Evaluation of Graph Management Systems for Monitoring and Analyzing Social Media Content with OBI4wan

Yordi Verkroost
MSc Computer Science
VU University Amsterdam
y.verkroost@vu.nl

October 8, 2015

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

1 Introduction

Research questions:

- **Architecture**: what is the best platform to work on, both for storing the data and execute the benchmarks using Titan and MonetDB (for example a cluster on Amazon, a server at the CWI, multiple servers from OBI4wan)?
  - What are the differences between the platforms? Advantages? Disadvantages?
  - If not, what can we do to overcome this problem (e.g. write plugins, use workarounds)?
- **Business questions**: Which business questions are interesting to answer for OBI4wan?
- **Main research question**: is a decentralized solution in combination with a dedicated graph database (Titan), graph query language (Gremlin) and text search (ElasticSearch) preferable over a centralized solution (column store MonetDB) in the specific use case of the OBI4wan data set?

The amount of data that is published every day on social media platforms like Twitter and Facebook is becoming larger and larger. People publish updates about their daily lives, companies express themselves to anyone who is interested, and sports clubs keep their supporters up to date with posts about new signings, team training sessions and live match reports. Without any form of organization, this gigantic stream of data would be useless, introducing the need to turn this raw data into useful information. The Dutch company OBI4wan [3] delivers a complete solution in this area, providing tools to monitor, analyze and report on this incoming stream of data, for a big part data originating from Twitter. Tracking social updates about a particular topic (by filtering out posts that contain specific terms) can give insights in how people talk about that topic online. Gaining these insights is useful for companies, giving them the opportunity to see post volumes and -sentiment about their brand and - if needed - start conversations with people who have complaints or comments that need response.

All this data (or in this case: all these tweets) must be stored somewhere, so that it is accessible to use later on. Often a relational database is used, storing tweets row by row. Another solution - which is used by OBI4wan - is to store the tweets in an index like ElasticSearch [4]. In order to store a tweet in an indexing backend, the tweets is first split into single terms. These terms are then stored in *key-value* pairs, where the *key* is the term and the *value* points to the complete tweet object (or tweet objects) in which this term is used. This allows for fast keyword search: enter a single search query term and the indexing backend will return those tweets in which that term was used. While this type of queries - retrieving information based on a single search query - is useful in many cases, it does not cover all data querying possibilities. For example, it is a lot harder to execute queries that retrieve multiple pieces of information, combining them into one single result. Given the tweet data retrieved by OBI4wan, one might for instance want to find all users who have mentioned a single (or multiple) topic(s), are not further than three steps away from some (or multiple) person(s), and have created their account after a certain date (or in a certain period between a start- and end date). Before a query like this returns a single result, separate queries must first be created, executed and their results must be combined. A more optimal solution for this type of queries would be to use a backend system that is capable of executing such a query in one go, traversing over all matching tweets in one single query. As section 2.4 of this paper will show, a graph database backend is a suitable solution for this querying challenge. The question that arises then, however, is which graph database to choose?

The main quest for this paper is to compare database backends that can store tweets in a graph format, and see which one performs the best. Globally there exist two possible database solutions: one using a single-server architecture with a lot of disk storage and memory, and the other using a distributed architecture, spreading the graph data over multiple machines in a cluster that could be expanded at any time. Two database backends that fulfill these requirements are MonetDB (single-server) [10] and Titan (distributed) [2].

