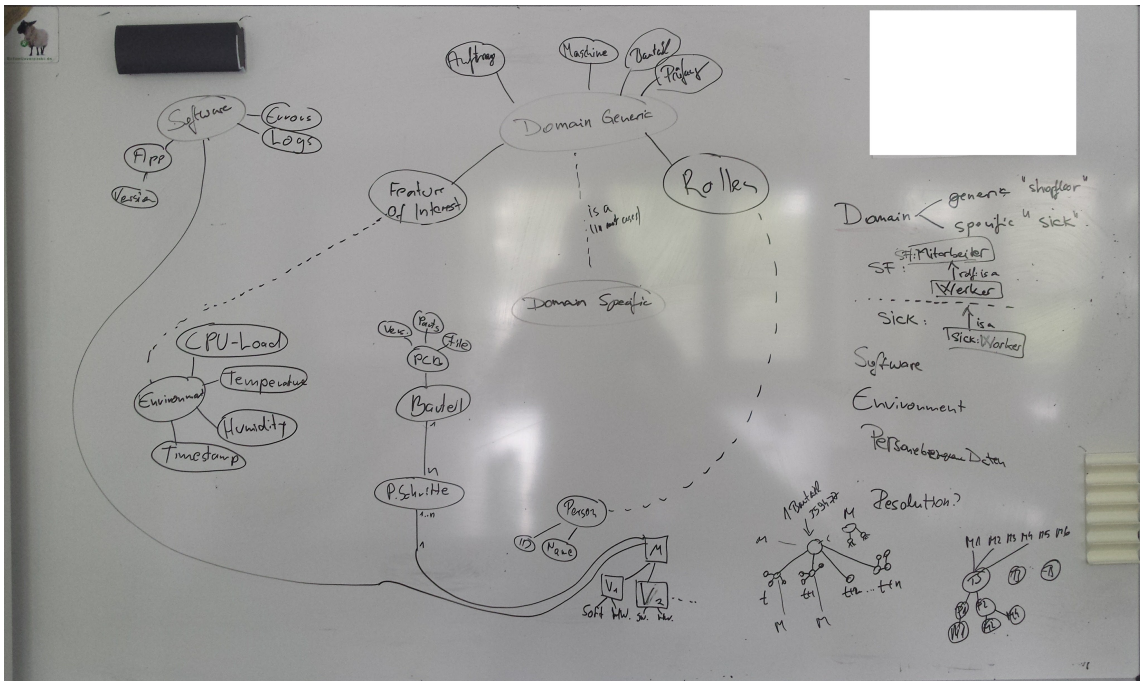

Figure 4.1: The first design of a data schema for industrial data. Created by researchers at our institute.

products made by that machine. A `Product` has incoming edges from the `Design` it is made after, the `Machine` it was produced by and the `Order` for which it is created. Variable `x` determines how many `Products` are made for each `Order`. The `Product` was made at a specific `Date` and consists of one or multiple `Components` depending on the value of variable `y`. Every `Component` undergoes a test suite (`Tests`) which contains of a number of `Test Parameters`, which number is defined by `z`.
For easier reference we will call `x` productsPerOrder, `y` componentsPerProduct and `z` testParameterCount.

4.2 Workloads

Our workload design will be separated into three parts. In subsection 4.2.1 we discuss the design of workloads aimed to reveal the ability to store large amounts of data. Subsection 4.2.2 will investigate the suitability of a database to be used in an industrial use case for storing data. We will design workloads to examine how well the databases can retrieve data under load in subsection 4.2.3. Finally we will give a summary over all workloads we are going to run on the databases in subsection 4.3.6

4.2.1 Throughput

To explore the throughput of the databases we will change some variables over the course of the different workloads. These variables are

- using an index on the key
- the size of a single property of the node
- using no edges.

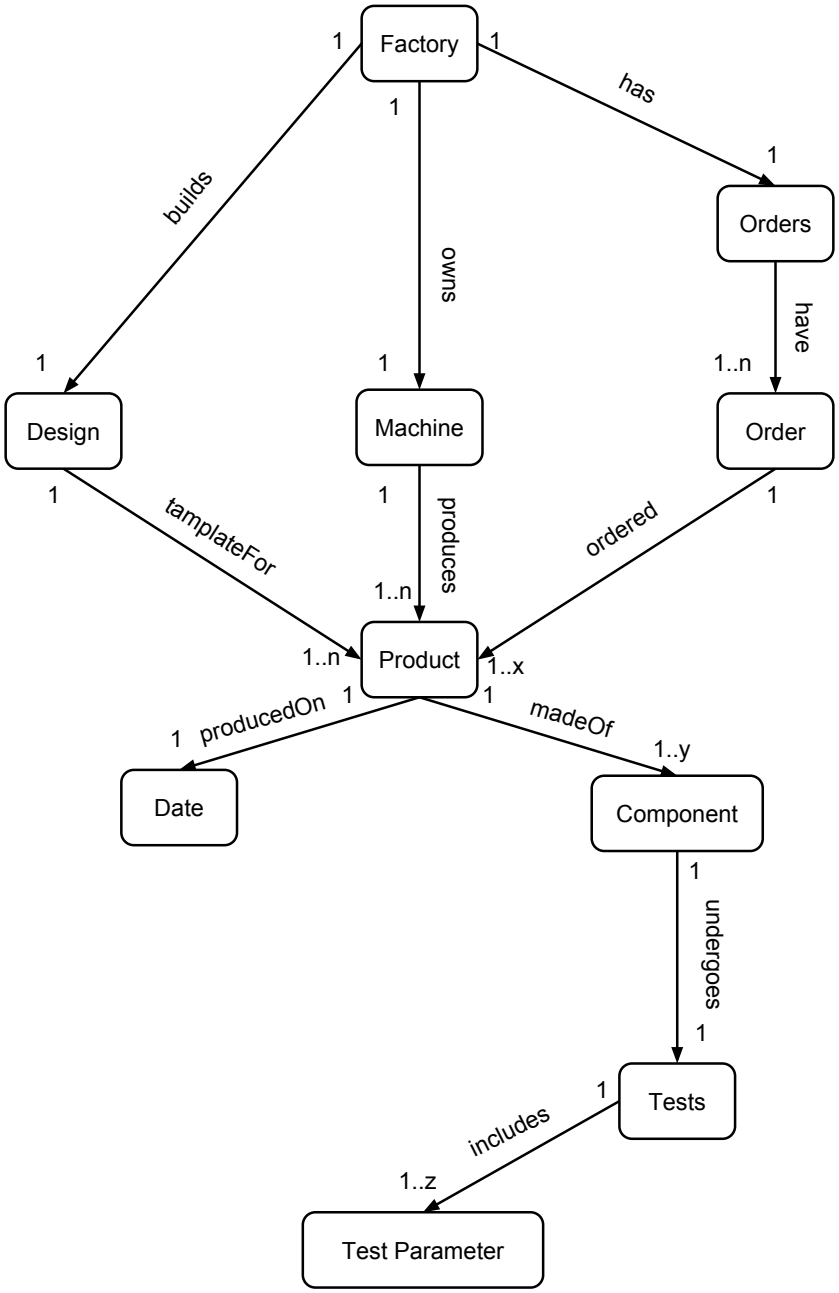

Figure 4.2: The final design of the data schema.

The last variable sounds counter intuitive, since edges add meaning to the data, but by eliminating them we want to see if edges could be a cause of delay, because to add an edge the start and end node need to be known and therefore be retrieved at first.

We will go over the different variables in the following subsections and motivate their purpose.

4.2.1.1 Index

For this category we will use different dataset sizes in terms of their number of nodes. We will use steps of multiplication by 10 from 1.000 nodes to 10.000.000 nodes, to examine if the throughput changes, as the database is filled with data.

Switching from indexed to not indexed we want to inspect how the throughput is effected by not using an index. Indexing is important to retrieve data more quickly for the cost of write speed. With this workload we will see if there is a sacrifice in write throughput, as to store an edge, two nodes have to be looked up. We will only use an index on the node and edge key, which will be used to search that graph component[3] in the database. Indexing the other properties would have no benefit in our example.

For this workload we will use a node property size of 10B. That is small enough to not have an impact on performance but large enough to represent most of the data stored in the properties of our example.

4.2.1.2 Node Property Size

After retrieving the amount of nodes that represents an acceptable execution time, we will vary the next variable which is the property size. We will go from 10B used in the index benchmarks to up to 1MB, again in steps of multiplying by 10 (10B, 100B, 1KB, ..., 1MB). We want to examine if there is a drawback for throughput when storing more information in the nodes and at which point the size is too large.

The typical property size is between 1B (1 character) and roughly 75B (75 characters) according to our example in listing 2.1.

The use of properties is not limited to short strings, that is why we will investigate if larger amounts of data influence the throughput more than linearly.

We will use an index on the keys, because that represents the use in the industry and since we are not indexing the growing values there will be no impact from using it.

4.2.1.3 No Edges

In subsection 4.2.1 we already justified why we will investigate the throughput with an exclusive use of nodes in the dataset. To summarise, the use of this workload is to see if there is a big difference in using edges and therefore determine how the edge to node ratio effects the throughput.

As in subsection 4.2.1.2 we will use a suitable large dataset in terms of node count resulting from the first workloads. We will use the same node size as in 4.2.1.1 and an index to be able to compare the result directly to the corresponding one from that workload.

[3] a node or an edge of the graph

4.2.2 Production Simulation

Related to production we will investigate the impact of the structure and the general suitability for an industrial use. The next two subsections will cover those aspects in more detail.

The property size will be set to 50B, which should be enough to cover on average most values stored in the database.

4.2.2.1 Structure

For production we have some variables to investigate, which affect the structure of our data and the edge to node ratio. We have three layers which we can scale up horizontally by increasing the corresponding parameters, which are

- productsPerOrder, this spreads the data graph apart at a level closer to the root
- componentsPerProduct, this changed the width in the middle of the graph
- testParameterCount, which widens the graph at the lowest level.

For production simulation we will first examine if the data structure impacts performance of the databases. To investigate this aspect, we will change the width of the graph with the variables mentioned above. We will use the numbers from section 3.1 as the maximum width, which would be `productsPerOrder = 64`, `componentsPerProduct = 128` and `testParameterCount = 128`. In the first workload we will set all variables to one, the next one will use `productsPerOrder = 16`, `componentsPerProduct = 32` and `testParameterCount = 32`. The third and last one will use the maximum width mentioned above. By this variation we will cover the minimum and maximum with an additional result in the middle to see if there are any changes in performance.

The keys of the graph components will be indexed, because indexing these values should be done to later work on that data more efficiently, which is necessary for the industry.

4.2.2.2 Suitability

To examine whether a database is suitable for the industry it should be able to store the data faster than it is coming from the machines. In section 3.1.2 we calculated that 1056833 nodes would be written to the database every three minutes.

Now that we have our data structure we can calculate how many edges are contained in that graph and finally how many inserts have to be performed every second. We

will count the incoming edges for every node and also use the variables `x`, `y` and `z` from the structure in 4.1.

$$
\begin{aligned}
n_{edges} &= n_{Order} + 3 \times n_{Product} + n_{Product} \\
&\quad \times (n_{Date} + n_{Component} \times (n_{Tests} + n_{TestParameter})) \\
\iff &= 1 + 3 \times x + x \times (1 + y \times (1 + z)) \quad | \quad x = 64, y = 128, z = 128 \\
\implies &= 1 + 3 \times 64 + 64 \times (1 + 128 \times (1 + 128)) \\
\iff &= 1 + 192 + 64 \times (1 + 128 \times 129) \\
\iff &= 1 + 192 + 64 \times (1 + 16512) \\
\iff &= 1 + 192 + 64 \times 16513 \\
\iff &= 1 + 192 + 1056832 \\
\iff &= 1057025
\end{aligned} \tag{4.1}
$$

Together with the number of nodes we can calculate the total amount of elements being inserted into the database, as shown in equation 4.3.

$$
\begin{aligned}
n_{total} &= n_{nodes} + n_{edges} \\
\iff &= 1056833 + 1057025 \\
\iff &= 2113858
\end{aligned} \tag{4.2}
$$

To convert that into our target throughput we divide that number by three minutes.

$$
\begin{aligned}
n_{target} &= n_{total} \times \frac{1}{3 \times 60s} \\
\iff &= 2113858 \times \frac{1}{180s} \\
\iff &= 11743,66\frac{1}{s}
\end{aligned} \tag{4.3}
$$

First, we will set up a dataset with that number of nodes and insert it into the database. That will allow us to compare the time needed to store all data with our three-minute limit. If the database should take more than three minutes, it would not be suitable, since data is produced faster than it can be stored.

We will use the structure with the maximum width, because it represents the industrial use case the best regarding the information given by our partners at SICK AG [2].

4.2.3 Retrieving under load

There would be no point in storing data if it is not retrieved at some point. To investigate the performance of reading and scanning (more on that in subsection 4.2.3.2) data from the database, the following workloads are designed.

As mentioned in section 4.2.1.1 indexing is important for retrieving data, therefore we will use it as a variable for this workload category. By doing so we want to

examine if the price we pay while writing is justified by the performance gain in retrieving data.

The node amount will be determined by the first workload investigating the throughput, to not take up too much time testing these features.

We want to retrieve both nodes and edges, because either could be useful, since the edges also can have informations stored in them.

4.2.3.1 Reading

Reading single values is the basic operation when it comes to retrieving data from a database. Since the database will be under constant load, because of production delivering data all the time, we will use 5% of the total operations executed in this workload for read operations, the rest will be insert operations.

4.2.3.2 Scanning

Scanning a graph can be done in multiple ways, one of them is depth first search [23], to retrieve values associated with connected nodes. For example, you could start scanning from a machine to get the test features of its produced products.

As in subsection 4.2.3.1 we will use a mix of 5% scan operations and 95% insert operations, to simulate the constant load present in an industrial environment.

The number of steps to do during scanning will be 1000 as that was the default value set in YCSB and it should also represent a good amount of data to read.

4.2.4 Summary

In this subsection we will give an overview over all workloads and their variables.

For the workloads measuring the throughput `productsPerOrder`, `componentsPerProduct` and `testParameterCount` will all be set to 1. Their overview is shown in table 4.1

The workloads to investigate the suitability for the industry are shown in table 4.2. For these workloads the property size is fixed to 50B and an index is used on all workloads. Edges are also used in these workloads to reflect the use in the industry.

The remaining workloads to examine the ability to retrieve data, are shown in table 4.3. These workloads will use an appropriate dataset size regarding execution time and a property size, as in the production simulation, of 50B. A simple structure is used to investigate the basic capabilities of data retrieval, that means `productsPerOrder`, `componentsPerProduct` and `testParameterCount` are set to 1.

4.3 Extension of the Benchmark

To be able to execute the introduced workloads and use the data structure designed above, we need to extend the YCSB benchmark. For the benchmark to be able to execute our workloads the way we want them to be executed the following parts of the benchmark need to be extended

- Generation of the dataset

Aspect	Node Count	Node Size	Index	Only Nodes
1. With Index	1.000	10B	True	False
2. With Index	10.000	10B	True	False
3. With Index	100.000	10B	True	False
4. With Index	1.000.000	10B	True	False
5. With Index	10.000.000	10B	True	False
1. Without Index	1.000	10B	False	False
2. Without Index	10.000	10B	False	False
3. Without Index	100.000	10B	False	False
4. Without Index	1.000.000	10B	False	False
5. Without Index	10.000.000	10B	False	False
1. Node Size	x	100B	True	False
2. Node Size	x	1KB	True	False
3. Node Size	x	10KB	True	False
4. Node Size	x	100KB	True	False
5. Node Size	x	1MB	True	False
1. No Edges	x	10B	True	True

Table 4.1: Workloads to investigate the throughput. x is a placeholder for a suitable dataset size in terms of execution time.

- Generation of random graph components
- Generation of an operation order
- Workload to use the generated dataset
- Database bindings.

In the following subsections we will go in more detail over the different areas we are planning to modify.

4.3.1 Graph Data Generator

YCSB doesn't include a graph data generator, therefore we need to create one that fulfils our needs.

The generator should create a dataset with the structure mentioned in section 4.1 and store the data for future reproduction when using the benchmark with the next database.

The two parts of the generator, one that creates and stores the data and one that recreates the data, are designed in subsection 4.3.1.1 and 4.3.1.2 respectively.

Generally, to represent a graph in YCSB we need some classes to represent nodes, edges and the graph. In section 2.1 we mentioned that a graph is a tuple of a set

Aspect	Node Count	productsPer-Order	components-PerProduct	test-Parameter-Count
1. Structure	x	1	1	1
2. Structure	x	16	32	32
3. Structure	x	64	128	128
1. Suitability (three minutes)	1.056.833	64	128	128
2. Suitability (hour)	21.136.660	64	128	128
3. Suitability (day)	507.279.840	64	128	128
4. Suitability (week)	3.550.958.880	64	128	128
5. Suitability (month)	15.218.395.200	64	128	128
6. Suitability (year)	185.157.141.600	64	128	128

Table 4.2: Workloads to simulate production. Again, x represents a placeholder for a suitable dataset size.

