Karlsruhe Institute of Technology

Benchmarking of Graph Databases - Suitability for the Industrial Environment

Bachelor's Thesis
by

Christian Navolskyi

Chair of Pervasive Computing Systems/TECO
Institute of Telematics
Department of Informatics

First Reviewer:	Prof. Dr. Michael Beigl
Second Reviewer:	M.Sc. Andrei Miclaus
Supervisor:	

Project Period: 01/01/2018 – 30.04.2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den TODO: date

Zusammenfassung

TODO: Zusammenfassung (Deutsch)

Abstract

TODO: Zusammenfassung (Englisch)

Contents

1. Introduction

TODO: Fix url in bibliography

1.1 Problem Statement

TODO: Highlight that current graph database benchmark papers are not covering our field. 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.

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 data and reading data.

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 running 24 hours a day the machines are producing a lot of data during the day. That will be the base load for the underlying database, to store all that data from the production machines.

1.1.1.2 Reading Data

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

1.2 Question

This thesis should give an answer to the question, if graph databases are suitable for an industrial application. Suitable in this case means that the database can withstand the amount of data written to it during production. In section 3.1 we analyse how much data could be written to a database and of which structure that data is. We motivate graphs and the use of graph databases in section 2.2 and section 2.3 respectively and will concentrate on those in this thesis, because of the structure of the data from production.

1.3 Methodology

We will chose the databases to use for our testing from other studies covering benchmarking graph databases to be able to compare the results and look to similarities in behaviour. To evaluate different databases we first will look up existing benchmarks and choose the best fitting 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. Next the workloads should be customisable and be able cover our goals. Also the measurements taken during the benchmark need to be useful for us. The databases should be able to connect to the benchmark in the same way.

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 an 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.

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. NOTE: Maybe mention related work

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 out 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 extension 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 which 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 as the software side. Next the procedure of running the benchmarks sequentially is explained following be 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 on 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 out work and give the answer to our question from above. Also ideas for future research and development in this field are presented.

Finally in chapter 8 with give a short summary of our work.

2. Background & Related Work

In this chapter we will give a introduction into 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 [27, p. 89] is a tuple of sets $G = (V, E)$ with $E \subseteq V \times V$. Elements of V are called vertices and elements of E are called edges. The set of vertices has to be not 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 not all are. 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 [27, 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

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

2.1.1 Trees

TODO: Explain tree graphs shortly.

2.2 Industrial Data

Under the term ”industrial data” we understand data that 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.

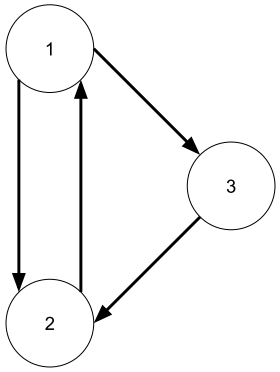

Figure 2.1: A directed graph with three labeled vertices and four edges.

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 [22] as an inspiration for our test data.

Listing 2.1 shows the graph excerpt of our given example.

Listing 2.1: An excerpt showing the observation of components.

```
"@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"
          }
        }
      ]
    }
]
```


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

2.3 Graph Databases

TODO: Write about available versions of the databases There is a variety of database types available 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." ([11])

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. Also they are able to scale horizontally for the cost of consistency. [29]

Graph databases are a type of NoSQL databases. They use graph theory to store their data as described in 2.1 with vertices and edges. Every vertex has a unique identifier (id) in the database, the edges coming from or going to that vertex and it can have properties assigned to it as key/value pairs. Edges also have a unique id, a start and end vertex and properties just as vertices. [21]