The rest of this paper is structured as follows. Section 2 describes the twitter data set retrieved by OBI4wan, how it is stored using ElasticSearch as an indexing backend, and how novel query types require to store the twitter data set in a graph format for a more logical structure and (potentially) better performance. Section 3 contains related work about the topics discussed in this research, including graph networks, graph databases (including Titan), graph programming, graph query languages (including Gremlin, the graph query language used by Titan), other database systems which are not necessarily developed for storing graph networks (including MonetDB) and graph benchmarks that can analyze how well a database backend performs on graph data. Then, sections 4 and 5 give a detailed overview of Titan and MonetDB, respectively. Both sections describe the architecture of these databases, show how data is stored, what kind of query capabilities exist and how to access the databases remotely (which is of importance for executing graph benchmarks). Section 6 talks about the generation of test data and how to transform this data into a format that is accepted by the databases under test. Section 7 details the data format that is used by OBI4wan to store all tweets, and als how this data set must be transformed in order to load it into the databases. Furthermore, this section describes the retrieval of data that was originally missing from the OBI4wan data set, namely information about the friends and followers of the users in the original data set. The original data and the freshly obtained friends- and followers data is then analyzed using a community detection algorithm called SCD. Section 8 shows the setup and execution of the database benchmarks, for which the LDBC benchmark has been used. The LDBC benchmark consists of a collection of benchmark queries, for which a database-specific implementation has been written. This implementation is also detailed in this section. After describing the setup, this section shows how the benchmark has been executed, analyses the obtained results and ultimately shows which database performs the best, given the benchmark results of various test data sizes. Finally, section 9 discusses and concludes this research's findings.

Plan: at this time, the introduction section of the paper is used to sketch the current situation at OBI4wan. It might be better to split the introduction in two sections: one section which is a general introduction into the topic of this master thesis (social networks, graph networks, benchmarks, and finally a [small] introduction of OBI4wan and how they are related to all of this), and another section which dives deeper into what OBI4wan is, what their current situation looks like and how we can use graph networks to improve this situation - or at least offer more opportunities for querying data. This second section is already partly written, see the next section below.

2 OBI4wan: Webcare & Social Media Monitoring

This section is used to describe what OBI4wan is and what they do, first looking at the current situation and then gradually refer to graph networks and how OBI4wan could use those type of networks in order to improve their data querying capabilities. This section should also contain a subsection that clearly describes the contributions of this paper, which is the study of how to use graph networks to enable more interesting queries on the OBI4wan data set, and whether it is best to store this data set distributed in a graph network (Titan) or centralized in a column store database (MonetDB).

The Dutch company OBI4wan offers a tool for online- and offline media monitoring, webcare, data analysis, social analytics and content publishing. The tool gives a complete overview of the millions of messages that are posted on social media and other online- and offline source every day. One the one hand users of OBI4wan can use the webcare module to respond on any of these messages from one single overview, and on the other hand users can create reports and analyze this data. Finally, users can create and plan original content from the publishing module [3].

All messages shown in the tool are retrieved and stored by OBI4wan since 2009. For this research, OBI4wan's Twitter data set is used. The structure of this data is detailed in section 2.1, while section 2.2 shows how ElasticSearch is used as an indexing backend to store the Twitter data. As discussed in the introduction of this paper, OBI4wan could benefit from storing their data in graph format, to enable new query functionality. This vision is described in section 2.4. The process of choosing a suitable graph management system to store the graph data is described in section 2.5, and continues with the comparison between centralized- versus decentralized databases in section 2.6.

2.1 OBI4wan Twitter data set

Describing the OBI4wan data set, focused on Twitter data (tweets).

The Twitter data set of OBI4wan consists of a collection of tweets (primarily from Dutch accounts), posted between 2009 until now. A small subset of this collection (ten days from November 2014) provides some exemplary statistics on the Twitter data set. In the given period, a total of 15.339.524 tweets have been sent by 3.433.224 users, boiling down to approximately 1.5 million tweets per day and 0.45 tweets per user per day. Out of the total number of tweets, 8.389.832 were regular status updates (55% of total and +/- 830K per day), 2.174.581 were replies to other tweets (14% of total and +/- 210K per day) and 4.587.215 were retweets of other tweets (30% of total, +/- 450K per day). The remaining 13.841 tweets in the data set were not classified into any of the aforementioned categories. Figure 1 shows a schema of the Twitter data set as it is stored by OBI4wan, containing the participating objects and their properties, along with the relationships between these objects.