Aspect	Index	Insert Proportion	Read Proportion	Scan Proportion
1. Reading	True	95%	5%	0%
2. Reading	False	95%	5%	0%
1. Scanning	True	95%	0%	5%
2. Scanning	False	95%	0%	5%

Table 4.3: Workloads to investigate capability to retrieve data under load.

of nodes and a set of edges. That can be directly mapped to a class with two lists, one for nodes and the other one for edges. We want the nodes to have a key for identification, a label to match it with an object that could exist in the real world and a value, which will represent the data stored in the node. The size of this value should be directly linked to the property size from 4.2.1.2. An edge should also have a key for identification, a label to add meaning to it and a start and an end node, represented by their identification keys.

The generator of the dataset should decide whether it should create a new one or recreate the dataset, by looking at the existing files.

4.3.1.1 Storing the Dataset

We want to control the size of the dataset with our variables mentioned in the workload section 4.2 so this generator should create small subgraphs with only one node and its corresponding edges every time it is called for a new value. By storing the current state of the created graph in the generator class, we can always determine the next subgraph to create.

The modify the structure of the graph with our three variables, these need to be passed into this class and used during subgraph creation.

To restore that data also one node at a time we will store each created subgraph in a file, for that we will serialise the graph and deserialise it when we are restoring the data.

To disable edges for the workload from subsection 4.2.1.3 we can simply skip the step of creating and adding them to the graph.

4.3.1.2 Restoring the Dataset

The recreation of the data should be easily accomplishable by deserialising it from the created file during creation of the dataset. Since the single subgraphs were stored in the file, we can pass them to the workload directly after deserialising them.

To avoid memory issues with larger datasets the subgraphs shouldn't be read all at the beginning but rather on demand, by deserialising only one line when called.

4.3.2 Random Graph Component Generator

Reading and scanning operations require a point to start with in the data, that's why we need the key of some component in the graph. The kay can be randomly chosen, but the node or edge associated with it has to be present in the database. Therefore, we need to somehow store the keys of the graph components we have already inserted into the database. That could be done in the `GraphDataGenerator` mentioned in subsection 4.3.1.1, because it touches all generated components anyways.

Because we want to retrieve edges and nodes randomly we have to pick the kind of graph component randomly every time it is required. As in 4.3.1.1 and its subsections, every created value needs to be stored to be retrieved later on. The data needed for this generator is not as complex as a graph and can therefore be stored directly in a file line by line for easy storing and restoring. That also means, that we can read the files at the beginning of the run so it is faster accessible during the benchmark without using too much memory.

For the workload which requires the absence of edges a method should be defined to return only a randomly chosen node.

Figure 4.3 shows an activity diagram of the generator returning a random graph component.

4.3.3 Operation Order Generator

To fix the execution order of inserting and retrieving data to and from the graph, we need to save the operations too. That can be done by simply storing the name of each operation in a file as it appears and reading it from there when running the benchmark.

In YCSB there is already a `DiscreteGenerator`[4] that takes pairs of wights and values and returns, distributed according to the weights, a value. This can be used to get the operations to run on the database.

Figure 4.4 visualises the procedure to return the next operations.

[4] com.yahoo.ycsb.generator.DiscreteGenerator

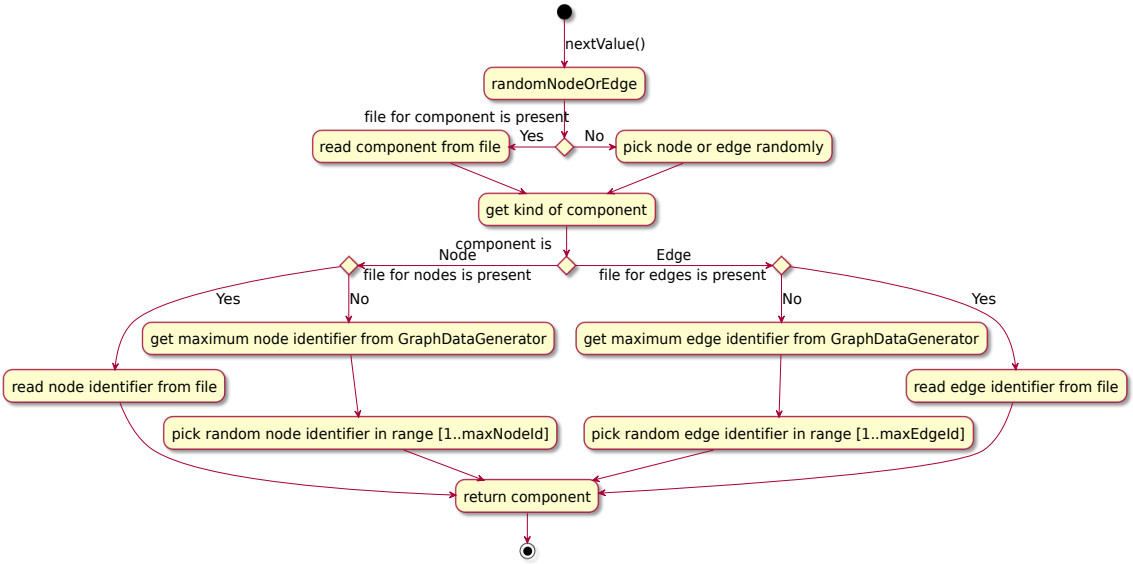

Figure 4.3: Activity diagram of the `RandomGraphComponentGenerator`, showing the process of storing and restoring a random graph component.

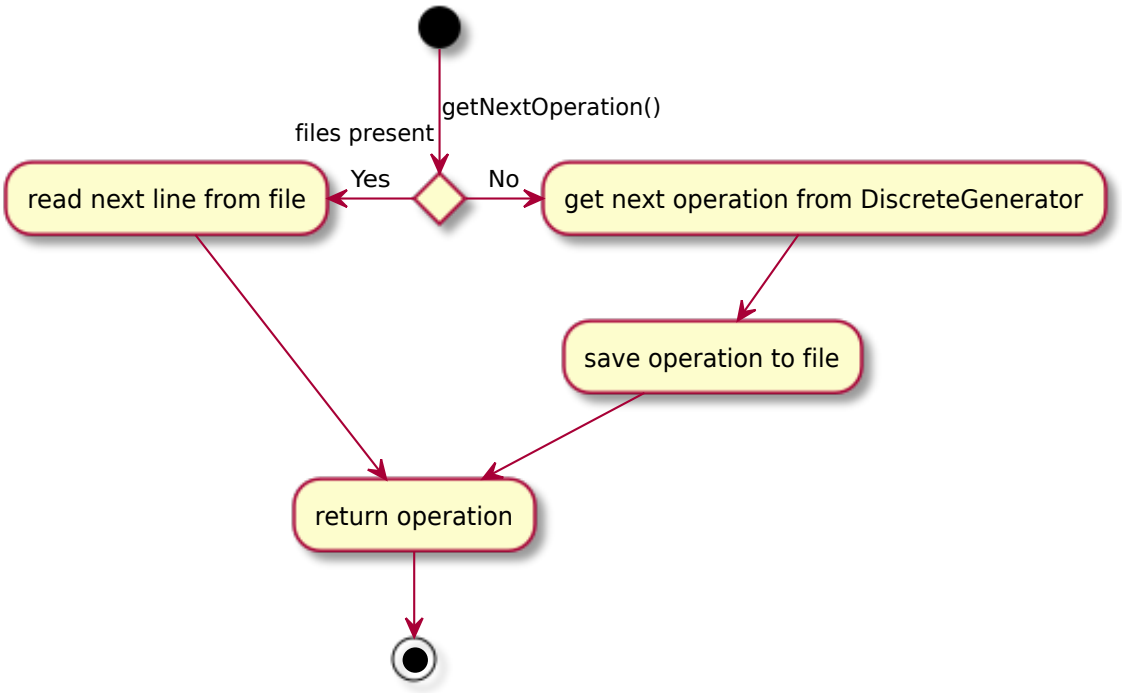

Figure 4.4: Activity diagram of the `OperationOrderGenerator::getNextOperation`.

4.3.4 Graph Workload

The `GraphWorkload`s task is to coordinate the different generators and to execute the workload as specified. To be able to store the generated dataset in a specific folder on the system the workload class should take a path to a folder and instrument the generators to store their data in that folder and recreate it from there respectively.

This class will be the interface between the client calling `Workload::doInsert` and `Workload::doTransaction` and the database. The `Workload::doInsert` method will only insert a subgraph into the database. To do so the workload class needs to get the subgraph from the `GraphDataGenerator` and redirect its value to the database. For the `Workload::doTransaction` method, the workload has to be able to call the available methods on a database which are

- DB::insert(String table, String key, Map<String, ByteIterator> values)
- DB::read(String table, String key, Set<String> fields, Map<String, ByteIterator> result)
- DB::scan(String table, String key, int recordcount, Set<String> fields, Vector<HashMap<String, ByteIterator>> result)
- DB::update(String table, String key, Map<String, ByteIterator> values)
- DB::delete(String table, String key).

We will only use the first three for our workloads, but the other ones should be implemented too, to support future workloads. To determine which operation should be executed the `OperationOrderGenerator` from subsection 4.3.3 will be used.

We see that a `table` is given as an argument, in a graph database we don't have tables as in relational databases, so we can use it to distinguish between nodes and edges, by simply passing the string "node" or "edge" to the database. Next is a `key`, which we can use to pass the key identifier of the graph component to the database. The `values` map will contain the values of the graph components parsed into a map when inserting data and vice versa for the `result` map and vector when retrieving data from the database. Our data design doesn't focus to much on the individual properties nodes and edges could have, therefore we will simply read all `fields` of the graph component.

DB::insert
As described above, the `DB::insert` method will take a value from the `GraphDataGenerator` and insert it into the database.

DB::read
The read operation will pick a random graph component with the `RandomGraphComponentGenerator`, use its kind (node or edge) as the `table` argument and the key identifier as the `key` argument.

DB::scan
Scanning also requires a random component which will be chosen by the `RandomGraphComponentGenerator`. The mapping is also the same as in `DB::read` for the

`table` and `key` arguments. `recordcount` will be set to 1000 as that is the default value specified by the `CoreWorkload`[5] and that value represents a good depth for scanning.

DB::update
For this operation we need a randomly picked graph component from the `RandomGraphComponentGenerator` to get a valid key identifier. Only the property value should be changed during update, not the identifier nor the label. That means that only nodes will be changes, as edges have no property value assigned to them.

DB::delete
Delete takes a random graph component via the `RandomGraphComponentGenerator` and calls the `delete` method of the database with the kind and key of the component.

To avoid calling these methods with edges when the workload specifies to not use them, a parameter which can be set should determine whether a random graph component or random node should be picked by the `RandomGraphComponentGenerator`.

Since the client only calls `Workload::doTransaction` to execute one of the various database operations the `OperationOrderGenerator` should be called to generate the next operation.

4.3.5 Bindings

To ensure compatibility with other workloads present in YCSB we will extend the `DB` class and implement the methods used for other databases. Because graph databases are slightly different we will explain how each database will map the arguments of the `DB` methods to their own API in the following subsections.

The basic functions we need from our database are

1. creating a node
2. creating an edge
3. adding properties to a node
4. adding properties to an edge
5. getting a node by its identifier
6. getting an edge by its identifier
7. getting the values of a node
8. getting the values of an edge
9. getting the outgoing edges of a node
10. getting the start node of an edge
11. removing a node

[5] com.yahoo.ycsb.workloads.CoreWorkload

12. removing an edge

Generally, the DB operations can then be implemented using these functions. A generic implementation is shown in listing 4.1. Every database will take a path to a folder in which it will store its internally used files. Also, if indexing is available as an option the database should take that as a parameter to set itself up accordingly.

We will cover the implementation of the individual methods in section 5.6. The following subsections will only mention specialities regarding the corresponding database.

```
public class Database extends DB {
  private Node creatingANode(String key);
  private Edge creatingAnEdge(String key, Node startNode, Node endNode);
  private void addingPropertiesToANode(Node node, Map<String, ByteIterator> values)
      ;
  private void addingPropertiesToAnEdge(Edge edge, Map<String, ByteIterator> values
      );
  private Node gettingANodeByItsIdentifier(String key);
  private Edge gettingAnEdgeByItsIdentifier(String key);
  private HashMap<String, ByteIterator> gettingTheValuesOfANode(Node node);
  private HashMap<String, ByteIterator> gettingTheValuesOfAnEdge(Edge edge);
  private List<Edge> gettingTheOutgoingEdgesOfANode(Node node);
  private Node gettingTheStartNodeOfAnEdge(Edge edge);
  private void removingANode(String key);
  private void removingAnEdge(String key);

  private void doDepthFirstSearchOnNodes(Node node, int recordcount, Vector<HashMap
      <String, ByteIterator>> result) {
    if (result.size() >= recordcount) {
      return;
    }

    result.add(gettingTheValuesOfANode(node));

    List<Edge> edges = gettingTheOutgoingEdgesOfANode(node);

    for (Edge edge : edges) {
      Node startNode = gettingTheStartNodeOfAnEdge(edge);
      doDepthFirstSearchOnNodes(startNode, recordcount, result);
    }
  }

  private void doDepthFirstSearchOnEdges(Node node, int recordcount, Vector<HashMap
      <String, ByteIterator>> result) {
    if (result.size() >= recordcount) {
      return;
    }

    List<Edge> edges = gettingTheOutgoingEdgesOfANode(node);

    for (Edge edge : edges) {
      result.add(gettingTheValuesOfAnEdge(edge));

      Node startNode = gettingTheStartNodeOfAnEdge(edge);
      doDepthFirstSearchOnNodes(startNode, recordcount, result);
    }
  }

  @Override
  public Status insert(String table, String key, Map<String, ByteIterator> values)
      {
    switch(table) {
    case "Node":
      Node node = creatingANode(key);
      addingPropertiesToANode(node, values);
      break;
    case "Edge":
      Node startNode = gettingANodeByItsIdentifier(values.get("startNode").toString
          ());
```

```
      Node endNode = gettingANodeByItsIdentifier(values.get("endNode").toString());
      Edge edge = creatingAnEdge(key, startNode, endNode);
      addingPropertiesToAnEdge(edge, values);
      break;
    default:
      return Status.NOT_FOUND;
    }
    return Status.OK;
  }

  @Override
  public Status read(String table, String key, Set<String> fields, Map<String,
      ByteIterator> result) {
    switch(table) {
    case "Node":
      Node node = gettingANodeByItsIdentifier(key);
      result = gettingTheValuesOfANode(node);
      break;
    case "Edge":
      Edge edge = gettingAnEdgeByItsIdentifier(key);
      result = gettingTheValuesOfAnEdge(edge);
      break;
    default:
      return Status.NOT_FOUND;
    }
    return Status.OK;
  }

  @Override
  public Status scan(String table, String startkey, int recordcount, Set<String>
      fields, Vector<HashMap<String, ByteIterator>> result) {
    switch(table) {
    case "Node":
      Node node = gettingANodeByItsIdentifier(startkey);
      doDepthFirstSearchOnNodes(node, recordcount, result);
      break;
    case "Edge":
      Edge edge = gettingAnEdgeByItsIdentifier(startkey);
      Node startNode = gettingTheStartNodeOfAnEdge(edge);
      doDepthFirstSearchOnEdges(startNode, recordcount, result);
      break;
    default:
      return Status.NOT_FOUND;
    }
    return Status.OK;
  }

  @Override
  public Status update(String table, String key, Map<String, ByteIterator> values)
      {
    switch(table) {
    case "Node":
      Node node = gettingANodeByItsIdentifier(key);
      addingPropertiesToANode(node, values);
      break;
    case "Edge":
      Edge edge = gettingAnEdgeByItsIdentifier(key);
      addingPropertiesToAnEdge(edge, values);
      break;
    default:
      return Status.NOT_FOUND;
    }
    return Status.OK;
  }

  @Override
  public Status delete(String table, String key) {
    switch(table) {
    case "Node":
      removingANode(key);
      break;
    case "Edge":
      removingAnEdge(key);
      break;
```

```
    default:
      return Status.NOT_FOUND;
    }
    return Status.OK;
  }
}
```

Listing 4.1: Generic example of a database implementation with the use of graph data.

4.3.5.1 Apache Jena

Apache Jena uses transactions to work on the database, therefore we will need to open and close them as we insert or retrieve data from the database. Transactions can be opened for either read or write operations, to guarantee data validity.

To get access to the data over Jena we can use the `TDBFactory::createDataset` method to get a `Dataset` which hands us a `Model` that represents the data. All operations are executed on this `Model`.