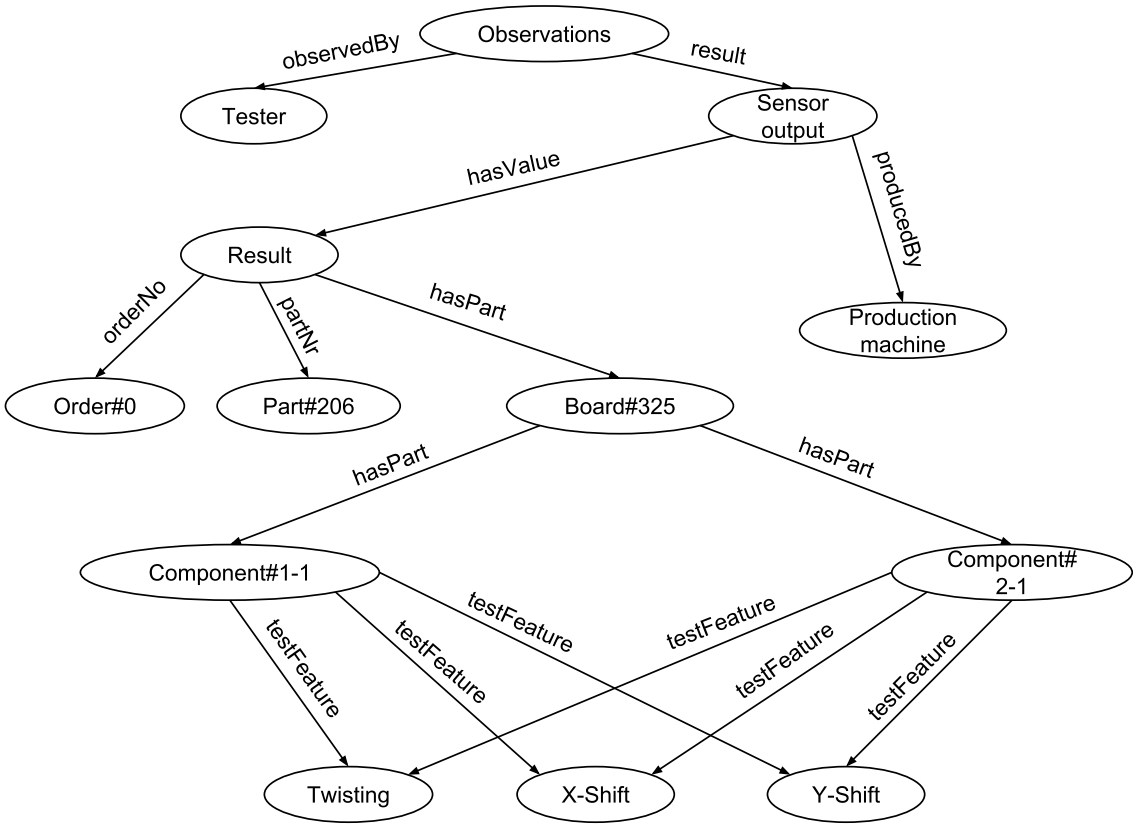

Figure 2.2: An example graph representing the observation of test features in the components of a board.

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 might have their own real world ids these properties can be used to store these real world ids as a key/value pair. Later a particular machine for example can be looked up by its id. That is critical to find the data stored in the database.

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 with transactions to ensure data consistency.

2.3.1 RDF/Triplestores

First in 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 support the evolution of a schema over time. The linking strucutre of the Web is extended by RDF by it using URIs[2] to name relationships and resources connected by those. [19, p. 4]

> This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the

[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. [5]

> graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations. ([26])

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. [19, p. 4]

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." ([4])

The TDB component in Jena is responsible to store and query RDF data. [2]

In section 2.4 we will discuss recent studies investigating Apache Jena, among others.

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 [20]. Its aggregates data in those documents and transforms them internally into an searchable form [25].

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." ([20]) Its designed as a robust, highly scalable database with a wide possible set of use cases. [20]

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. [17]

2.3.3.1 Neo4j

Neo4j is a native graph database and was build as such from the ground up. In their introduction they say "The architecture is designed for optimizing fast management, storage, and traversal of nodes and relationships. In Neo4j, relationships are first class citizens that represent pre-materialized connections between entities." ([18])

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." ([23]) Sparksee is implemented in C++ but provides a low level Java API.

2.4 Graph Database Benchmarks

TODO: Add benchmark results As the need to compare similar programs exists benchmarks are needed to hand results over certain parameters 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 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 [6, p. 2].

Graphalytics uses Datagen to create social network graphs, which are easy to understand for their users [6, 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 [12, p. 7].

Neo4j was used among others in the study of Capotă et al. [6], which we will refer to in our evaluation in chapter 6.

2.4.2 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 the Yahoo! Cloud Serving Benchmark 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” ([8]).

XGDBench also focuses on social networks for their data structure, that is generated by a procedure called Multiplicative Attribute Graph (MAG), see [13] for more information.

It specifically targets read, update and graph traversal operations for its performance aspects [8, p. 366].

This study featured following graph databases which we are also testing Fuseki, Neo4j and OrientDB [8, p. 364]. Fuseki is a SPARQL[3] server providing a HTTP interface to Jena [3], so that research covers three of our four databases.

2.4.3 YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) was not designed specifically for graph databases, but rather for key-value and cloud stores. The project consists of the YCSB client which is responsible for generating the data, as well as the Core

[3]SPARQL is a language to query and manipulate RDF data. [10]

workloads those are a set of workloads executed by the client. The client is extensible to that new workloads, new databases and new generators can be integrated. [28]

The core workload is designed to use simple CRUD[4] operations on any database with no special structure of the generated data.

TODO: maybe add [1]

2.5 Other Related Work TODO: Add if more present/needed

2.5.1 Graph Database: Anna

2.5.2 TODO: Add more

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

3. Analysis

In this chapter we will analyse the data which could occur in an industrial use case, that includes 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 chose 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 us how that data could look like.

Additionally our partners at SICK AG [22] gave us the following key data for a product example. A component carrier with up to 64 circuit boards on it each with up to 128 components is produced every three minutes. Each component has up to 128 test features.

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 structure for our implementation and finally our 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.