Other interesting statistics can be retrieved about hashtags (*#subject*) and mentions (*@username*), which have been extracted from the tweets. There are 526.603 unique hashtags, where each individual hashtag is used on average 15.0 times. Each tweet contains an average of 0.52 hashtags and an average of 0.76 mentions (both real, original mentions and mentions in retweets). The data set also allows to calculate statistics about the inter-connectivity between users. Comparing the total number of mentioned users (grouped by unique users) to the total number of mentions shows that users who are mentioned at least once, are being mentioned an average 6.8 times during the ten day time frame. Furthermore, users mention other users in their tweets an average 3.4 times during the same time frame.

An important statistic is the arrival rate of new tweets. Because the OBI4wan solution provides real-time access to up-to-date data, it is important to be able to cope with incoming

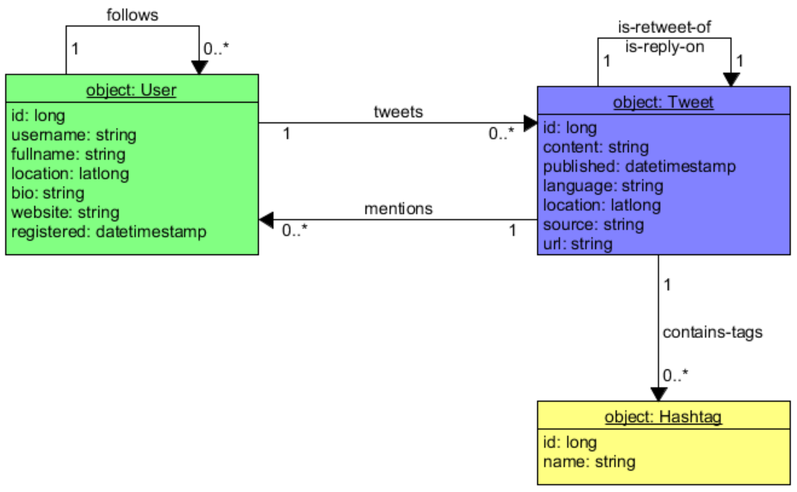


Figure 1: Schema of the OBI4wan data set of Twitter messages.

data continuously. As mentioned before, the discussed data set contains about 1.5 million tweets per day (1.533.952,4 exactly), which means 63.914,68 tweets per hour, 1.065,24 tweets per minute and 17.75 tweets coming in per second. Ideally, these tweets must be processed immediately or at least as quickly as possible, to minimize the amount of buffering of incoming tweets.

2.1.1 Friends and Followers

All unique Twitter users have been retrieved from the ten day subset (from November 2014) of the complete OBI4wan data set. These users have been ranked based on their activity on Twitter (measured in the number of posts per user during this ten day period), and the one million most active users have been taken out as another subset. For these users, their friends and followers (maximum of 5000 for both, corresponding to the maximum set returned by a single Twitter API call) have been retrieved over a period of a couple of months using the Twitter API, meaning that the content of these two sets may have changed at the time of reading this. An analysis of the users and their friends and followers has resulted in some statistics about the twitter graph network of OBI4wan, as shown in Figure 2. Before looking at these figures, a definition of the exact analysis has to be given. In words, the following statistics are calculated:

- The set of unique followers that have been seen in the data set up until a certain point. For example, after iterating over one user (say, user A), A's followers have been retrieved and are all unique. When iterating over the next user (say, user B), B could have some followers which were already found through user A. These followers are not stored in the unique followers set.

- The set of unique friends that have been seen up until a certain point. Here, the same reasoning applies as for the set described in the previous point, but now for friends.

- The set of followers that have already been seen as a user up until a certain point. When iterating over the set of users, the set of users seen keeps growing. Some of these users might also exist in the set of unique followers; this set keeps track of those users.

- The set of friends that have already been seen as a user up until a certain point. Here, the same reasoning applies as for the set described in the previous point, but now for friends.