Jena has no option to disable indexing, so we can't use it for the workloads which have an index as their variable. But we can still compare its performance to the indexed and not indexed results of the other databases.

In Jena we will use the following mapping for the method arguments.

key
Should be used on the `Model` retrieved from the `Dataset` to create a `Resource`, which would represent a node or create a `Property` to form an edge. To retrieve data the `Model::createResource` or `Model::createProperty` method can be used as well, because if the passed key is already used on another node, the returned node will be equal to the already existing node.

values
Properties can be stored as so-called `Statement`s, which represent a triple as mentioned in section 2.3.1.1. The subject will be the graph component itself, the predicate will be the identifier of the value in the map and the value will be the object of the statement.

4.3.5.2 Neo4j

To index the keys of the nodes and edges we have to create an index with an `Index Manager`. Over this `Index` the graph components have to be inserted and retrieved.

Neo4j also uses transactions, but we don't have to set them as read or write transactions, because it will mark it accordingly to the called methods.

The mapping for this database will be as follows.

key
Nodes will use the key as a native `Label` and also set it as a specific property. The property is needed to retrieve a node, as we have to find a node by passing a label, the property key and the property value to the database. Edges should use the key as the edge type, that way they can be retrieved more easily, as the type can be directly returned by an edge to compare it to the key we are looking for.

values
Neo4j directly supports setting properties with a key and a value, therefore we can directly store the values as properties in the graph components of Neo4j.

4.3.5.3 OrientDB

OrientDB also supports indexing specific keys, in contrast to Neo4j the index only needs to be enabled to be used.

Transactions are also part of OrientDB, as in Neo4j they are initially not read or write specific, but adapt as the corresponding methods are called.

OrientDB supports creating a vertex with a key and a map of values directly, but the values of the values map need to be mapped to a `String`, because `ByteIterator`s are not supported. Edges will take the key, a start and end node and a label. The label has to be set to a constant value over all edges, because edges have to be looked up by the label and the key, but the label is only handed in the `DB::insert` method. The edge properties can be set after creating the edge by passing the string map of properties.

4.3.5.4 Sparksee

Sparksee only has a very low-level API, which uses ids for all its nodes, edges and attributes.

As with OrientDB the index has only to be activated on the specific fields.

key
Nodes are created with a type, which can be the same for all nodes. After creating the node its attributes have to be set, here we will add the key to identify the node. Edges are created similarly except they need a start and end node during creation. The graph components can be retrieved by looking up the component with the attribute identifier and the corresponding value, which is the key.

values
The values can be set as `Attribute`s to a graph component, by the attribute and its corresponding value. An `Attribute`s has to be created first with a type it belongs to, which will be a node or an edge and a key, which can be the key in the values map.

4.3.6 Summary

To sum up our design decisions we will give an overview of the different parameters each class should take and why in table 4.4.

The general workflow of the generators is shown in figure 4.5.

4.4 Execution Tool

YCSB has a script to run one workload on one database. We have many workloads and multiple databases, therefore it would save us a lot of time during evaluation, if all workloads were executed on all databases sequentially.

That could be implemented as a script that takes the databases and their parameters together with the workload description files and executes one after another. The results should be saved in a specified folder.

4.5 Evaluation Tool

To gather the results another script should iterate through the result folders of each database and workload and collect the results in a file for further evaluation.

Class	Parameters	Purpose
GraphDataGenerator	folder, ”productsPerOrder”, ”componentsPerProduct”, ”testParameterCount”, ”noEdges” and ”nodePropertySize”	Return subgraphs that form the data structure described in 4.1.
RandomGraphComponentGenerator	folder	Return a randomly chosen graph component already in the database.
OperationOrderGenerator	folder	Return operations to execute on the database.
GraphWorkload	folder, recordcount and ”noEdges”	Run the workloads on the databases with the help of the different generators.
ApacheJena	dbFolder	Use the Jena TDB API to create and access the database.
Neo4j	dbFolder and ”useIndex”	Use the Neo4j API to create and access the database.
OrientDB	dbFolder and ”useIndex”	Use the OrientDB API to create and access the database.
Sparksee	dbFolder and ”useIndex”	Use the Sparksee API to create and access the database.

Table 4.4: Overview of command-line parameters and the purpose of every class.

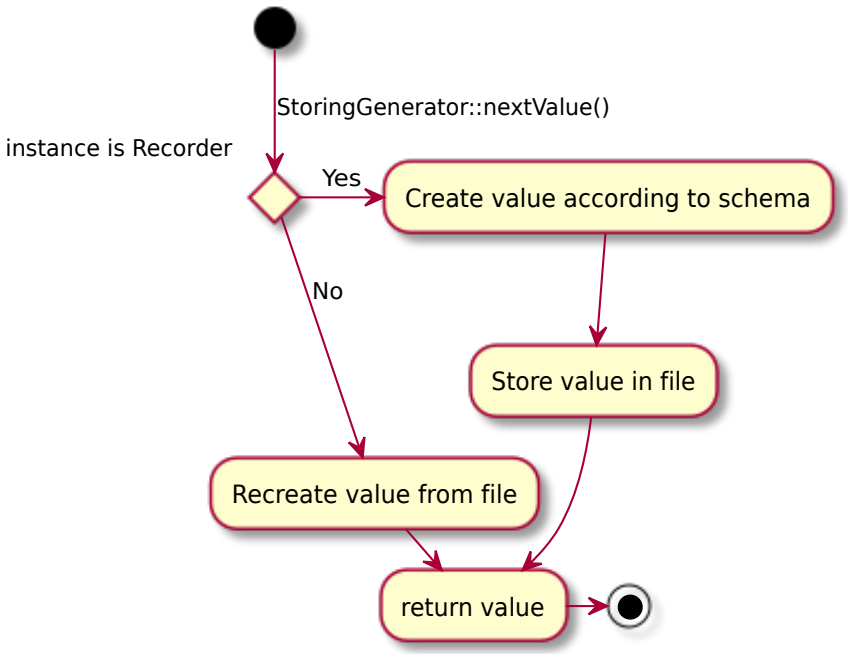

Figure 4.5: Generic activity diagram showing how the generators will work.

5. Implementation

In this chapter we will cover how we implemented the different classes to execute our workloads. We will start with the graph and its components, then move on to the different generators for the graph data, the random graph components and the operation order. Then we will show the workload class in section 5.5 and finally describe the database bindings in section 5.6.

The code of our implementation is available on GitHub[1].

In figure 5.1 we see a diagram of the YCSB benchmark with our added implementations. The classes we added are inside the red border on the right side. In `Package db` we added the bindings for our four databases.

5.1 Graph

As mentioned in section 2.1 a graph simply contains two lists, one for nodes and one for edges. This class is only a container for those two lists.

To extract some shared values of nodes and edges, we added an abstract class `GraphComponent`, that holds the identifier and the label of the graph component.

5.1.1 Node

The `Node` class assigns the identifiers by counting the created nodes and incrementing the counter for every new node. If the property value of a node is not set, a call to `Node::getHashMap` will randomly fill the property with the amount of characters specified by the `nodePropertySize` parameter.

5.1.2 Edge

As the `Node` class the `Edge` class also uses a counter field to assign the correct identifier to each edge. Additionally the ids of the start and end `Node` are stored in fields.

[1] https://github.com/ChristianNavolskyi/YCSB

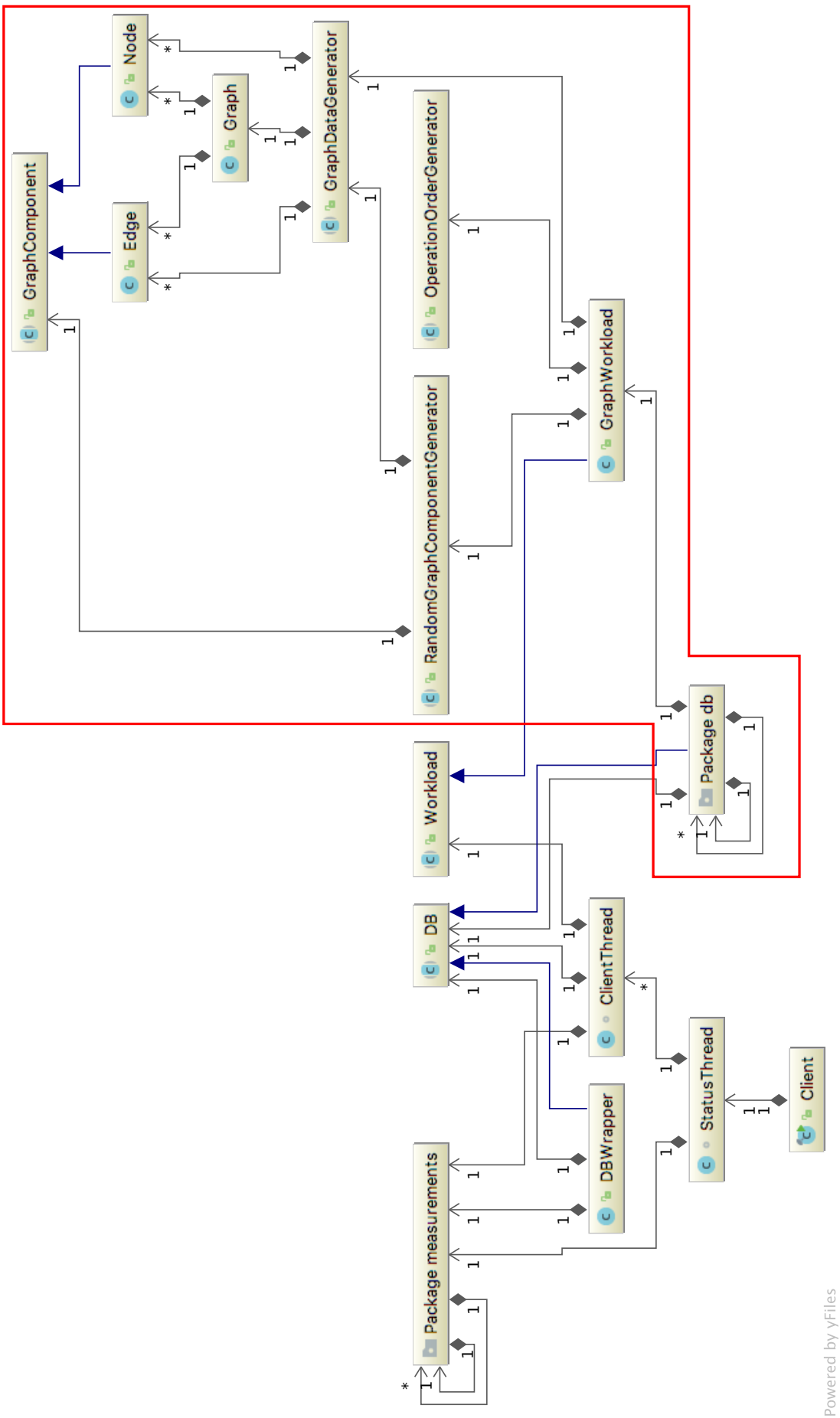

Figure 5.1: Class diagram of YCSB with the most important classes we added to it placed inside the red border.

5.2 Generator

The general workflow of a generator was mentioned at the end of section 4.3.6. Because all three generators share that behaviour we created an abstract class `StoringGenerator`[2], that extends the generic `Generator<V>`[3] class and adds methods to check if the files are present for recreation or not.

Every generator offers a `create` method, in which it will check for present files and set up the correct implementation (recorder or recreator) for the `GraphWorkload`[4]. The generator classes are all abstract and use abstract methods to call the underlying implementation. How this is useful will be described in the implementations of the different kinds of generators.

The abstract generator classes also contain the values needed for both implementation types (recorder and recreator), to avoid code duplication.

5.2.1 Graph Data

The `nextValue` call encapsulates the call to get the subgraph from the underlying implementation and also stores the returned identifiers of the created nodes and edges for the `RandomGraphComponentGenerator`[5].

The `Gson`[6] used in both implementations of this abstract class is initialised here with the `GraphAdapter`[7].

Since there are two phases of the benchmark (see section 3.4) the generator needs to know from what point it should move on with creation. When the current phase is the transaction phase, it will call the underlying implementation to create the amount of data that was created during the load phase, to equalise the progress of the generator. That is also important for the `RandomGraphComponentGenerator`, because the identifiers of the graph components created by the `GraphDataGenerator` are kept there for it to use them.

5.2.2 Random Graph Component

Calling `nextValue` on a `RandomGraphComponentGenerator` will invoke the implementing class to choose between a node and an edge. Then a random graph component of that type is chosen. A random node can also be picked directly, as it's needed for the `GraphWorkload::update` method, since it only will use nodes.

5.2.3 Operation Order

Here the generator only holds common fields shared by the recorder and the recreator.

[2] com.yahoo.ycsb.generator.StoringGenerator
[3] com.yahoo.ycsb.generator.Generator
[4] com.yahoo.ycsb.workloads.GraphWorkload
[5] com.yahoo.ycsb.generator.graph.randomcomponents.RandomGraphComponentGenerator
[6] com.google.gson.Gson
[7] com.yahoo.ycsb.generator.graph.GraphAdapter

5.3 Recorder

For every generator we have a creator that creates the initial values for the workload and stores them in a corresponding file for the recreator presented in section 5.4.

How the creation of the values is implemented in each generator is described in the following subsections 5.3.1 to 5.3.3.

5.3.1 Graph Data

As shown in figure 4.5 when `GraphDataGenerator::nextValue`[8] is called to create the next subgraph, the `GraphDataRecorder` is called and creates the subgraph according to the diagram shown in figure 5.2. Each subgraph is then serialised and the string coming from serialisation is written into a file line by line.

Table 5.1 shows how the parameters x, y and z of the data structure from figure 4.2 are implemented in that schema. They all affect when the specific if block is executed at the end of figure 5.2 to reset the corresponding values for the if blocks above. The creation of a subgraph can be seen in a loop, in every iteration another if-condition is fulfilled to return the next value.

Variable	Usage
x	Determines after how many products the order is fulfilled
y	Determines after how many components a product is finished
z	Determines after how many tests all test parameters are finished

Table 5.1: Implementation of the structure variables in the creation of the dataset.

The serialisation process is done in the `GraphAdapter` that implements both a `JsonSerializer`[9] and a `JsonDeserialzer`[10] with a `Graph` as the generic argument. Since a graph object contains two lists, these lists are serialised into `JsonElements`[11], which will be retrieved as a string by calling `Gson::toJsonTree`. The following listing 5.1 shows the Java code used to implement the serialisation of a graph.

```
@Override
public JsonElement serialize(Graph graph, Type typeOfSrc, JsonSerializationContext
    context) {
 JsonObject result = new JsonObject();

 JsonElement nodeJsonElement = gson.toJsonTree(graph.getNodes(), nodeListType);
 JsonElement edgeJsonElement = gson.toJsonTree(graph.getEdges(), edgeListType);

 result.add(nodes, nodeJsonElement);
 result.add(edges, edgeJsonElement);

 return result;
}
```

Listing 5.1: Serialisation of a `Graph` object.