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

Over the course of a hour 3.2, day 3.3, week 3.4, month 3.5 and year 3.6 we get the following number of nodes created by the machines.

$$\begin{aligned}
n_{hourTotal} &= n_{total} \times \frac{n_{minutesPerHour}}{n_{timeForSingleComponentCarrier}} \\
\iff &= 1.056.833 \times \frac{60min}{3min} \\
\iff &= 1.056.833 \times 20 \\
\iff &= 21.136.660
\end{aligned} \tag{3.2}$$

$$\begin{aligned}
n_{dayTotal} &= n_{hourTotal} \times n_{hoursPerDay} \\
\iff &= 21.136.660 \times 24 \\
\iff &= 507.279.840
\end{aligned} \tag{3.3}$$

$$\begin{aligned}
n_{weekTotal} &= n_{dayTotal} \times n_{daysPerWeek} \\
\iff &= 507.279.840 \times 7 \\
\iff &= 3.550.958.880
\end{aligned} \tag{3.4}$$

$$\begin{aligned}
n_{monthTotal} &= n_{dayTotal} \times n_{daysPerMonth} \\
\iff &= 507.279.840 \times 30 \\
\iff &= 15.218.395.200
\end{aligned} \tag{3.5}$$

$$\begin{aligned}
n_{yearTotal} &= n_{dayTotal} \times n_{daysPerYear} \\
\iff &= 507.279.840 \times 365 \\
\iff &= 185.157.141.600
\end{aligned} \tag{3.6}$$

In section 4.2.2 we will use these numbers for our 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 our database in an industrial environment, the first one is described in section 3.2.1. Another one is illustrated 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 our workloads for our evaluation, the following subsections should only motivate the use for these 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. This shows, 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 natural use for it. That could be in the form of looking up a certain product produced in the past to get 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 chose a benchmark for our upcoming research we will look at the following aspects of each benchmark.

- Data Structure - Of what structure is the generated data?
- Workloads - How are the workloads designed?
- 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 results of our comparison are shown in the following table 3.1.

Since we are not using a social network structure for our data the graph generators in Graphalytics and XGDBench do not aid us much, as the generators would be difficult to extend because of their use of complex algorithms to create that structure in the created data ([9], [8]). YCSB on the other hand does not serve any particular

Table 3.1: Aspects of the different databases

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

[1] 8 contributors and 16 forks on GitHub [14]
[2] 1 contributor and 1 fork (which is from us) on GitHub [15]
[3] 108 contributors and 1278 forks on GitHub [7]

structure presumably as they are not designed for graph databases and therefore don't need a particular structure on their data. So YCSB should be easy to extend with out data model.

For the workload aspect Graphalytics uses common algorithms which does not represent out workload scenario. XGDBench fits our needs better but not quite well, as inserting data is very important for us. YCSB supports inserting, reading and scanning[4] so it fulfils all our requirements for a benchmark.

Only XGDBench uses another programming language then the other benchmarks namely X10, which could take some time to learn.

Lastly the community aspect in which 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 has an open design[5].

3.4 YCSB

TODO: Reference figure in explanation of aspects, maybe do subsections and move figure up. Go in more detail about the flow, reading inputs, creating which classes executing what over what class TODO: show how values are generated in the generator class 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 operation are performed.

A wrapper class[6] is used to start and end the measurements for each operation executed on the database.

[4] See com.yahoo.ycsb.DB class in the code of [7]
[5] See com.yahoo.ycsb.Workload and com.yahoo.ycsb.generator.Generator in the code of [7]
[6] See com.yahoo.ycsb.DBWrapper in [7]

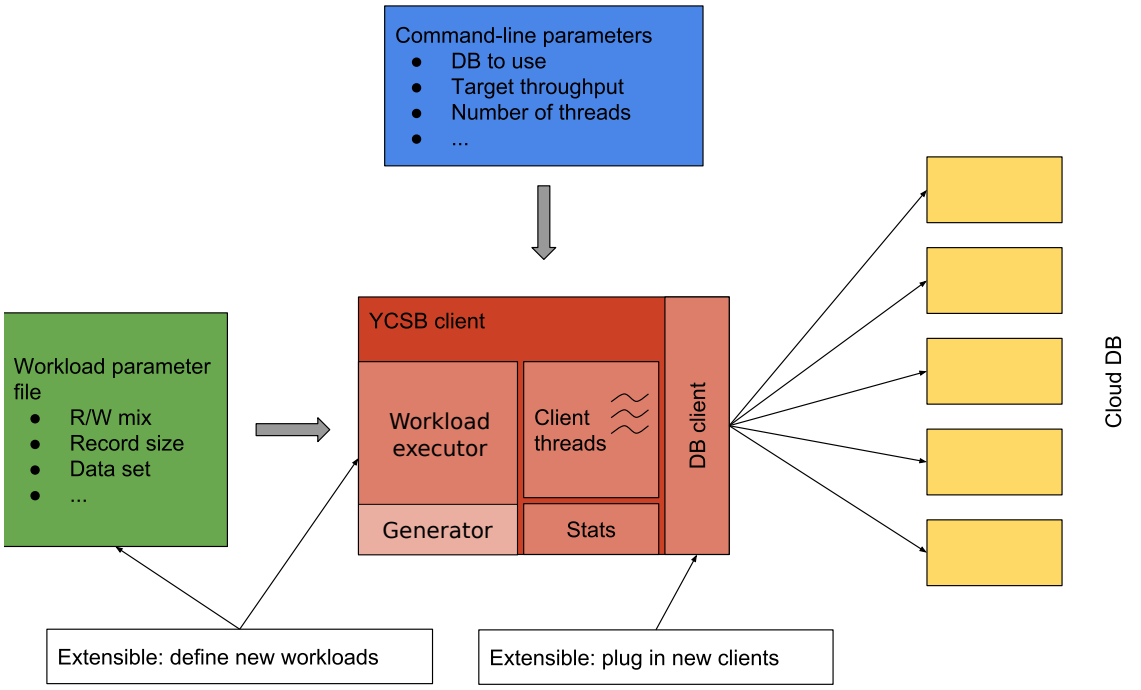

Figure 3.1: The rough architecture of YCSB. Recreated and modified from [1, p. 25]

To start a benchmark run you need to specify certain parameters, such as the database to use and the workload file to use. The client then loads the database, sets up the specified workload class and executes the specified amount and kind of operations on the database.