Based on the definitions of these sets, the following logic applies. Set U is the set of unique users obtained from the OBI4wan data set. $u \in U$ is a single user in U. Then:

- $TW = \{(u, txt) |$ tweets from the OBI4wan data set$\}$.
- $OBI = \{u | \exists (u, txt) \in TW\}$
- $OTOPx = \{u | |TW| \geq CNTx\}$

In words: TW is a single tweet from the OBI4wan data set, consisting of the user u who created the tweet, and txt containing the tweet's content. OBI is the (unique) set of users from the OBI4wan data set, where for each user in this set there exists at least one tweet in the OBI4wan data set. $OTOPx$ is the set of most active users from the unique OBI set, defined by all users who have tweeted more than some threshold $CNTx$. The value of this threshold is set so that $OTOPx$ contains one million users. Then:

- $u.Fo$ is the set of users following u.
- $u.Fr$ is the set of users who are u's friends (e.g. the users u follows).

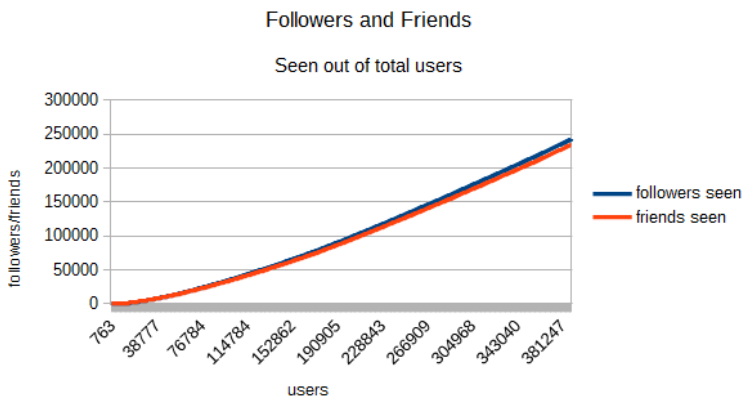
2.2 ElasticSearch architecture

Describing how OBI4wan's data set is stored and indexed, using ElasticSearch.

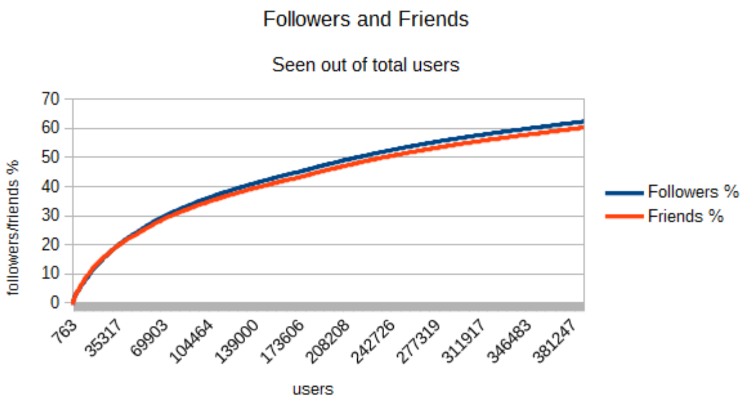
To store all the incoming Twitter data, OBI4wan makes use of an ElasticSearch indexing backend [4]. ElasticSearch is an open-source search engine that has been built as an extension to the full search-engine library Apache Lucene [7]. ElasticSearch provides a simple API to perform searches on indexed data, and also allows for the distribution of this data across multiple nodes. Figure 3 shows a schematic of an ElasticSearch cluster, containing three nodes and three shards per node. A *shard* is a part of the index that holds a fragment of the indexed data (one index points to one or more shards). The figure shows that this example cluster contains a total of three primary shards ($P0...P2$), which together hold all the indexed data for an application. The remaining shards ($R0...R2$, two of each) are replicas of the primary shards, serving as fallback in case one of the nodes holding an active, primary shard dies. One of the nodes acts as the *master node*, handling all cluster-related operations such as adding a new node, removing a node, creating a new index, etc. A cluster can be expanded (scaled out horizontally) with more nodes and shards if an increase in data volume necessitates this.