[8] com.yahoo.ycsb.generator.graph.GraphDataGenerator
[9] com.google.gson.JsonSerializer
[10] com.google.gson.JsonDeserialzer
[11] com.google.gson.JsonElement

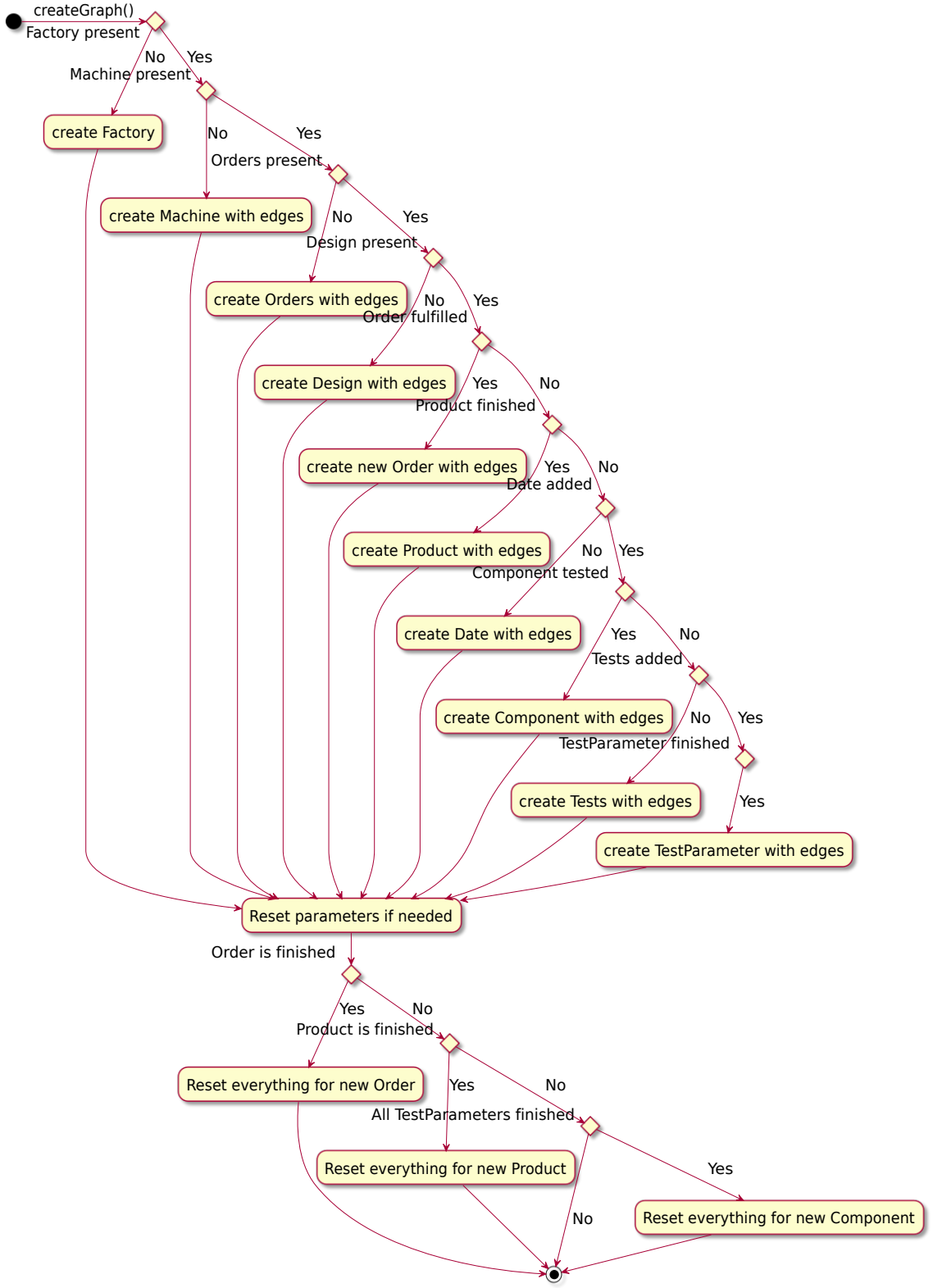

Figure 5.2: Activity diagram of the creation process for the dataset.

5.3.2 Random Graph Component

To choose between a node and an edge a random number between zero and one will be picked ($r \in \mathbb{N}_0 \wedge r \in [0, 1]$) and stored in a file. To select a random graph component the `GraphDataGenerator` will be asked what the last id was and then a

random value between zero and that number will be generated. That value will also be stored in a file corresponding to the type of the graph component.

5.3.3 Operation Order

The `OperationOrderRecorder`[12] receives a `DiscreteGenerator`[13], which supplies the string values for the operations that will be saved in a file and then returned to the caller.

5.4 Recreator

To retrieve the values stored by the recorder classes described in section 5.3 the upcoming recreators are needed.

5.4.1 Graph Data

If the files for the dataset are present the `GraphDataRecreator` will be called to return the next subgraph. It does that by deserialising the next line with the `Gson::fromJson` method which uses the `GraphAdapter` described in subsection 5.3.1 together with a `Type`[14]. The code of the `GraphAdapter` to deserialise a `Graph` is shown in listing 5.2.

```
@Override
public Graph deserialize(JsonElement jsonElement, Type type,
    JsonDeserializationContext context) throws
    JsonParseException {
  Graph graph = new Graph();
  JsonObject jsonObject = jsonElement.getAsJsonObject();

  JsonElement jsonNodes = jsonObject.get(nodes);
  JsonElement jsonEdges = jsonObject.get(edges);

  List<Node> nodeList = gson.fromJson(jsonNodes, nodeListType);
  List<Edge> edgeList = gson.fromJson(jsonEdges, edgeListType);

  nodeList.forEach(graph::addNode);
  edgeList.forEach(graph::addEdge);

  return graph;
}
```

Listing 5.2: Deserialisation of a `Graph` object.

This class uses a `BufferedReader`[15] to read the file line by line, to avoid extensive memory usage with larger datasets.

5.4.2 Random Graph Component

At the beginning the files will be read and their values will be stored in three different `Iterator<String>`s[16], one for the type and the other two for the identifiers of the different kinds of graph components.

When a value is required the corresponding `Iterator<String>` returns the next value in the list and increments its pointer.

[12] com.yahoo.ycsb.generator.operationorder.OperationOrderGenerator
[13] com.yahoo.ycsb.generator.DiscreteGenerator
[14] java.lang.reflect.Type
[15] java.io.BufferedReader
[16] java.util.Iterator<E>

5.4.3 Operation Order

As the `RandomGraphComponentRecreator` from subsection 5.4.2, this recreator reads the file directly during initialisation and stores the values in an `Iterator<String>`.

Every time `OperationOrderRecreator::nextValue` is called the next line from the `Iterator<String>` is returned.

5.5 Graph Workload

During initialisation the `GraphWorkload` creates the three generators mentioned in section 5.2, by using the `create` method. That way it will receive the correct type (recorder or recreator) for each generator. This process is shown in figure 5.3

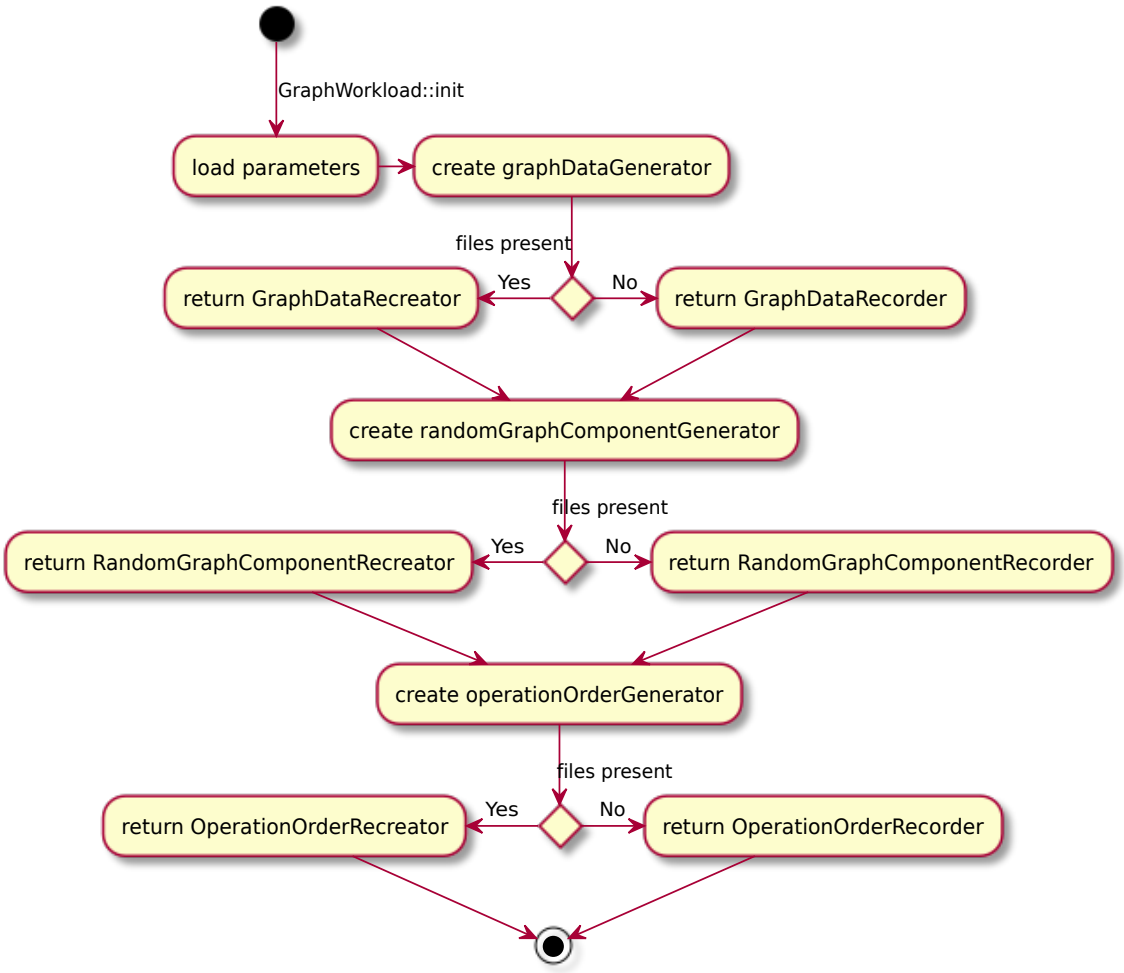

Figure 5.3: Initialisation of the generators used in the `GraphWorkload`.

It also parses the parameters to get the values for `noEdges`, the `property size` of a node, how many fields should be scanned (`recordcount`) and the `folder`. The `noEdges` parameter is needed to execute the operations on the correct available graph components. The `property size` is stored to be retrievable by the `Node` to know how many random characters it should generate. The `recordcount` option is needed for the `scan` operation. Lastly the `folder` is used to create the folder for the dataset if it isn't present and also pass it to the individual generators.

In the load phase the `Client`[17] calls `GraphWorkload::doInsert`. The `GraphWorkload` then retrieves a subgraph from the `GraphDataGenerator` by calling `GraphDataGenerator::nextValue`, separates it into its graph components and calls the `DB::insert` method with each individual component to add them to the database one by one.

If the `Client` calls `GraphWorkload::doTransaction` the `GraphWorkload` will first get the operation to execute on the database by the `OperationOrderGenerator`. After that it has an implementation for every available database operation. The general workflow for the `GraphWorkload::doTransaction` method is shown in figure 5.4.

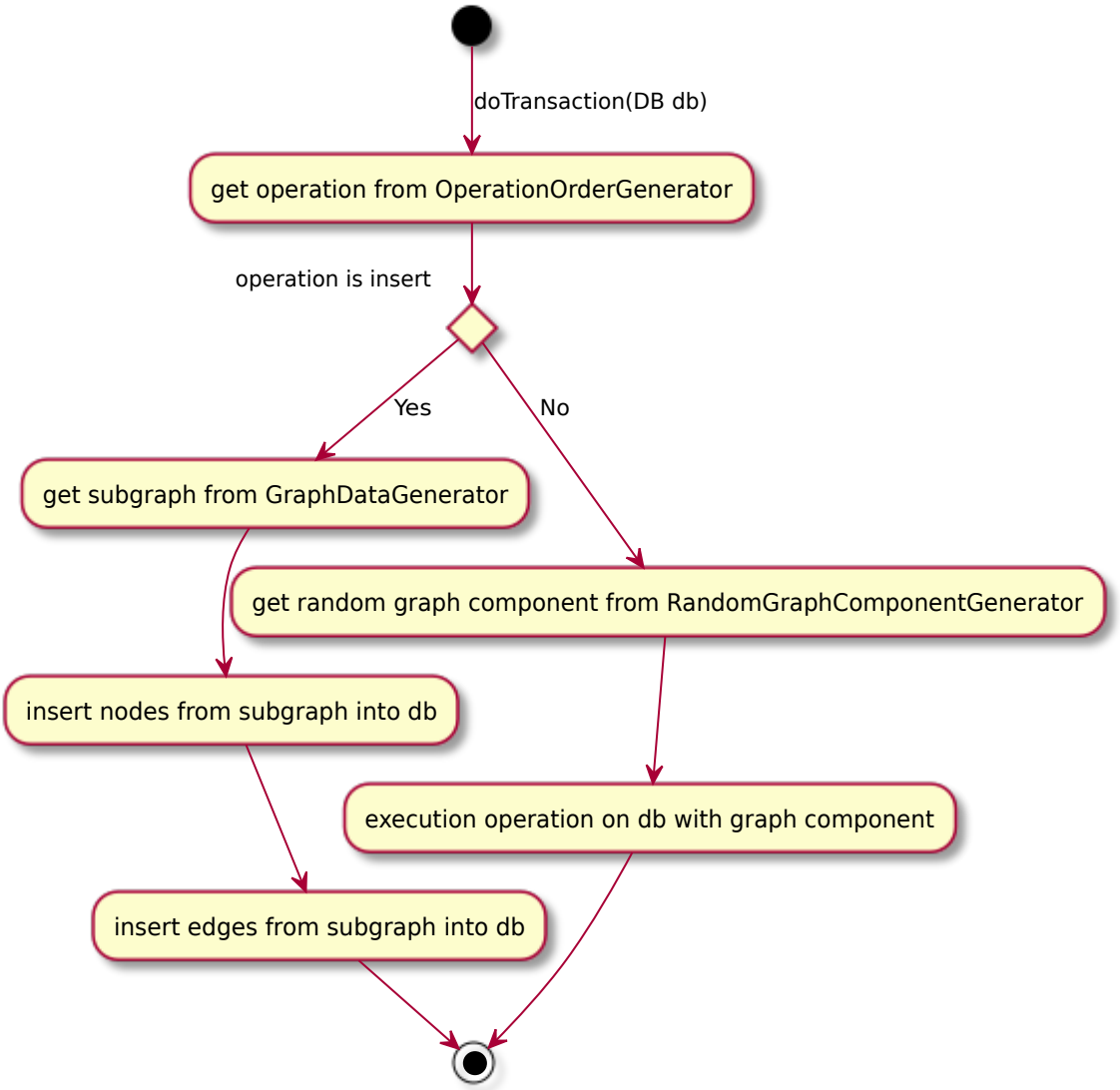

Figure 5.4: Overview of the execution of the different database operations separated into insert and other operations.

doTransactionInsert
Works as in the `doInsert` method, by taking a subgraph from the `GraphDataGenerator` and inserting its components one by one into the database.

[17] com.yahoo.ycsb.Client

doTransactionRead
Depending on the `noEdges` option the `RandomGraphComponentGenerator` will be asked for a graph component, if the option is `false` and a node if the option is `true`. With the identifier of the graph component, its type and its available fields the database is queried to look up those fields of the specified component.

doTransactionScan
As in the `doTransactionRead` method a graph component is chosen from the `RandomGraphComponentGenerator` depending on the set `noEdges` option. Then the necessary arguments from the graph component will be passed to the `DB::scan` method, alongside the specified `recordcount`.

doTransactionUpdate
The `update` method is not used by our workloads, but to make the `GraphWorkload` accessible to other workloads we implemented it as follows. A random graph component is picked and the `DB::update` method of the database is called. If the graph component is a node its property value will be randomly assigned.

We did not implement the `delete` method of the database, as we won't use it in our workloads and the `CoreWorkload` that we used as reference also did not use it.

5.5.1 Parameters

This subsection covers the naming of the parameters in the code.

Our name	Name in the code
folder	datasetdirectory
productsPerOrder	productsperorder
componentsPerProduct	componentsperproduct
testParameterCount	testparametercount
recordcount	maxscanlength
noEdges	onlynodes
nodePropertySize	fieldlength

Table 5.2: This table shows the name the parameters as they can be found in the YCSB project.

The `dbFolder` option is different for each database and will be mentioned in the corresponding binding subsection. The same goes for the `useIndex` option.

5.6 Graph Database Bindings

In this section we will describe the different binding implementations, their specialities and how they implemented the different operations mentioned in section 4.3.5. Table 5.3 shows the available options for the different databases.

At the beginning of each subsection we will show how we initialised the database and how the instance to work with the database is retrieved.

Database	Folder option	Index option
Apache Jena	outputdirectory	-
Neo4j	neo4j.path	neo4j.index
OrientDB	orientdb.url	orientdb.index
Sparksee	sparksee.path	sparksee.index

Table 5.3: Parameter names of the different databases for the database folder and the index option.

5.6.1 Apache Jena

In the following listing 5.3 the initialisation and the beginning of a transaction with the retrieval of a model to work on the data is shown.

```
String outputDirectory = getDirectoryFromProperties();
Dataset dataset = TDBFactory.createDataset(outputDirectory); // Create dataset,
    represents the database.

dataset.begin(ReadWrite.WRITE); // Starts a write transaction, ReadWrite.READ is
    used for read operations.

try {
  Model model = dataset.getDefaultModel(); // Needed to access the database.

  performOpertaionOnModel();

  dataset.commit();
} finally {
  dataset.end(); // Finish transaction.
}
```

Listing 5.3: Implementation of the initialisation and model retrieval in Jena.

To modify the database with Jena we need to start a transaction and specify whether it is a read or a write transaction. After that we retrieve the model of the database to work on the data. After we are done with our operation we need to commit or abort the transaction, similar to a relational database.

creating a node
A node is created by calling `Model::createResource`[18] with an `AnonId`[19] that receives the `key` as an argument.

creating an edge
To create an edge we use the `Model::createProperty` method with the `key` as the argument. To connect the edge with their start and end node, we have to add this triple to the model by calling `Model::add` with the start node, the edge and the end node.

adding properties to a node
Properties are mapped as statements in Jena and to create those we use the `Model::createStatement` method that takes the node, the key for the property and the property value as arguments. After all statements are created we add them to the model with `Model::add` and the list of statements as the argument.