YCSB uses a workload file to specify some parameters about the workload. These are among others the workload class to use, how much data should be added in the load 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 respectively.

The measurements can be saved as histograms each covering on 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, operations per second and some more meta information.

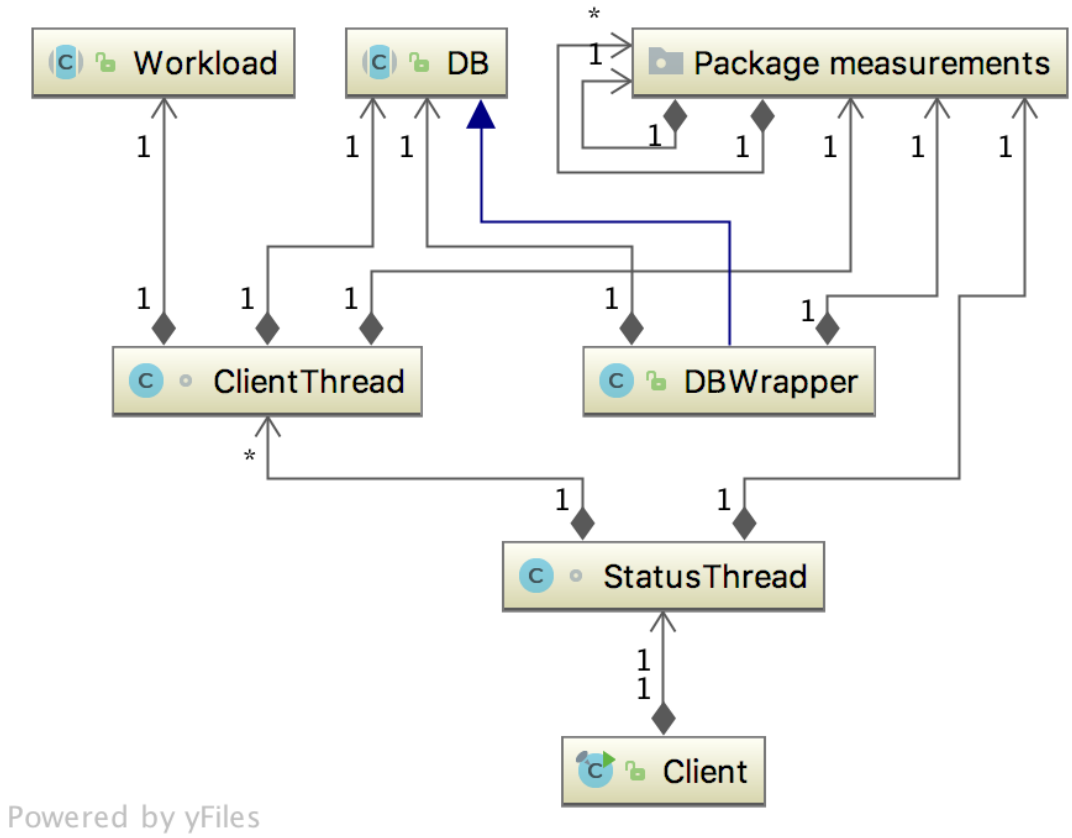

Figure 3.2: Class diagram about the main classes involved in a benchmark run.TODO: reference and explain

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 our examined databases.

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.

In 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 create a schema for our data structure we had a meeting with other researchers at our institute. The result of out 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 some product. At the bottom we see a "M" which stands for "Machine", its connection to "P. Schritte"[1] shows that this machine does one to n production steps. Every production step is associated with a component which consists of a PCB[2] what 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.

TODO: create graphic and describe its content. Graphic should have numbers on its edges, maybe use puml. Naming of parameters