The current OBI4wan data set is stored using the just mentioned ElasticSearch indices, distributed over multiple nodes. The complete data set is partitioned into multiple indices, each one containing data from a time period of ten days. To have an easy reference to all data that has been collected during one month, aliases are created that point to three indices. For example, the alias *nov2015* points to the three ten-day time frames from November 2014. Figure 4 shows what an ElasticSearch cluster at OBI4wan looks like. The layout is essentially the same as in Figure 3. The shards $P0...P2$ are the primary shards representing three ten-day time frames, pointed to by a month-alias.

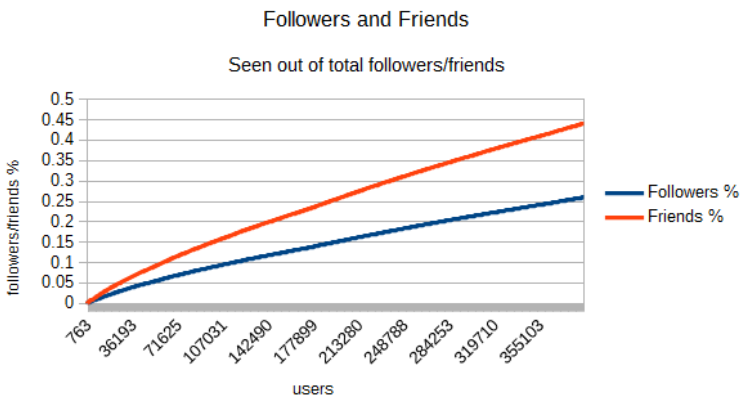
The data of a tweet inside a shard is stored in JSON-format. An example of a tweet in this format is shown below. Most of the property should speak for themselves. The **loc** is the location from where this tweet was posted, the **inreplytoid** contains the tweet ID to which this tweet is a reply or a retweet, or -1 if this tweet is an original tweet, and *posttype* contains the type of this tweet (either *status*, *reply* or *retweet*).



(a) Followers and friends seen out of total users, in numbers



(b) Followers and friends seen out of total users, in percentage



(c) Followers and friends seen out of total followers and friends

Figure 2: Twitter statistics

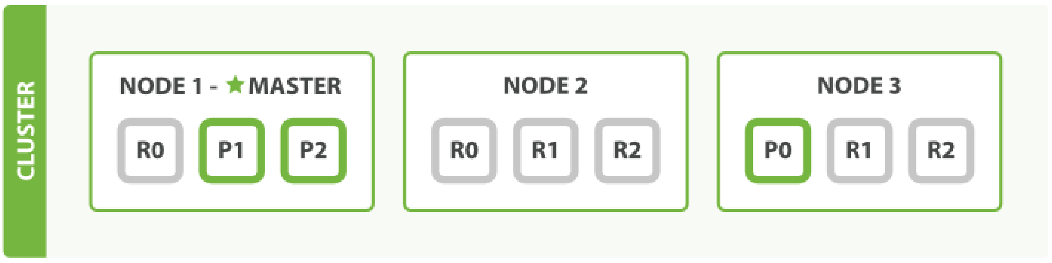


Figure 3: An ElasticSearch cluster with three nodes and three shards per node.