[18] org.apache.jena.rdf.model.Model
[19] org.apache.jena.rdf.model.AnonId

adding properties to an edge
To add properties to an edge, we use the `Property::addProperty` method on it with the key of the property and its value as the arguments.

getting a node by its identifier
Retrieving a node is done by creating a resource with the same identifier. Jena will look up the database whether one already exists and the returned node will be equal to an existing one.

getting an edge by its identifier
Similar to retrieving a node from the database we create a property with the `key`, that returns an existing edge if one exists for that `key`.

getting the values of a node/an edge
To get the values associated with a node, we create a `SimpleSelector`[20], which can be used as a query on the database. We supply it the node and the key of the value and leave the object of the query empty, so it looks up the matching values for the object.

getting the outgoing edges of a node
To get these edges we list the properties of the node.

getting the start node of an edge
To do this, we take the start property of the edge and look up that node on the dataset.

removing a node
Removing a node is done by calling `Model::removeAll` twice, once with the node as the subject and once with the node as the object of the statement. That will remove all statements associated with that node, which effectively removes the node from the database.

removing an edge
Here we also call `Model::removeAll` but the with edge as the predicate of the statement.

5.6.2 Neo4j

If an `Index`[21] should be used we create two of them, one for `Nodes`[22] and one for `Relationships`[23] (edges). Neo4j also uses transaction, but we don't need to specify their kind. At the end of a transaction we call `Transaction::success`[24] to finish the transaction.

An example of our implementation is shown in the following listing 5.4. The start and end of a transaction for an operation are implemented as in the if-block of the listing.

```
String path = getPathFromProperties();
boolean useIndex = shouldUseIndex();
```

[20] org.apache.jena.rdf.model.SimpleSelector
[21] org.neo4j.graphdb.index.Index<T extends PropertyContainer>
[22] org.neo4j.graphdb.Node
[23] org.neo4j.graphdb.Relationship
[24] org.neo4j.graphdb.Transaction

```
GraphDatabaseService graphDbInstance = new GraphDatabaseFactory().
    newEmbeddedDatabase(new File(path)); // Creates to object to access the
    database.

if (useIndex) {
  try (Transaction transaction = graphDbInstance.beginTx()) { // Start a
      transaction.
    IndexManager index = graphDbInstance.index();
    nodeIndex = index.forNodes("nodes");
    relationshipIndex = index.forRelationships("relationships");
    transaction.success(); // End a transaction.
  }
}
```

Listing 5.4: Implementation of the initialisation and beginning of a transaction.

creating a node
We create a node with the `GraphDatabaseService::createNode`[25] method, where we specify the `key` as the `Label`[26] of the node. If an `Index` is used, we add the node to the index after creation. After that we add the identifier of the node as a property to be able to look the node up by its identifier.

creating an edge
For this we have to first create a `RelationshipType`[27] with the `key` as the name of the relationship. Then we create a relationship from the start node to the end node by calling `Node::createRelationshipTo`. Finally, we add the edge to the relationship `Index`.

adding properties to a node/an edge
Both `Node`s and edges are `PropertyContainer`s[28], which support the setting of properties, by calling `PropertyContainer::setProperty` with the key of the property and its value.

getting a node by its identifier
When an `Index` is used a node can be looked up on it with `Index::get`, the key for the identifier and the identifier value. Without an `Index` we call `GraphDatabaseService::findNode` with the `Label`, the key for the identifier and the identifier as arguments.

getting an edge by its identifier
With an `Index` a `Relationship` can be found similar to a node. Without an index we have to iterate over all `Relationship`s in the graph and check their types to match the `key`.

getting the values of a node/an edge
The `PropertyContainer::getAllProperties` method supplies all values set to the node or edge. We can simply parse the `Map<String, Object>`[29] returned by it to the needed `Map<String, ByteIterator>`.

getting the outgoing edges of a node
`Node`s offer a method to get their `Relationship`s in a specified `Direction`[30] with `Node::getRelationships`.

[25] org.neo4j.graphdb.GraphDatabaseService
[26] org.neo4j.graphdb.Label
[27] org.neo4j.graphdb.RelationshipType
[28] org.neo4j.graphdb.PropertyContainer
[29] java.util.Map<K, V>
[30] org.neo4j.graphdb.Direction

getting the start node of an edge
`Relationship`s also offer a method to directly get their start node with `Relationship::getStartNode`.

removing a node/an edge
To remove a `Node` or a `Relationship`, we look it up, remove it from the corresponding `Index` and then call `Node::delete` or `Relationship::delete` respectively, to remove it from the database.

5.6.3 OrientDB

To create an index in OrientDB we call `OrientGraph::createKeyIndex`[31] with the key of the identifier and the graph component classes, once with `Vertex`[32] and once with `Edge`[33]. As Neo4j OrientDB uses transactions to execute operations on the database, which have to be closed after finishing the operation by calling `OrientGraph::shutdown`.

An example of our implementation covering the initialisation and retrieval of an `OrientGraph` for a transaction is shown in listing 5.5.

```
String url = getURLFromProperties();

OrientGraphFactory factory = new OrientGraphFactory(url, userName, password); //
    Create object to access database.

if (useIndex) {
  OrientGraph graph = factory.getTx(); // Start a transaction.
  if (graph.getIndexedKeys(Vertex.class).size() == 0) {
    graph.createKeyIndex(nodeIdIdentifier, Vertex.class);
  }

  if (graph.getIndexedKeys(com.tinkerpop.blueprints.Edge.class).size() == 0) {
    graph.createKeyIndex(edgeIdIdentifier, com.tinkerpop.blueprints.Edge.class);
  }
}

try {
  performOperationOnGraph();
} finally {
  graph.shutdown(); // End a transaction.
}
```

Listing 5.5: Implementation of the initialisation and the retrieval of an `OrientGraph` for a transaction.

creating a node
To add a node, we simply call `OrientGraph::addVertex` with the `key` and the `value` map we want to put in. Before we add the value map, we have to transform the `ByteIterator`[34] values to `String`s with the `Object::toString` method.

creating an edge
An edge is created by calling `OrientGraph::addEdge` with the `key`, the start node, the end node and a label, which we will simply set to "Edge", because the label of our `values` map will be set as a property.

[31] com.tinkerpop.blueprints.impls.orient.OrientGraph
[32] com.tinkerpop.blueprints.Vertex
[33] com.tinkerpop.blueprints.Edge
[34] com.yahoo.ycsb.ByteIterator

adding properties to a node
As mentioned in ”creating a node”, the values for the properties are directly passed during creation.

adding properties to an edge
We can add the `values` to an edge by calling `OrientElement::setProperties`[35] with the map of string values.

getting a node by its identifier
A node is looked up by `OrientGraph::getVertices` with the identifier key and the identifier value.

getting an edge by its identifier
`Edge`s can be retrieved similarly, by calling `OrientGraph::getEdges` with the according parameters.

getting the values of a node/an edge
The properties of an `OrientElement` can be obtained by calling `OrientElement::getProperties`. The values of the returned map are then casted to `ByteIterator`s.

getting the outgoing edges of a node
The edges of a node can be gathered by calling `OrientVertex::getEdges` with the specified direction.

getting the start node of an edge
The procedure is analogous to that of getting the outgoing edge of a node. We call `OrientEdge::getVertex` with the specified direction.

removing a node
The `OrientGraph::removeVertex` method can be used to delete the vertex from the database.

removing an edge
As to remove a node, the `OrientGraph` provides a method to remove an edge internally, that means the connected nodes are not removed.

5.6.4 Sparksee

The index can be activated on certain attributes by `Graph::indexAttribute`[36] with the attribute and the `AttributeKind.Indexed`[37] as its arguments. Sparksee uses `Session`s[38] as transaction, these are also closed at the end of the transaction.

In the following listing 5.6 we show how we implemented the initialisation, the activation of an index and the retrieval of a graph instance to work on the database. After the graph is retrieved any operations on the database can be executed, in our example we initialised the index.

```
String path = getPathFromProperties();
boolean useIndex = shouldUseIndex();

Sparksee sparksee = new Sparksee(new SparkseeConfig());
```

[35] com.tinkerpop.blueprints.impls.orient.OrientElement
[36] com.sparsity.sparksee.gdb.Graph
[37] com.sparsity.sparksee.gdb.AttributeKind
[38] com.sparsity.sparksee.gdb.Session

```
if (new File(path).exists()) {
  database = sparksee.open(path, false);
} else {
  database = sparksee.create(path, "SparkseeDB");
}

try (Session session = database.newSession()) {
  Graph graph = session.getGraph();

  nodeIdAttribute = getAttribute(graph, getNodeType(graph), "sparksee.nodeId");
  edgeIdAttribute = getAttribute(graph, getEdgeType(graph), "sparksee.edgeId");

  if (useIndex) {
    try {
      graph.indexAttribute(nodeIdAttribute, AttributeKind.Indexed);
      graph.indexAttribute(edgeIdAttribute, AttributeKind.Indexed);
    } catch (RuntimeException e) {
      // The presence of an index cannot be queried, so we will catch and ignore
          the exception thrown when an index already exists.
      e.printStackTrace();
    }
  }
}
```

Listing 5.6: Implementation of the initialisation and starting of a session.

creating a node
To create a node, we first are creating a type for the node, which is the same for all nodes. Then we call `Graph::newNode` and set a identifier attribute to store the `key` in the node.

creating an edge
Here we are also first looking up the two corresponding nodes and then we create an edge type, that is also the same for all edges. Then we create an edge by calling `Graph::newEdge` with the type, the start and the end node. Lastly the identifier for the edge is set as an attribute.

adding properties to a node/an edge
To add attributes, we have to create an attribute in the database with the name of the property. Then we call `Graph::setAttribute` with that attribute and its value.

getting a node/an edge by its identifier
Retrieving a graph component works by creating a `Value`[39] with the key of the component, which is then passed to the `Graph::findObject` method with the attribute specifying a node or an edge.

getting the values of a node/an edge
The attributes of a graph component are obtained by calling `Graph::getAttributes`, which hands us an `AttributeList`[40] that is then looked up for the attributes we want to get.

getting the outgoing edges of a node
To get the edges connected to a node, we call `Graph::neighbors` with the node, the type of edge and the direction.

getting the start node of an edge
The `EdgeData::getHead` method serves us that node.

[39] com.sparsity.sparksee.gdb.Value
[40] com.sparsity.sparksee.gdb.AttributeList

removing a node/an edge
To remove a graph component from the database we look the component up and then call `Graph::drop` on it, to delete it from the database.

6. Evaluation

This chapter will cover the execution and evaluation of our benchmark with the workloads specified in section 4.2. We will present the results of each workload and have a discussion on them directly after that.

A conclusion will be drawn in section 7.1 in the next chapter.

6.1 Objective

The main goal is to see, if the databases are capable of handling the production workloads. Besides that, we will investigate if we can generalise graph database benchmark results by comparing our results with the results of other studies examining the performance of graph databases with social network graphs. The generalisation would allow to use benchmark results created with social network graphs for the evaluation of performance of graph databases in an industrial environment.

We will measure the time per operation and with the average we can calculate the throughput in $\frac{operations}{s}$, that way we can compare the databases without take into account the overhead of the benchmark itself, which would be reflected in the overall run time.

In section 6.3 we will show which workloads we will compare with each other.

6.2 Setup

In this section we will describe the software and hardware we used to execute the benchmark.

6.2.1 Hardware

The machine used for the benchmark had the specifications shown in table 6.1.

6.2.2 Software

The versions of the software components we used are shown in table 6.2.

Component	Description
CPU	Intel i7-3770K @ 3.5GHz
RAM	16GB DDR3 @ 1.600MHz
Storage	Seagate ST2000DL003 2 TB 5900rpm, only a 400GB partition was used
GPU	NVIDIA GeForce GTX 670

Table 6.1: The hardware specifications of the machine used for the benchmark.

Software	Version
Ubuntu	17.10
Java	1.8.0_171
OpenSSH	7.5p1
YCSB	0.14.0-SNAPSHOT
ApacheJena	3.6.0
Neo4j	3.3.4
OrientDB	2.2.33
Sparksee	5.2.3

Table 6.2: The software specifications of the machine used for the benchmark.

6.3 Overview

In figure 6.1 the execution process is illustrated and explained in the following enumeration.

Step 1: One workload is chosen from the set of workloads.

Step 2: The dataset is created for that workload.

Step 3: One database is chosen from the set of databases.

Step 4: The workload is executed on the database with the created dataset.

Step 5: The results of the benchmark run are stored in a folder specific to the constellation of workload, database and execution pass.

Step 6: Repeat from Step 4 three times.

Step 7: Repeat from Step 3 until all databases ware benchmarked.

Step 8: Repeat from Step 1 until all workloads have been executed.

In table 6.3 and 6.4 the groups of workloads we are comparing with each other are shown. The naming of the workloads is similar to the naming introduced in section 4.2.

After execution we have to combine the results for further inspection. Figure 6.2 illustrates this process of evaluation. With all the results in one place we filtered the measurements for those we wanted, then we calculated the average over the three benchmark runs we did with every database and workload. Next, we grouped the

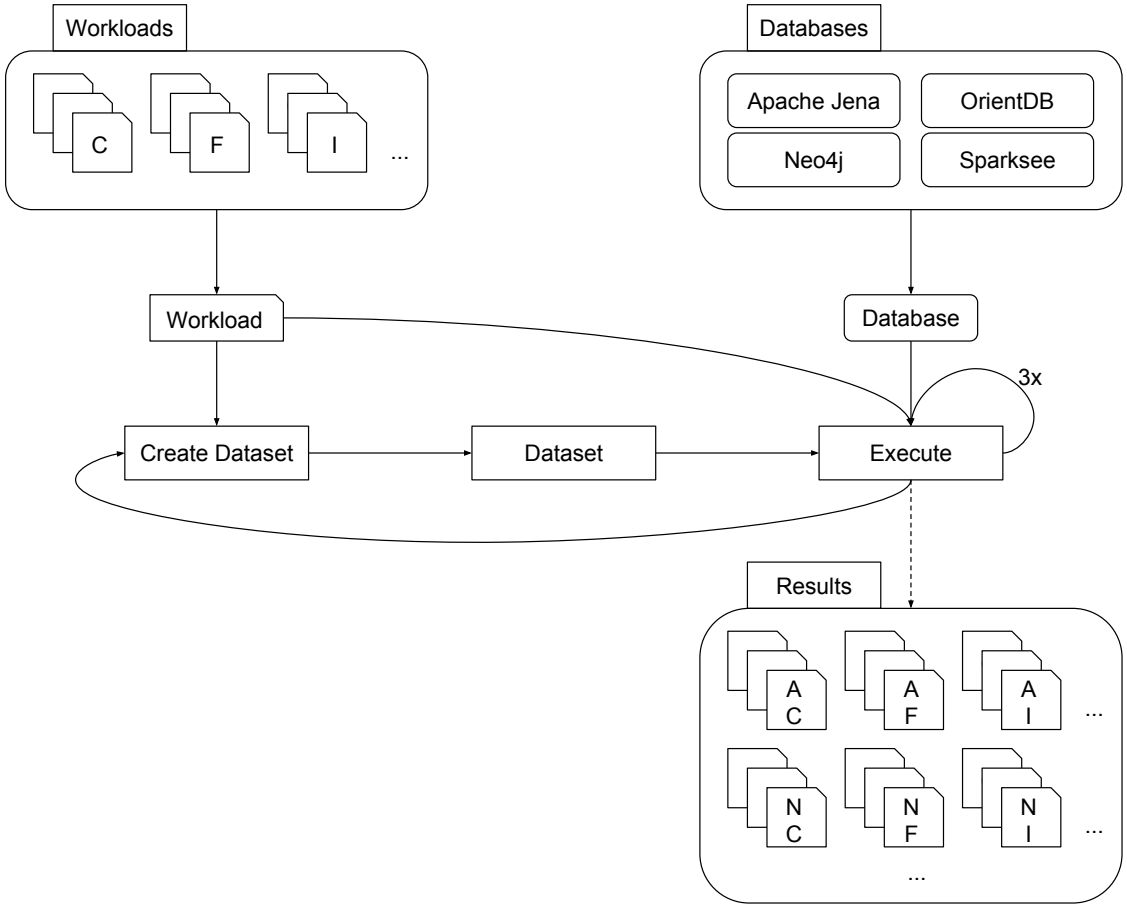

Figure 6.1: Workflow for the execution process.

measurements as shown in tables 6.3 and 6.4.

Finally, we create the diagrams shown in subsections "Results" of the following sections and interpret them to draw a conclusion, which is presented in the "Discussion" subsections. The standard derivation is not shown in the diagrams, as it is around 5% for OrientDB and below 1% for the other databases, which is too small to be seen in the figures.