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

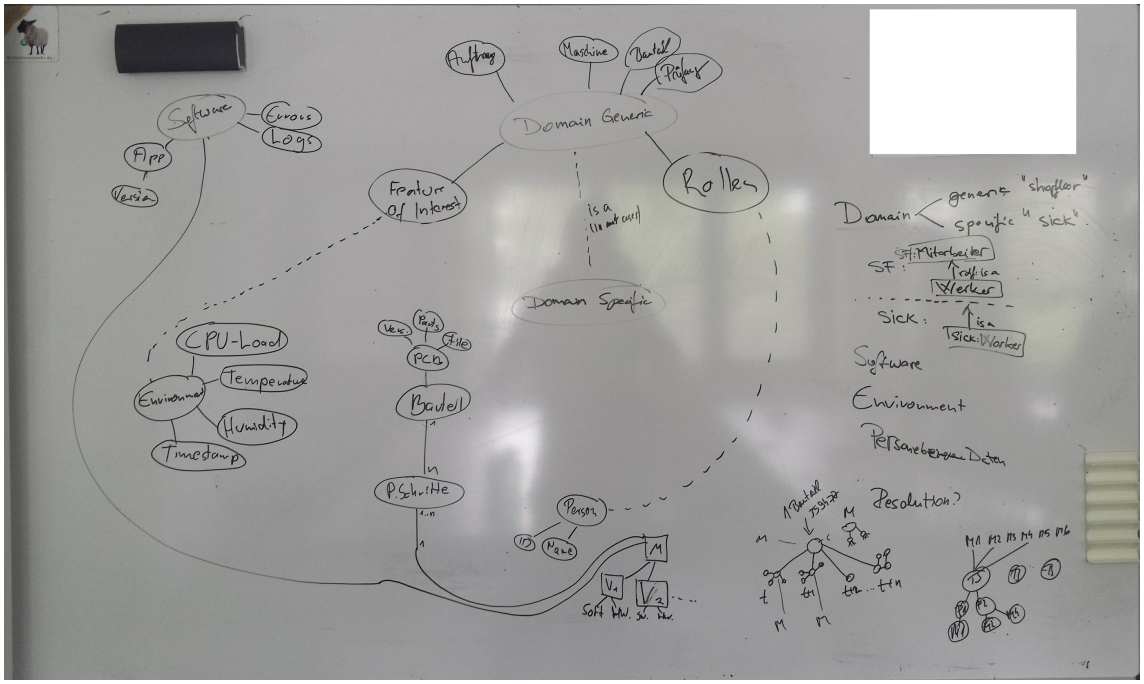

Figure 4.1: The first design of a data schema for industrial data. Created by researchers at our institute.

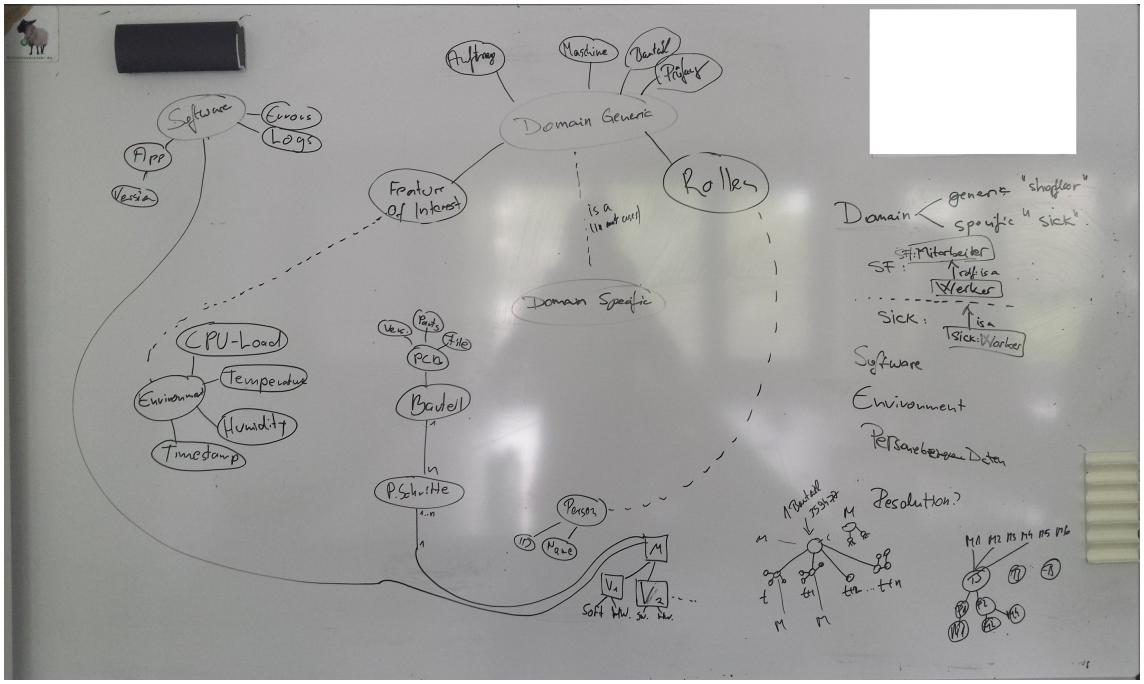

Figure 4.2: The final design of the data schema.TODO: figure with final data structure as implemented

4.2 Workloads

Our workload design will be separated into three part. In subsection 4.2.1 we discuss the design of workloads aimed to uncover the ability to store large amounts of data. Subsection 4.2.2 will directly investigate the suitability of a database to be used in an industrial use case for storing data. We will design workloads to examine the other industrial use case of retrieving 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 have some variables we will change 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 with an index.

The last variable sound counter intuitive, since edges add meaning to the data, but by eliminating them we want so 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 data set 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 there is a linear correlation between the number of nodes and the time needed to store them.

Switching from indexed to not indexed we want to inspect how write speed or operations per second are effected. Indexing is important to retrieve data more quickly for the cost of write speed, with this workload we will see if the sacrifice in write speed is reasonable. 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. NOTE: would have, e.g. temperature of component. Results could be mapped to those.

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.

[3] a node or an edge of the graph

4.2.1.2 Node Property Size

After retrieving a number of nodes that represents a 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).

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 using an index would represent 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

NOTE: maybe remove index, as it has no real purpose. Underline importance of investigating the speed difference without edges. Maybe indexing helps, make sure to compare both with the workloads with edges to see if index helps with edges. In subsection 4.2.1 we already justified why we will investigate the throughput with an exclusive use of nodes in the data set.

As this workload does not allow for much variation besides using an index or not, it will only include these two workloads.