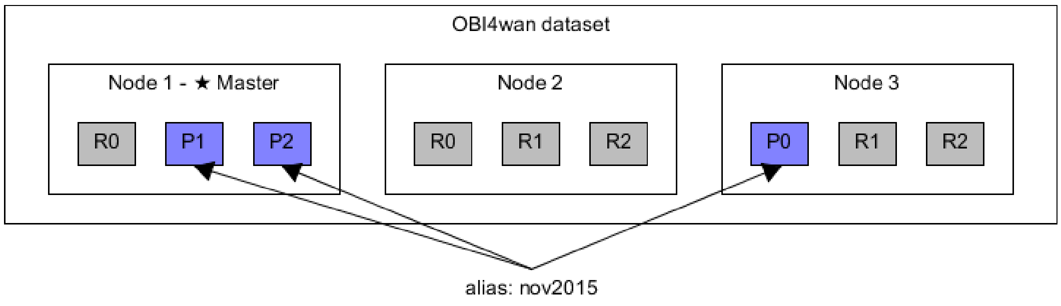


Figure 4: An ElasticSearch cluster as it is used by OBI4wan.

```
{"id":"535648671644536832", "user":"kimber33", "title":"",
"content":["@ter808 so I'm catching up on \#TheVoice \& someones singing I wanna
dance w somebody, to get saved by America... Clearly my vote goes to her"],
"published":"2014-11-21T04:19:29.000Z", ..., "language":"EN", ..., "friends":295,
"followers":45, "loc":"Rhode Island", "source":"Twitter for iPhone", ...,
"hashtags":["thevoice"], "mentions":["ter808"], "inreplytoid":"-1", ...,
"posttype": "STATUS", "url":"http://twitter.com/kimber33/status/535648671644536832/"}
```

On top of the ElasticSearch cluster, OBI4wan has build a layer that acts as the entry point for all data that enters the system from social platforms. This stream of data has to be distributed over the cluster, preferably in such a way that related data is stored on the same node. This is an advantage when queries are executed on the data: if a query only has to retrieve data from one single node, the need to communicate with other nodes (introducing communication overhead) disappears. However, finding subsets of related data that can be stored on the same node is hard, because of the many relationships that exist between social data. For example, creating subsets of data based on timeframes (all data posted between two moments in time) is not always the most optimal choice, because one published message could be a reply to another message that was posted in a different timeframe and therefore belongs to a different subset (and is placed on a different node). Another possibility is to create subsets based on clusters in a social network, consisting of people (and their published messages) who often interact with each other. The downside of this type of subset creation is that such clusters can differ in size, and there might still be interactions between people from different clusters, inevitably resulting in network communication and the subsequent communication overhead.

Summarizing, partitioning social data is not easy and the best way to create partitions heavily depends on the structure and type of the data set. Finding the optimal way to create partitions of social network data could be the subject of a separate research project.

2.3 Novel questions for OBI4wan's data set

The next section (*In need of graph query functionality*) argues for graph query functionality, and presents an example query that could benefit from using graphs. However, it is more interesting to know which type of novel questions OBI4wan would like to ask to its data set. This section could introduce these questions, describing each one and showing their relevance. After this description, it is known what OBI4wan would like to know, and then it is more clear why graph query functionality is needed - or at least works better than traditional databases - in order to answer these questions. These OBI4wan-questions are later translated into real benchmark queries for both Titan and MonetDB.

2.4 In need of graph query functionality

Introducing the concept of graph networks and how OBI4wan could benefit from storing its data in a graph. This is one of the sections that should contain a paragraph that shows the contributions of this paper (e.g. how to use graph networks to enhance OBI4wan's querying capabilities).

With ElasticSearch, it is possible to execute search queries on structured document collections like the one from OBI4wan that has been described in section 2.1. An example query that could be executed using ElasticSearch is the following:

```
GET /my_index/my_type/_search
{
    "query": {
        "match": {
            "content": "Coca Cola"
        }
    }
}
```

This full-text search query retrieves all documents that contain the term "Coca Cola" in their *content* field. The retrieved documents are ranked based on the frequency of the term inside the *content* field, taking into account the average number of occurrences in the *content* field over all documents in the document set. Furthermore, content sections with relatively few words containing the query term are ranked higher, because a larger portion of the section is represented by the term, indicating the term could be important in the scope of that document [6].

The same kind of syntax can be used to retrieve documents containing certain terms in other field types, for example all documents that are published by a specific author, or all documents that were published during a given time frame. In addition to that - in the use case of OBI4wan - all of these queries need to be executed and return a result in (near) real-time, thereby minimizing the waiting time for the customers executing the queries.

The full-text search capabilities of ElasticSearch are useful when trying to retrieve documents containing certain terms in field types, but queries that require to retrieve multiple pieces of information, which are afterwards combined to produce a final result, are much harder to execute using ElasticSearch. For example, consider the following query:

Return all users from Twitter that mention "Coca Cola" in one of their tweets (posted in a specific time frame) and are no more than three relational steps away from the official Coca Cola Twitter account.