6.4 Throughput

In this section we will examine the combinations of workloads mentioned in table 6.3. The throughput will show us how well suited the graph databases are in general for applications heavily using insert operations.

Note that in order to insert an edge the start and end node has to be looked up.

6.4.1 Probing Node Count

Here we will compare how the throughput, measured in inserts per second, of the databases is effected by increasing the number of nodes we are inserting into it. We will also look at the execution time, to determine a reasonable large dataset in terms of execution time for the upcoming benchmark runs.

Section	First workload	Other workload(s)	Units of measurement
6.4.1	1. With Index	2.-5. With Index	Inserts/second, total time, database size
6.4.1	1. Without Index	2.-5. Without Index	Inserts/second
6.4.1	n.[1] With Index	n. Without Index	Inserts/second
6.4.2	1. Node Size	2.-5. Node Size	Inserts/second, database size
6.4.3	n. With Index	1. No Edges	Inserts/second

[1]the workload with the largest possible number of nodes in terms of execution time

Table 6.3: Overview for the throughput workloads.

Section	First workload	Other workload(s)	Units of measurement
6.5.1	1. Structure	2.-3. Structure	Inserts/second
6.5.2	x.[2] Suitability	-	Total time
6.6	1. Reading	2. Reading	Reads/second
6.6	1. Scanning	2. Scanning	Scans/second
6.6	1. Structure	1. Reading & 1. Scanning	Operations/second

[2]every workload will be evaluated

Table 6.4: Overview for the production and retrieval workloads.

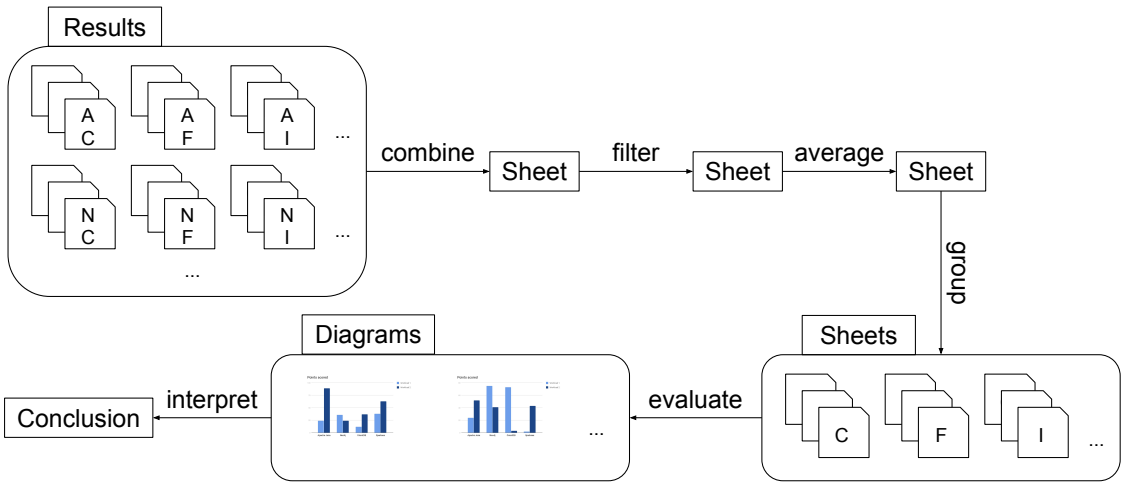

Figure 6.2: Workflow of the evaluation process.

The throughput is listed in inserts per seconds, which include both inserting nodes and inserting edges.

Apache Jena has no option to turn off indexing as mentioned in section 2.3.1.1, but it is still shown in the diagrams as reference.

6.4.1.1 Results

The first figure 6.3 shows how the different databases perform with an increasing dataset size. Apache Jena and Neo4j only have values for 1.000 and 10.000 nodes, because execution with more than 10.000 nodes would take too much time. Sparksee only delivered results up to 100.000 nodes, because the free license only included

database sizes of up to 1.000.000 elements and a workload with 1.000.000 nodes would contain 2.333.333 elements in total with the edges.

In figure 6.4 we see the execution time of the different databases. At 10.000 nodes Apache Jena and Neo4j took almost an hour for one run, because of that we did not run it with 100.000 nodes or more.

Table 6.5, 6.6 and 6.7 show the measured numbers from the figures 6.3, 6.4 and 6.5 respectively.

Database/# Nodes	1000	10000	100000	1000000	10000000
apachejena	8,86	7,21	T	T	T
neo4j	11,50	8,75	T	T	T
orientdb	884,1	2317,07	5672,69	10112,36	8572,34
sparksee	15109,17	17829,1	16425	L	L

Table 6.5: Throughput in inserts/s rounded to two decimal places of the workload using an index for the different dataset sizes. T indicates that too much time would be needed; L indicated license issues.

Database/# Nodes	1000	10000	100000	1000000	10000000
apachejena	263	3238	T	T	T
neo4j	203	2667	T	T	T
orientdb	3,64	10	41	231	2722
sparksee	0,15	1,31	14	L	L

Table 6.6: Execution time in seconds of the workload using an index for the different dataset sizes. T indicates that too much time would be needed; L indicated license issues.

Database/# Nodes	1000	10000	100000	1000000	10000000
apachejena	8,79	7,41	T	T	T
neo4j	12,3	8,83	T	T	T
orientdb	1020,81	2357,49	4616,85	9395,86	8415,18
sparksee	4715,63	635,17	63,37	L	L

Table 6.7: Throughput in inserts/s rounded to two decimal places of the workload using no index for the different dataset sizes. T indicates that too much time would be needed; L indicated license issues.

Figure 6.5 shows the throughput over different dataset sizes without using an index. In figure 6.6 we see a comparison of using an index and not with a dataset size of 10.000 nodes.

6.4.1.2 Discussion

Figure 6.6 shows us, that there is no performance change for Jena, Neo4j and OrientDB in using an index or not. Sparksee shows a significant drop in throughput without the use of an index. That is what we expected, because the throughput also contains insertions of edges, which have to look up nodes, what is faster with

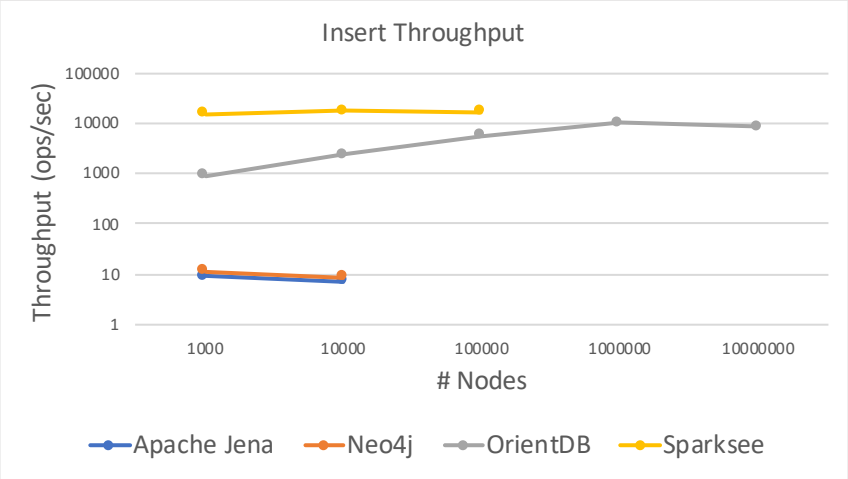

Figure 6.3: This figure shows the throughput in inserts/second of every database over different dataset sizes.

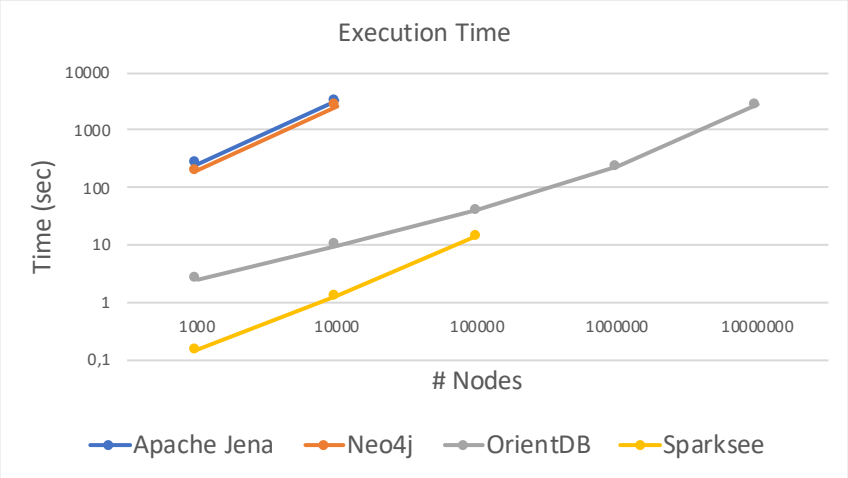

Figure 6.4: The execution time of the databases is shown over different dataset sizes.

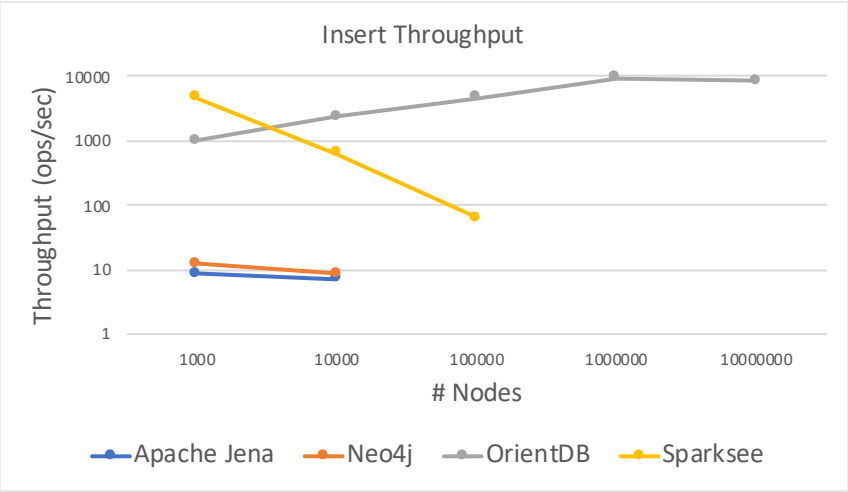

Figure 6.5: This diagram shows the throughput in inserts per second while using no index.

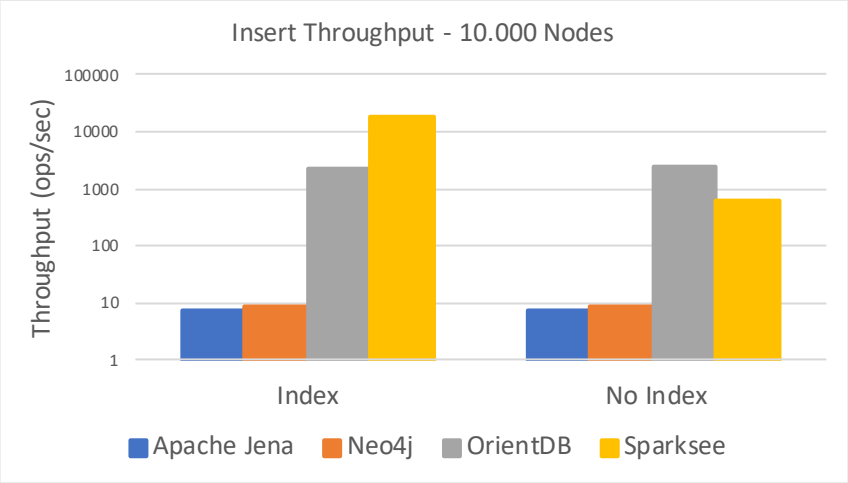

Figure 6.6: The throughput at a fixed dataset size to compare between indexing and not.

an index. For the other databases the lack of difference in performance might by, that the benefit of using an index to retrieve the nodes for an edge is equalised by the time they take to insert the nodes into the index.

The execution time grows linearly, which is a good sign, because that means that the databases do scale for larger amounts of data.

If we compare the archived throughput with our target throughput of $11743,66 \frac{inserts}{s}$, we see that Sparksee exceeds our target with $16435 \frac{inserts}{s}$. OrientDB misses our goal slightly, at the larges dataset it only archived a throughput of $8572 \frac{inserts}{s}$. Jena and Neo4j didn't even reach $10 \frac{inserts}{s}$. These throughput values are measured with another data structure and node size than the one we will use for the suitability workload, so we will investigate the factors differentiating this workload from the suitability workload and reference these results again in section 6.5.2.2.

From these results alone, without looking at read performance separately we can say, that an index is useful, even for insert operations, because edges need to look up two nodes, which is faster when an index is used.

6.4.2 Probing Node Size

In this subsection we will take a look at how the databases perform with different node property sizes. We will pick a dataset size of 10.000 nodes, as all database have a reasonable execution time with that number of nodes.

By investigating the performance under node size variation, we will see if the databases can store larger number of data in one node. That can be useful depending on the use case, in our example given by the industry only a two-digit number is stored, but it could be desirable to store longer number or more complex information.

6.4.2.1 Results

In figure 6.7 we see, how an increasing node size has an impact on insert throughput.

Sparksee only has values for node sizes up to $1KB$, because the property we used to store the value of the node only supports up to 2048 characters or Bytes.

Figure 6.8 shows the size of the database folder, in which the database stores its files.

Table 6.8 and 6.9 show the numbers used for the figures 6.7 and 6.8 respectively.

Node Size (Byte)/ Database	10	100	1000	10000	100000	1000000
apachejena	7,21	9,66	7,83	7,54	6,21	4,92
neo4j	8,75	11,47	8,31	8,59	8,17	4,06
orientdb	2317,07	2521,90	3141,73	1976,84	487,21	14,10
sparksee	17829,10	17155,63	15670,67	x	x	x

Table 6.8: Throughput in inserts/s rounded to two decimal places of the workload using different node sizes. x indicates issues with storing large values.

Node Size (Byte)/ Database	10	100	1000	10000	100000	1000000
apachejena	15	16	25	113	994	9600
neo4j	24	29	49	244	2200	22000
orientdb	75	77	98	172	1300	9500
sparksee	4,7	6	15	x	x	x

Table 6.9: Database sizes of the workload using different node sizes. x indicates issues with storing large values.

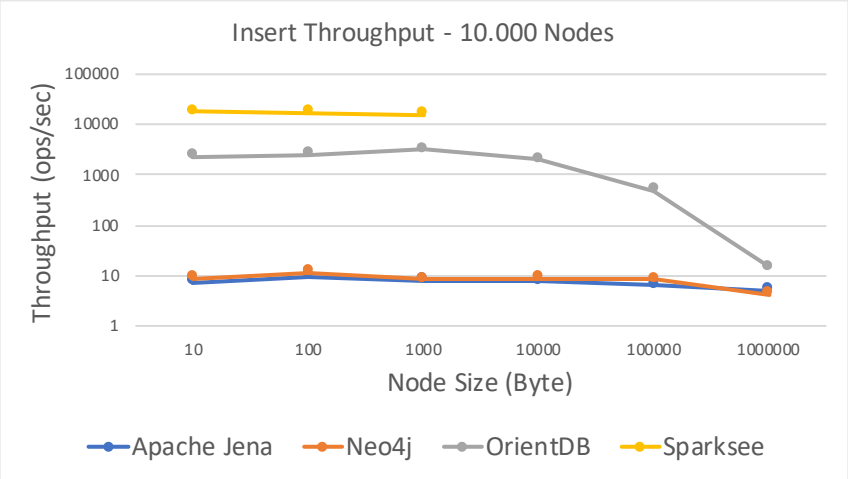

Figure 6.7: Insert throughput over different node sizes with 10.000 nodes total.

6.4.2.2 Discussion

Figure 6.7 show that the throughput of Jena and Neo4j is quite low and it doesn't show much difference with larger node sizes, but at $1MB$ we can see that the performance decreases even more.

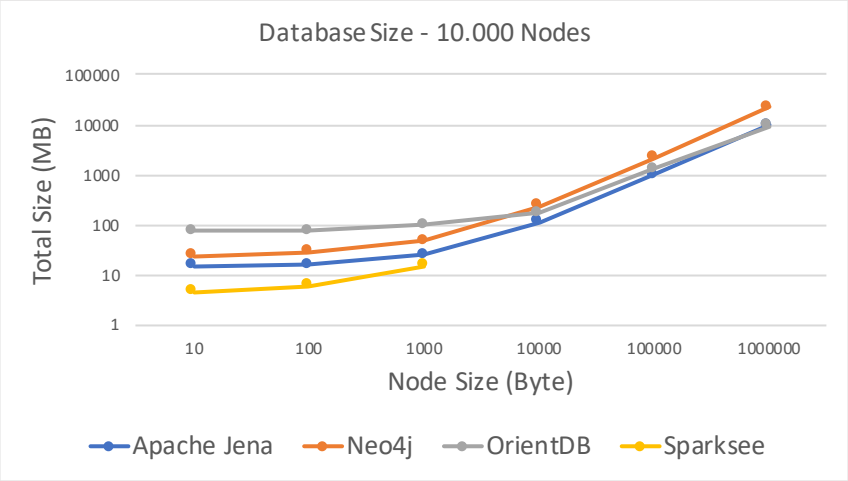

Figure 6.8: The size of the databases over growing node sizes.