As in subsection 4.2.1.2 we will use a suitable large data set in terms of node count resulting from the first workloads. We will use the same node size as in 4.2.1.1 to be able to compare the results directly to the corresponding ones from that workload.

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 subsection 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 mainly effect the structure of our data. We have three layers which we can blow up horizontally by increasing the corresponding parameters, which are

- NOTE: maybe link to figure if suitable
- 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 if a database is suitable for the industry it should be able to store the data faster then it is coming from the machines. In section 3.1.2 we calculated that 1.056.833 nodes would be written to the database every three minutes. First we will set up a data set with that amount 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.

The next steps would be to repeat that procedure with a node amount that represents a whole hour, day, week, month and year, iff the databases are storing the data fast enough to not take up an unreasonable amounts of time in running the benchmark.

We will use the structure with the maximum width, because it represents the industrial use case best regarding the information given by our partners at SICK AG [22].

4.2.3 Retrieving under load

TODO: mention potential to compare to other studies. There would be no point in storing data if it is not retrieved at some point. To investigate on 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 us 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 can also store properties 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 inserting data.

4.2.3.2 Scanning

Scanning a graph can be done in multiple ways, the simplest being depth first search [24], to retrieve values associated with connected nodes. For example you could scan from a machine to get the test features of their produced products.

As in subsection 4.2.3.1 we will use a mix of 5% scan operations with 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 ”Products per Order”, ”Components per Product” and ”Test Parameter Count” 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 a appropriate data set 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 ”Products per Order”, ”Components per Product” and ”Test Parameter Count” 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
- Generation of rondom graph components
- Generation of a 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 planing to modify.

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
2. No Edges	x	10B	False	True

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

4.3.1 Graph Data Generator

YCSB does not include a graph data generator, therefore we need to create one that fulfils our needs.

The generator should create a data set 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, which are creating together with storing the data and recreating 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 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 the 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 keys.

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

Aspect	Node Count	Products per Order	Components per Product	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 data set 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.

4.3.1.1 Storing the Dataset

TODO: Activity diagram of *GraphDataRecorder* :: *createGraph*() 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 asked 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 parsed in this class and used during subgraph creation.

An activity diagram is shown in figure 4.3 to illustrate the creation process with usage of the parameters for tweaking the width of the graph.

Figure 4.3: Activity diagram of the creation process for the dataset.TODO: create and describe

Figure 4.4: Activity diagram of the *RandomGraphComponentGenerator* showing the process of storing and restoring.TODO:

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

For larger datasets we should read the subgraphs from the file as needed and not at the beginning, because that could fill up the RAM with the dataset and leave less memory for the database to work with.

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 put into the database, that could be done in the *GraphDataGenerator* created for subsection 4.3.1.1, because it anyway touches all created values.

Because we want to retrieve edges and nodes randomly we have to pick one of the two randomly every time a random component 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 to much memory.

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

Figure 4.4 shows an activity diagram of the generator.

4.3.3 Operation Order Generator

To fix the execution order of inserting and retrieving data to and from the graph, we need to store 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] which take a value and a weight and returns distributed according to the weights a value, this can be used to get the operations to run on the database.

Figure 4.5 visualises the procedure of storing the operation order.

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

Figure 4.5: Activity diagram of the operationOrderGenerator.TODO:

4.3.4 Graph Workload

NOTE: Graph Workload functionality TODO: Activity diagram of the workflow with graph data. Create Activity diagram with database from workload view. The *GraphWorkloads* 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 or 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.

In general 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 to insert parsed into a map for compatibility and vice versa for the *result* map and vector. Our data design does not focus to much on the individual properties the 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 get 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* and 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 with 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 amount 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
This takes a random graph component via the *RandomGraphComponentGenerator* and calls the *delete* method of the database.

To avoid calling these methods with edges when the workload specifies to not use them, a parameter which can be set should determine if a random graph component or a 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

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

10. getting the start node of an edge
11. removing a node
12. removing an edge

Generally the *DB* operations can then be implemented using these functions. A rough 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 possible every database should take it as a parameter to set itself up correctly.

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

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

```
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;
  }
}
```

TODO: Extend what information will be given TODO: Also extend over sections to exactly tell what information will be given and adapt subsections accordingly.

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.

Jena has no option to use an index, so we can't use it for the workloads which have the index as their variable, but we still can 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 create resource or property 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 above. 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 $IndexManager$. Over this $Index$ the graph components have to be inserted and retrieved.

Neo4j also uses transactions, but we can not set them as read or write transactions. That is no disadvantage, because it will mark it accordingly after 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, that

is needed to retrieve the nodes easily 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 up.

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 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 the 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.

4.3.5.4 Sparksee

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

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

key
Nodes are created by 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 value can be set as attributes to the graph components, by the attribute and its corresponding value. An attribute 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 /reftab:designOverview.

The workflow of the generators is shown in figure 4.6.