For OrientDB we see good performance up to $1KB$, it starts to decline for node sizes of $10KB$ and above with a significant drop at $1MB$.

Sparksee has the highest throughput of all four databases, but as it could only handle sized of up to $2KB$ or $1KB$ in our test scenario. In that range the other databases also show no noteworthy change in performance, so we can't draw a conclusion about the behaviour of Sparksee with larger node sizes.

In figure 6.8 we see that the database size grows linearly with the node size, from $10KB$ and above. So, for smaller node values the overhead of the database itself determines the size of the database.

When we look closely at the values of Neo4j, we can see that they are above the other databases. In fact, at $1MB$ node size, which would result in $10GB$ data for 10.000 nodes, Neo4js database folder had a size of $22GB$, so the overhead is more than the data itself.

6.4.3 Difference without Edges

Here we will investigate how the absents of edges has an impact on performance. These workloads to not represent a real-world scenario, but they will provide us knowledge about how the edge to node ratio effects throughput, as for every edge its start and end node have to be looked up.

6.4.3.1 Results

Table 6.10 and figure 6.9 shows a comparison of all databases between using edges and not with a dataset size of 10.000 nodes.

6.4.3.2 Discussion

Figure 6.9 shows an unexpected behaviour, as the throughput of Neo4j and OrientDB is even higher when using edges whereas ApacheJena and Sparksee shows a decrease in performance. The largest differences can be seen for OrientDB and Sparksee, but as their results point in two different directions we can only say, that it depends on the database.

Database	With Edges	Without Edges
apachejena	7,21	9,95
neo4j	8,75	6,66
orientdb	2317,07	1425,64
sparksee	17829,1	19592,96

Table 6.10: Comparison of throughputs measured in inserts/s and rounded to two decimal places between using edges and not.

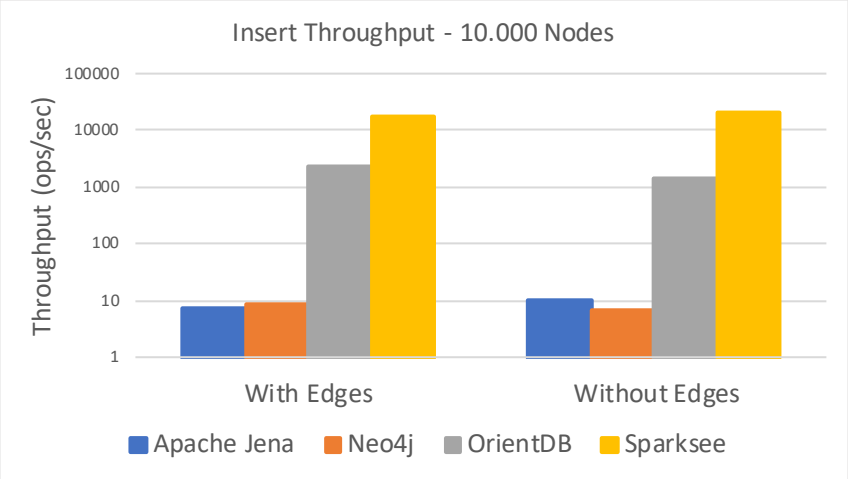

Figure 6.9: Comparison of insert throughput between using edges and not.

6.5 Production Simulation

The workload results presented in this section will cover the production specific variables. The first one being product complexity and the other one execution time.

6.5.1 Product Complexity

The product complexity describes, how much the tree representing our data structure is widened at the three different levels shown in section 4.1.

The wider the data structure becomes the less edges we have per node and also we have more edges coming from one node to other nodes. That can be interesting if we want to investigate on the generalisation of the graph structure to see if other benchmark results from workloads with social network graphs can be used in an industrial application.

6.5.1.1 Results

In figure 6.10 and table 6.11 we see the impact different data structures have on the insert throughput. ”Simple” refers to the structure variables (x, y, z) set to (1, 1, 1), ”More Complex” represents (16, 32, 32) and ”Most Complex” (64, 128, 128).

6.5.1.2 Discussion

As we see in figure 6.10 the structure of the data as we modelled it, doesn't affect the throughput of most databases, except OrientDB shows the largest drop in performance. That fits into our findings from section 6.4.3.2 where OrientDB

Database	Simple	More Complex	Most Complex
apachejena	9,42	8,01	7,58
neo4j	11,44	11,64	8,5
orientdb	2606,5	2330,12	2118,38
sparksee	17345,38	17463,8	17576,06

Table 6.11: Throughput in inserts/s rounded to two decimal places of the workload comparing different structure parameters.

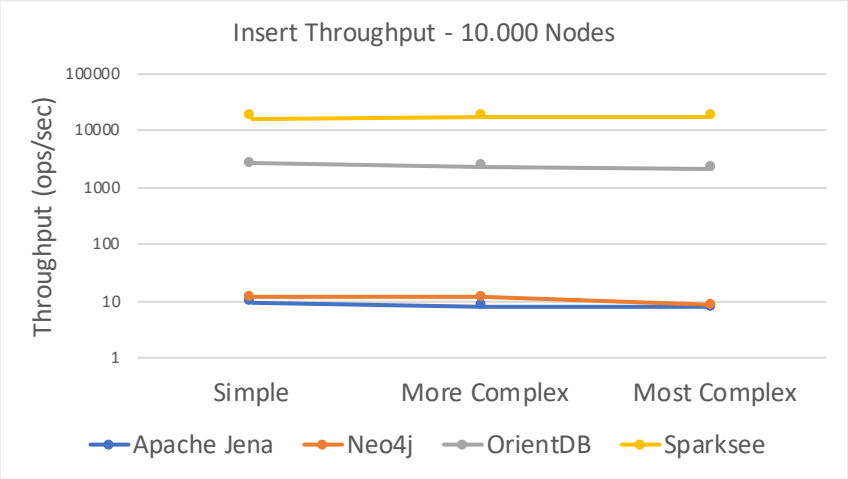

Figure 6.10: Shows the difference in insert throughput over changing data structure.

showed a higher throughput with edges, because the ”Most Complex” workload has a lower edge to node ratio. Jena also seems to suffer from less edges per nodes, what is contradictory to the experiment with no edges, as Jena showed there a better performance without edges.

To draw a conclusion about the comparability with other related work using social network graphs, which have a much higher edge to node ratio, we can at least say, that depending on the database, more edges can benefit the performance.

6.5.2 Production Suitability

The production simulations will finally show, if the databases we chose are capable of storing the necessary amount of data in a specified time interval.

In the discussion of this section we will also investigate the throughput based on the previous workloads.

6.5.2.1 Results

Figure 6.11 and table 6.12 show how long OrientDB took, to store three minutes of production data (1.056.833 nodes). Sparksee is mentioned with a theoretical time, since it only allowed us to store 500.000 elements. We took the throughput during inserting these 500.000 elements and calculated the time it would need to complete the whole workload.

Only OrientDB and Sparksee were used in these workloads, because Apache Jena and Neo4j would take too long to insert that number of nodes.

Database	Time (s)	Throughput (ops/sec)
orientdb	263	8042,84
sparksee	(134)	15789,47

Table 6.12: Time needed to execute three minutes of simulated production. The value in parentheses indicates a theoretical value, which was calculated with the throughput reached until the license ran out of size.

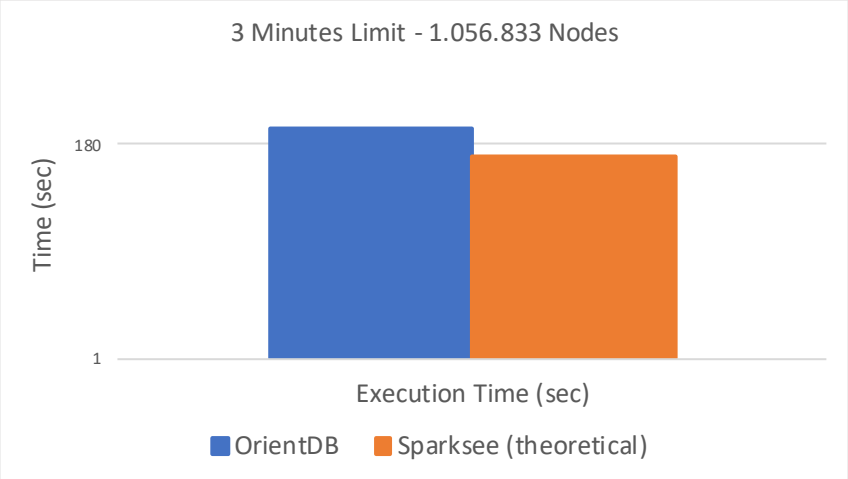

Figure 6.11: Shows the execution time with a dataset that represents three minutes of production.

6.5.2.2 Discussion

We only executed three minutes of production simulation, because the throughput of OrientDB does not grow anymore above 1000000 nodes the one-hour workload would not succeed either.

Figure 6.11 shows us, that OrientDB did not manage to store three minutes of production simulation in the specified time.

Sparksee could theoretically store that amount without exceeding the time limit, but since the free license did not allow for the amount of data we used the average until the limit was reached. Of course, it could be that the throughput of Sparksee drops with an increasing number of elements in the database, but we couldn't investigate that.

The difference of this workload compared to the first workload we discussed in section 6.4.1 is the structure and the node size. The results of 6.4.2 and 6.5.1 show, that the structure has a slight impact on the throughput and the node size has no impact below $10KB$, since we used $50B$ we can compare the first measured throughput.

By doing so we see that Sparksee, again theoretically, exceeds our target throughput of $11743\frac{inserts}{s}$. OrientDB would have a throughput of $101112,36\frac{inserts}{s}$ with a simple structure, but since our "most complex" structure decreases OrientDBs throughput even more to only $8042,84\frac{inserts}{s}$, it would be even less suitable.

6.6 Retrieving under load

This section will cover the results about retrieving data while the database is under load. First, we will take a look at how using an index is effecting the read and scan throughput, then we will compare the throughput of the different operations (6.14) and their impact on the insert operation (6.15).

6.6.1 Results

In figure 6.12, 6.13 and table 6.13, 6.14 we see the throughput of read and scan operations while using an index and not.

Database	Index	No Index
apachejena	47	49,11
neo4j	954,77	795,94
orientdb	8484,44	7816,22
sparksee	12411,84	989,3

Table 6.13: Throughput in reads/s of read operations rounded to two decimal places of the workload using a mix of read and write operations.

Database	Index	No Index
apachejena	43,58	42,56
neo4j	574,17	495,58
orientdb	308,93	319,89
sparksee	174,28	135,41

Table 6.14: Throughput in scans/s of scan operations rounded to two decimal places of the workload using a mix of scan and write operations.

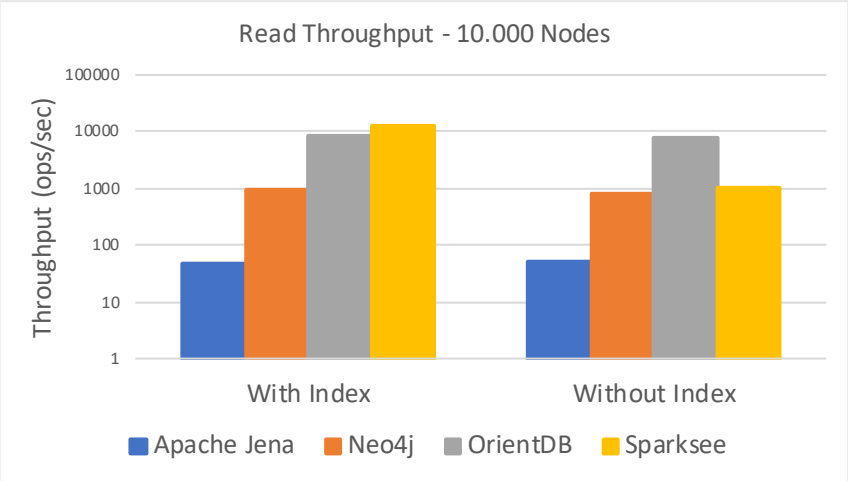

Figure 6.12: Shows the throughput of read operations with and without the use of an index.

Figure 6.14 shows the throughput of the different operations. In figure 6.15 and table 6.15 we see the impact of the read and scan operations on the insert operations.

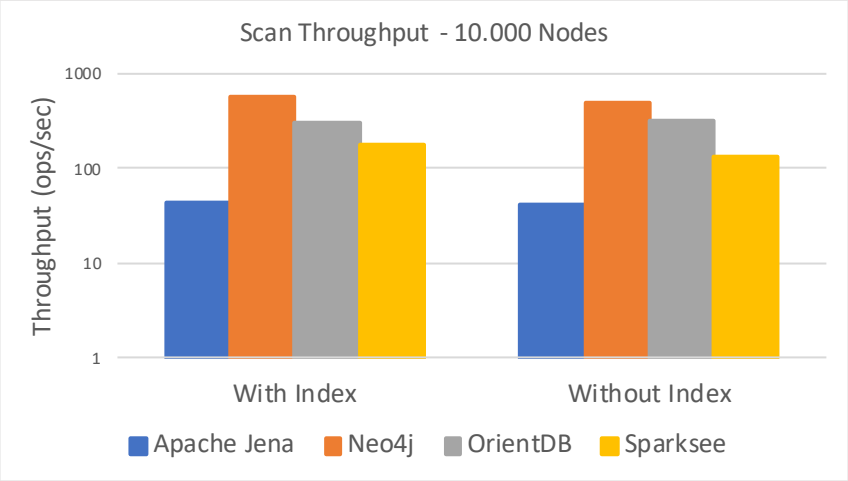

Figure 6.13: Shows the throughput of read operations with and without the use of an index.

Database	Only Insert	With Read	With Scan
apachejena	9,42	6,35	7,15
neo4j	11,44	7,08	7,03
orientdb	2606,5	2180,7	1546,69
sparksee	17345,38	17162,63	17139,49

Table 6.15: Throughput in inserts/s of insert operations rounded to two decimal places of the workloads using only insert, read or scan operations.

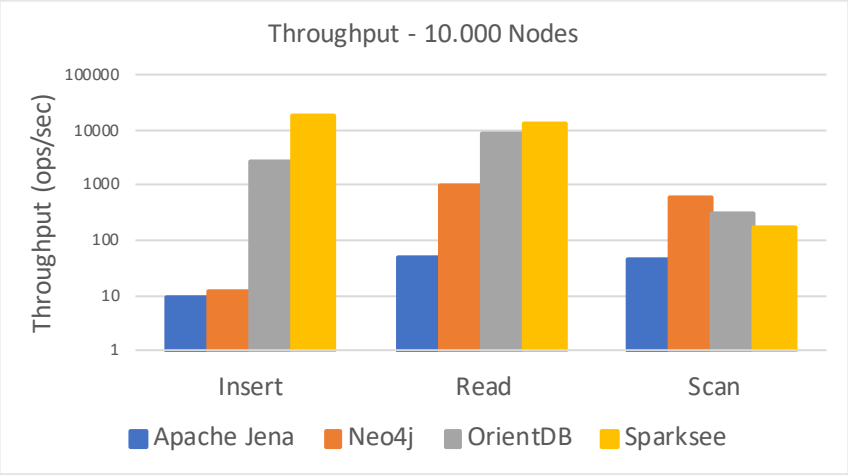

Figure 6.14: Shows the throughput of the different operations.

6.6.2 Discussion

First, we will discuss the results regarding the use of an index and not.

For read operations figure 6.12 shows, that all databases benefit from using an index, except Apache Jena which always uses an index and is only presented as reference. Sparksee shows the biggest difference in throughput of read operations, whereas the others only show a slight decrease in performance without an index. That was what we would expect, since an index really benefits these kinds of operations, although we expected the increase with the use of an index to be higher for Neo4j and OrientDB.

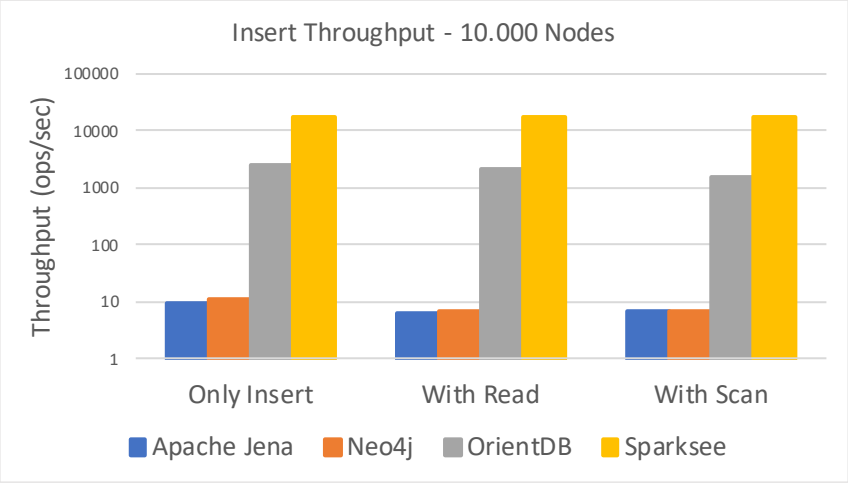

Figure 6.15: Shows the throughput of insert operations when using different operations.

It could be, that the dataset size is too small to show an effect between using an index and not.

Similar results can be seen for the scan operations shown in figure 6.13. The absence of an index has not much effect here either, that could be the case, because scan operations only use one read operations for the start node and then traverse the graph, which is not effected by the index.

The comparison of the different operations shown in figure 6.14 shows us where the strengths and weaknesses are in the different databases. Apache Jena and Neo4j are the slowest when it comes to inserting nodes, but they are much faster in retrieving nodes, with Neo4j even being the fastest of all four in graph traversal.
This also explains why Neo4j and OrientDB have a higher throughput when using edges, because they can look up the edges much faster than they can insert an edge. That also concludes, that the insertion part of creating an edge is cheap compared to inserting a node.

OrientDB and Sparksee seem to be a good choice then inserting and reading is the main concern of the application.

When we compare the results of Apache Jena from its read performance to its scan performance, we see almost no difference in performance, which means it is even faster than Neo4j in graph traversal, but it is limited by the relatively slow read operation at the beginning of the scan operation.

The last figure 6.15 shows us, that using other operations on the database does effect insert throughput, except for Sparksee, which seems to stay stable in its throughput even when other operations are being used.

Jena and Neo4j are low in throughput anyways, but they still suffer from other operations being executed regularly. OrientDB has a slightly worse throughput when using read operations and even worse with scan operations, that is not good for our industry scenario, because in the industry read and scan operations will be used in the database more or less regularly, depending on the specific use case.

6.7 Related Work and Generalisability

In this section we will compare our results with the findings of our related work as it's suitable. Our main goal is to investigate if the difference in structure and edge to node ratio has an impact on performance, when that is not the case we can assume, that benchmark results from research on social network graph can be used to evaluate the performance of a graph database in an industrial environment. The key point we will investigate is how the number of edges in relation to the number of nodes effects the throughput performance of the databases, as for social network graphs this number is quite high at around 8[16, p. 41] to 22[20] whereas our graph structure contains a edge to node ratio of 1.3 to ~ 1 (depending on the variables x, y and z; higher variable values lead to a ratio closer to 1) or 0 if we use the workload without edges.

By comparing our finding with the ones from Dominguez-Sal et al.[16] which can be seen in figure 6.16 we see, that all databases performed much better than in our experiment. The throughput of Neo4j and Jena is well above $100\frac{inserts}{s}$ and Sparksee too reaches a higher throughput with $29770\frac{inserts}{s}$.
Our findings would suggest, that their performance for Sparksee and Jena should be lower than ours and Neo4j would perform better, but since all databases perform better something else has to be different in their case.
This higher performance could be the results from a lack of information stored in the nodes, as the paper only mentions weights on the inserted edges we cannot surely tell if that is the case.

With this comparison we would conclude, that graph databases perform worse in an industrial environment, with graphs that have a lower edge to node ratio compared to a social network graph. Therefore, the results of other benchmarks executed on graph databases with social network graph cannot be used to determine the throughput in an industrial application, as we can't tell how much throughput is sacrificed by storing any information in the nodes. We investigated the impact of different node sizes, but to measure the throughput with absolutely no information in the nodes would require a different implementation that doesn't use the API to store information in the nodes.

The research of Dayarathna et al.[20] used a edge to node ration of 22 with a node count of only 1024. Comparing their results shown in figure 6.17 with our results from figure 6.3 leads to the conclusion, that the databases perform better with an industrial graph structure and less edges.
But by looking closer at the data and our findings from the workload using no edges in section 6.4.3.2 we would expect that the results would be better for OrientDB and Neo4j, since they both perform better with more nodes.
Overall this comparison leads to the conclusion, that graph databases perform better with industrial data structures.

If we look at our own findings in figure 6.9 we can come to the conclusion that depending on the database and moreover the read performance of it the overall throughput with more edges is better, if the read performance is better compared to the insert performance. That would correlate with Dominguez-Sal et al.[16], as they have a higher edge to node ration and better throughput in all cases.

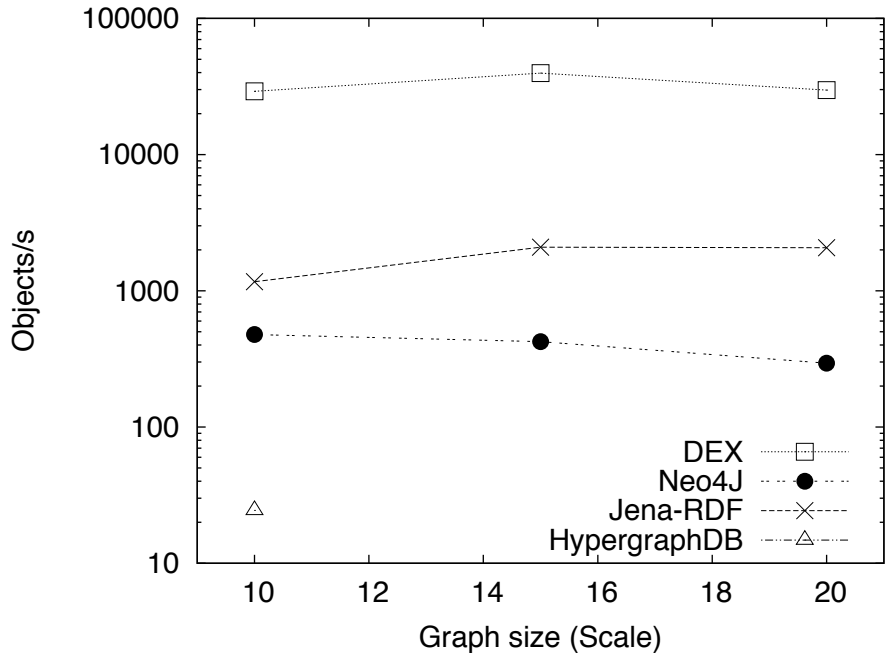

Figure 6.16: Throughput results from Dominguez-Sal et al.[16].

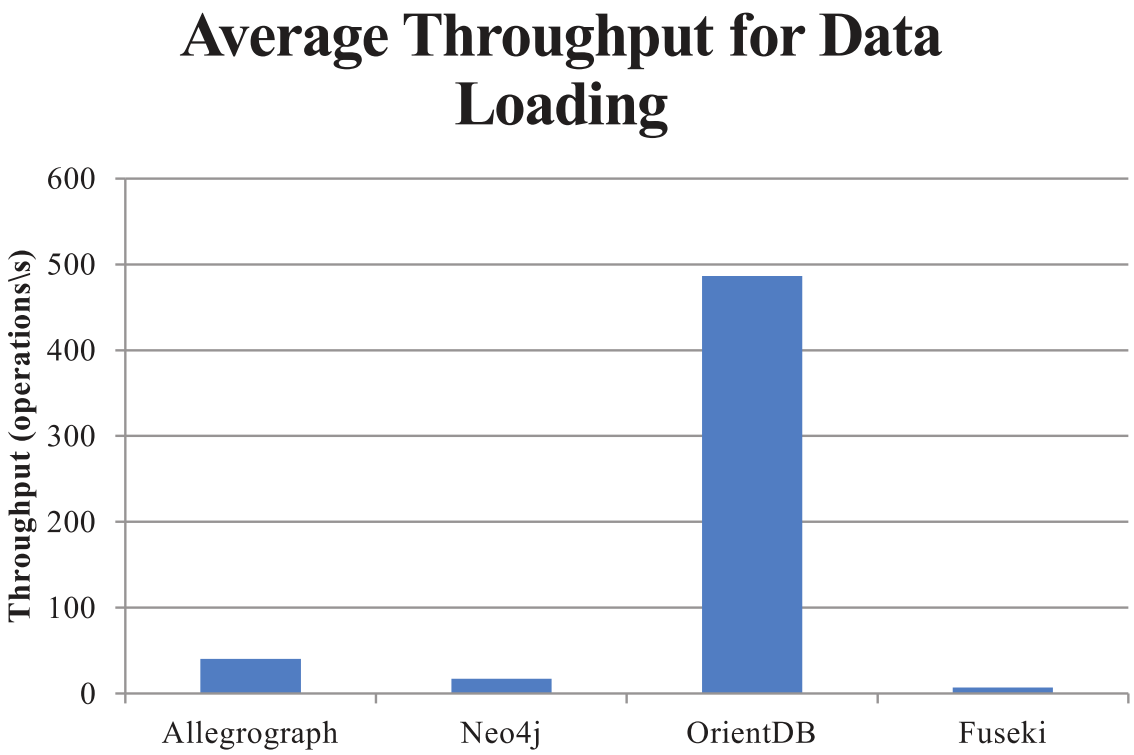

Figure 6.17: Throughput results from the XGDBench Benchmark[20].

To generalise our finding, we can say that the overall throughput for insert operations also depends on the read performance. With that other research examining the performance of graph databases can be used to determine the suitability of a database for the industrial environment if the insert and read throughputs are listed.

7. Conclusion and Future Work

After conducting our experiments and evaluating the results we will finally end with a conclusion and give ideas for future research in this field. At the end we will summarise the research we have done.

7.1 Conclusion

In this section we will draw a conclusion regarding the suitability for the industrial data space and the generalisability of graph benchmarking results measured with social graphs.

7.1.1 Suitability

From our findings we can say, that no database is able to store the necessary amount of data as we dimensioned within the corresponding time frame.

Sparksee could be capable of handling our calculated amount, but we couldn't test it at scale, because of its license limitations.

7.1.2 Generalisability

With our results and the comparison with other research in this field, we can say that the throughput of a graph database depends not only on the insert performance, but among others, also on read throughput as this is needed to insert edges into the database.
It is indirectly possible to transfer the throughput measured on social network graphs to throughput in an industrial application, by taking the read throughput relative to the insert throughput into account and minding the data structure in particular the edge to node ratio.
Besides that, there are other factors effecting throughput as our comparison in section 6.7 shows.

7.2 Future Work

As our investigations on the throughput of graph databases with data from the industrial internet of things could not lead to a solid conclusion about the comparability between performance results measured with social network graphs and industrial graphs, a study should be conducted that investigates the impact of different edge to node ratios covering also different graph properties as the clustering coefficient for example to evaluate which graph properties effect the throughput of graph databases in which way.

7.3 Summary

The purpose of this research was to investigate the suitability of current graph databases for the use in an industrial environment and furthermore examine if the results from previous research conducted in the field of graph database benchmarking can be generalised to be applied on the industrial use case.
To do so available database benchmarks have been looked up alongside with graph databases analysed in other studies. A lack of results for the industrial data space was found and a data structure was created to represent the industrial use case for a graph database and an available benchmark was extended to produce datasets with that structure. Workloads were designed to mirror the use of a graph database in an industrial environment. After executing the workloads with the designed data structure of the graph databases, their throughput under different situations was measured and compared with other studies.

The results show that most current databases are not suitable for use in the industry. Sparksee was the only database able to reach the target throughput for insert operations. OrientDB missed our target only slightly, whereas Apache Jena and Neo4j are far from being able to store that amount of data in the specified time.

From the results no clear conclusion can be made about the generalisation of benchmark results of graph databases, as the result of comparison with other research points in opposite directions. However more arguments suggest, that graph databases perform worse in an industrial application and therefore the results of other studies cannot be applied to determine the performance of a graph database in an industrial environment.

Bibliography

[1] Thomas Worsch. "Grundbegriffe der Informatik". In: November (2011). URL: https://physik.leech.it/pub/Info/Grundbegriffe_der_Informatik/Skripte/WS_10-11_Skript_Worsch.pdf.

[2] SICK AG. *SICK Deutschland | SICK*. URL: https://www.sick.com/de/de/ (visited on 04/30/2018).

[3] Chua Hock-Chuan. *A Quick-Start Tutorial on Relational Database Design Database Design Objectiv e*. URL: https://www.ntu.edu.sg/home/ehchua/programming/sql/Relational_Database_Design.html (visited on 04/28/2018).

[4] Serdar Yegulalp. *What is NoSQL? NoSQL databases explained | InfoWorld*. 2017. URL: https://www.infoworld.com/article/3240644/nosql/what-is-nosql-nosql-databases-explained.html (visited on 04/28/2018).

[5] Margaret Rouse. *What is data collection? - Definition from WhatIs.com*. 2016. URL: https://whatis.techtarget.com/definition/graph-database (visited on 04/28/2018).

[6] Ontotext. *The Truth About Triplestores*. 2014. URL: https://ontotext.com/wp-content/uploads/2014/07/The-Truth-About-Triplestores.pdf.

[7] W3C. *Semantic Web Standards*. 2014. URL: https://www.w3.org/RDF/ (visited on 04/29/2018).

[8] Apache. *Getting Started with Apache Jena*. 2015. URL: https://jena.apache.org/getting_started/ (visited on 04/29/2018).

[9] Apache. *Apache Jena*. URL: http://jena.apache.org/documentation/tdb/architecture.html (visited on 05/13/2018).

[10] Orient Technologies. *Getting Started · OrientDB Manual*. URL: https://orientdb.com/docs/last/Tutorial-Introduction-to-the-NoSQL-world.html (visited on 04/29/2018).

[11] Techopedia. *What is a Data Dictionary? - Definition from Techopedia*. 2017. URL: https://www.techopedia.com/definition/30329/document-oriented-database (visited on 04/29/2018).

[12] Yusuf Abubakar, ThankGod Sani Adeyi, and Ibrahim Gambo Auta. "Performance Evaluation of NoSQL Systems using YCSB in a Resource Austere Environment". In: *Int. J. Appl. Inf. Syst.* 7.8 (2014), pp. 23–27. ISSN: 22490868. DOI: 10.5120/ijais14-451229. URL: https://pdfs.semanticscholar.org/7f04/ab70ba5bc2ba711915a7bb82a346d6a16aaa.pdf.

[13] Neo Technology Inc. *Relational Databases vs. Graph Databases: A Comparison*. 2016. URL: https://neo4j.com/developer/graph-db-vs-rdbms/#_from_relational_to_graph_databases (visited on 04/29/2018).

[14] Neo4j Inc. *The Neo Database - A Technology Introduction*. Tech. rep. 2006, pp. 1–8. URL: http://dist.neo4j.org/neo-technology-introduction.pdf.

[15] Sparsity Technologies. *User Manual - Sparksee*. Tech. rep., p. 147. URL: http://www.sparsity-technologies.com/downloads/UserManual.pdf.

[16] D. Dominguez-Sal et al. "LNCS 6185 - Web-Age Information Management". In: (), pp. 37–48. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-16720-1.pdf#page=53.

[17] Mihai Capotă et al. "Graphalytics". In: *Proc. GRADES'15 - GRADES'15* (2015), pp. 1–6. DOI: 10.1145/2764947.2764954. URL: http://dl.acm.org/citation.cfm?doid=2764947.2764954.

[18] Alexandru Iosup et al. *LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms, a Technical Report*. Tech. rep. Delft University of Technology. URL: http://www.ds.ewi.tudelft.nl/fileadmin/pds/reports/2016/DS-2016-001.pdf.

[19] Yahoo! *Yahoo! Cloud Serving Benchmark (YCSB) Wiki*. 2010. URL: https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/?guccounter=1 (visited on 04/29/2018).

[20] Miyuru Dayarathna and Toyotaro Suzumura. "XGDBench : A Bench m arking Platfor m for Graph Stores in Exascale Clouds". In: (2012), pp. 363–370.

[21] Myunghwan Kim and Jure Leskovec. "Multiplicative Attribute Graph Model of Real-World Networks". In: *Internet Math.* 8.1-2 (2012), pp. 113–160. ISSN: 15427951. DOI: 10.1080/15427951.2012.625257. arXiv: 1009.3499.

[22] Orri Erling et al. "The LDBC Social Network Benchmark". In: *Proc. 2015 ACM SIGMOD Int. Conf. Manag. Data - SIGMOD '15*. 2015, pp. 619–630. ISBN: 9781450327589. DOI: 10.1145/2723372.2742786. URL: https://homepages.cwi.nl/$\sim$duc/papers/snb_sigmod15.pdf.

[23] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. ISSN: 0097-5397. DOI: 10.1137/0201010.