Class	Parameters	Purpose
GraphDataGenerator	folder, ”products per order”, ”components per product”, ”test parameter count”, ”no edges” and ”node property size”	Return subgraphs that form the data structure described in 4.1.
RandomGraph-ComponentGenerator	folder	Return a randomly chose graph component already in the database.
OperationOrderGenerator	folder	Return operations to execute on the database.
GraphWorkload	folder, recordcount and ”no edges”	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 parameters and the of for every class

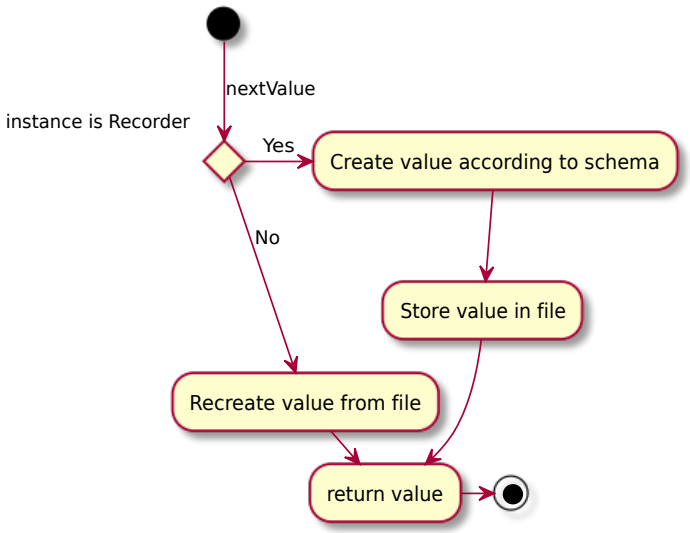

Figure 4.6: Activity diagram roughly showing how the generators will work.

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 the workloads are 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.

5. Implementation

In this chapter we will cover how we implemented the different classes to run 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 can be seen on GitHub [16].

TODO: Class diagram and introduce the upcoming explanation

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 the 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 ”node property size” option.

5.1.2 Edge

As the *Node* class the *Edge* class also uses a counter field to assign the correct identifier to each edge.

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*[1], that extends the generic *Generator* $< V >$[2] 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*[3] to use. 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 have them all in one place.

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*[4], that needs them to know which values it can return.

The *Gson*[5] used in both implementations of this abstract class is initialised here with the *GraphAdapter*[6].

Since there are two phases of the benchmark (see section 3.4) the generator needs to know from which point it should move on with creation, therefore if the current phase is the transaction phase, it will call the 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 chose between a node and an edge and then chose a random graph component of that type. A random node can also be picked directly, that is 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. Besides that it offers the *OperationOrderGenerator* :: *create* method, which observes the present files and initialises the corresponding implementation for the *GraphWorkload*.

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

5.3 Recorder

TODO: diagram to show methods used during nextValue call for the specific use of the classe. For every kind we have a creator that creates the initial values for the workload and stores them in a corresponding file for the recreator 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 4.6 when $GraphDataGenerator :: nextValue$[7] is called to create the next subgraph, the *GraphDataRecorder* is called and creates the subgraph according to the diagram shown in figure 4.3, then serialises it and writes the string coming from serialisation into a file line by line. TODO: maybe deepen the explanation of how the schema is represented in this class.

The serialisation process is done in a *GraphAdapter* that implements both a $JsonSerializer$[8] and a $JsonDeserialzer$[9] with a *Graph* as the generic value. Since a graph object contains two lists, these lists are serialised into a JsonElement, which will be retrieved as a string by calling $Gson :: toJson$.

5.3.2 Random Graph Component

To chose 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. That value will also be stored in a file corresponding to the type of the graph component.

5.3.3 Operation Order

The $OperationOrderRecorder$[10] receives a $DiscreteGenerator$[11], which supplies the string values for the operations. These values will be taken from the *DiscreteGenerator*, 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 data set 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 $Type$[12].

This classes uses a $BufferedReader$[13] to read the file line by line, to avoid extensive memory usage.

[7] com.yahoo.ycsb.generator.graph.GraphDataGenerator
[8] com.google.gson.JsonSerializer
[9] com.google.gson.JsonDeserialzer
[10] com.yahoo.ycsb.generator.operationorder.OperationOrderGenerator
[11] com.yahoo.ycsb.generator.DiscreteGenerator
[12] java.lang.reflect.Type
[13] java.io.BufferedReader

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[14] one for the type and the other two for the identifiers of the different kinds of graph components.

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

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 generator mentioned in section 5.2, by using the *create* method, that way we will have the correct type of generator (recorder or recreator). It also parses the parameters to get to know if "noEdges" should be used, what the "property size" of a node should be, 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. "property size" is stored to be retrievable by the $Node$ to know how much random value 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 is not present and also pass it to the individual generators.

In the load phase the $Client$[15] calls $GraphWorkload :: doInsert$. The $GraphWorkload$ then retrieves a subgraph from the *GraphDataGenerator* by calling $GraphDataGenerator :: nextValue$, separates it into its core 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.

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

doTransactionRead Depending on the "noEdges" option the *RandomGraphComponentGenerator* will be asked for a graph component, if the option is $false$ or 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 *RandomGraphCompone*

[14] java.util.Iterator<E>
[15] com.yahoo.ycsb.Client

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. It also pick a random graph component and calls the $DB :: update$ method of the database. 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 DBWrapper

During implementation we noticed, that the $DBWrapper$[16], which measures the time of each operations on the database, could not distinguish between nodes being inserted or edges. Therefore we additionally adding a measurement that includes the table name in the measurement name, because the database will get the information about the type of the graph component being used by the *table* parameter, we can use it here to measure the nodes and edges separately.

5.5.2 Parameters

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

Our name	Name in the code
folder	datasetdirectory
products per order	productsperorder
components per product	componentsperproduct
test parameter count	testparametercount
recordcount	maxscanlength
no edges	onlynodes
node property size	fieldlength

Table 5.1: This table shows by which name the parameters 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

NOTE: Highlight the mapping of data and other specialities In this section we will describe the different binding implementations and their specialities. Table 5.2 shows the options for the different databases.

[16] com.yahoo.ycsb.DBWrapper

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

Table 5.2: Parameter names of the different databases for the database folder and the index option

5.6.1 Apache Jena

5.6.2 Neo4j

5.6.3 OrientDB

5.6.4 Sparksee

6. Evaluation

6.1 Objective

6.2 Setup

6.2.1 Hardware

6.2.2 Software

6.3 Overview

TODO: table to show which workloads will be compared, for what and where it will be presented.

6.4 Throughput

6.4.1 Probing Node CountNOTE: Comparing indexed to not indexed

6.4.1.1 Results

6.4.1.2 Discussion

6.4.2 Probing Node SizeNOTE: See change over increasing node size

6.4.2.1 Results

6.4.2.2 Discussion

6.4.3 Difference without Edges

6.4.3.1 Results

6.4.3.2 Discussion

6.5 Production Simulation

6.5.1 Product ComplexityNOTE: More child nodes.

6.5.1.1 Results

6.5.1.2 Discussion

6.5.2 Production SuitabilityNOTE: Testing production like workload

6.5.2.1 Results

6.5.2.2 Discussion

6.6 Retrieving under load

6.6.1 Reading under load

6.6.1.1 Results

6.6.1.2 Discussion

6.6.2 Scanning under load

6.6.2.1 Results

6.6.2.2 Discussion

7. Conclusion and Future Work

7.1 Conclusion

7.1.1 Suitability

7.1.2 General Performance of Databases

7.2 Future Work

7.2.1 More Bindings

7.2.2 Concurrency

7.2.3 Other input methodsNOTE: I only used native Java APIs, to directly test the database.

7.2.4 WorkloadsTODO: what kind?

NOTE: Vary RAM amount.

8. Summary

Bibliography

[1] 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.

[2] Apache. *Apache Jena - Apache Jena - TDB*. http://jena.apache.org/documentation/tdb/.

[3] Apache. *Apache Jena - Fuseki serving RDF data over HTTP*. https://jena.apache.org/documen 2016.

[4] Apache. *Getting Started with Apache Jena*. https://jena.apache.org/getting_started/. 2015.

[5] Tim Berners-Lee, R. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Tech. rep. 2005. DOI: 10.17487/rfc3986. arXiv: arXiv:1011.1669v3.

[6] Mihai Capotă et al. "Graphalytics". In: *Proc. GRADES'15 - GRADES'15* (2015), pp. 1–6. DOI: 10.1145/2764947.2764954.

[7] Brian Frank Cooper. *YCSB*. https://github.com/brianfrankcooper/YCSB.

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

[9] 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.

[10] Steve Harris and Andy Seaborne. *SPARQL 1.1 Overview*. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/. 2013. DOI: citeulike-article-id:2620569.

[11] Chua Hock-Chuan. *A Quick-Start Tutorial on Relational Database Design Database Design Objectiv e*. https://www.ntu.edu.sg/home/ehchua/programming/sql/Relation

[12] 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.

[13] 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.

[14] LDBC. *LDBC: Graphalytics*. https://github.com/ldbc/ldbc_graphalytics.

[15] Miyurud. *XGDBench*. https://github.com/miyurud/XGDBench.

[16] Christian Navolskyi. *ChristianNavolskyi/YCSB*. https://github.com/ChristianNavolskyi/YCSE 2017.

[17] Neo Technology Inc. *Relational Databases vs. Graph Databases: A Comparison*. https://neo4j.com/developer/graph-db-vs-rdbms/#_from_relational_to_graph_databases. 2016.

[18] Neo4j Inc. *Chapter 3. Cypher - The Neo4j Developer Manual v3.3*. https://neo4j.com/docs/deve manual/current/introduction/#introduction-highlights.

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

[20] Orient Technologies. *Getting Started · OrientDB Manual*. https://orientdb.com/docs/last/Tuto Introduction-to-the-NoSQL-world.html.

[21] Margaret Rouse. *What is data collection? - Definition from WhatIs.com*. https://whatis.techtarg database. 2016.

[22] SICK AG. *SICK Deutschland | SICK*. https://www.sick.com/de/de/.

[23] Sparsity Technologies. *User Manual - Sparksee*. Tech. rep., p. 147.

[24] 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.

[25] Techopedia. *What is a Data Dictionary? - Definition from Techopedia*. https://www.techopedia. oriented-database. 2017.

[26] W3C. *Semantic Web Standards*. https://www.w3.org/RDF/. 2014.

[27] Thomas Worsch. "Grundbegriffe der Informatik". In: November (2011).

[28] Yahoo! *Yahoo! Cloud Serving Benchmark (YCSB) Wiki*. https://research.yahoo.com/news/yah cloud-serving-benchmark/?guccounter=1. 2010.

[29] Serdar Yegulalp. *What is NoSQL? NoSQL databases explained | InfoWorld*. https://www.infoworld.com/article/3240644/nosql/what-is-nosql-nosql-databases-explained.html. 2017.