This query requires to retrieve multiple pieces of information, namely (1) all users that have mentioned *Coca Cola* on Twitter (in the specific time frame), and (2) all users who are no more than three relational steps away from the official *Coca Cola* Twitter account. This type of query cannot be executed with ElasticSearch directly, but only by going over multiple steps:

1. Retrieve all users that have mentioned Coca Cola in one of their tweets. This query can be executed using the text-search capabilities of ElasticSearch, searching for the term "coca cola" in the content field of a tweet. Save this collection of users in a variable, say W.
2. Retrieve the users that are related to the Coca Cola account (on Twitter defined by a 'follows' relationship). Save this set of users in another variable, say X.
3. Retrieve all users that are related to one of the users in X and are not already in X; call this new set of users Y. These are the users who are two relational steps away from the official Coca Cola Twitter account. Note that this query has to be executed for *every* user in X.
4. Execute the previous query once more, to retrieve all users who are three relational steps away from the official Coca Cola Twitter account. Call this resulting set of users Z. Again note that this query has to be executed for *every* user in Y.
5. Finally, take the intersection of W and Z, returning all users who have mentioned "coca cola" and are three relational steps away from the official Coca Cola Twitter account.

In ElasticSearch, retrieving the set of all users that follow a specific user (for example everyone who follows the official Coca Cola Twitter account) could look like this:

```
"query" : {
  "filtered" : {
    "filter" : {
      "terms" : {
        "user" : {
          "index" : "users",
          "type" : "user",
          "name" : "cocacola",
          "path" : "followers"
        },
        "_cache_key" : "user_cocacola_friends"
      }
    }
  }
}
```

The above query is optimized by caching the results (the follower set of the official Coca Cola Twitter account) using a unique cache key (*user_cocacola_friends*). This prevents retrieving the set of followers of the same account multiple times, resulting in more efficient behavior. The same type of query has to be executed for every user of which the set of followers has to be retrieved based on the set of steps defined earlier. For example, when retrieving the followers set for the official OBI4wan Twitter account, the *name* field becomes *OBI4wan* and the cache key changes into something like *user_obi4wan_friends*.

The problem of retrieving the final user set based on the "Coca Cola"-query is the high number of queries that have to be executed. This number is so high because the queries that retrieve relationships have to be executed for *every* user in the intermediate user sets. Furthermore, these intermediate user sets are always send back to the client, who in turn sends user names from these intermediate sets back to the server for subsequent look-ups of this user's followers. In other words, all coordination of the necessary steps to retrieve the final user set (saving intermediate results, sending back new users for which follower sets must be retrieved, taking the intersection of user sets, ensuring no duplicates exist, etc.) has to be performed on the client-side. The combination of all these factors results in a relatively complex system and slow response time before a final result is returned.

To overcome this problem, the data set of OBI4wan has to be stored in another, more matching format, that allows to execute queries like the one above more efficiently. In recent years, a new class of database systems have emerged that allow to store data in this format and execute queries on this data: graph databases. A graph (data store) consists of a collection of heterogeneous objects with properties (for example *tweets* and *people*), connected to each other by a certain relationship (for example *favorite*, *reply*, *retweet* or *follow*). Examples of graph data stores that have been in development over the past few years are Neo4j [16], Sparksee (DEX) [18] and Titan [2].

Figure 5 shows an example of a Twitter data set in graph format. The real data in the OBI4wan data set looks similar to this example. The graph contains three object types, namely users (accounts on Twitter, represented by the green circles), tweets (blue circles) and hashtags (used in tweets, yellow circles). Users can follow each other and post tweets, and tweets can contain mentions of users. Tweets can be retweets (reposts) of other tweets, or replies on on other tweets.

From the database systems that have been mentioned earlier, Titan [2] is the one that can store data in a graph format, execute queries on it *and* provides the possibility to store data on multiple nodes in a distributed cluster. Titan is a cluster-based graph database (allowing to distribute data over multiple nodes), has an accompanying graph query language (Gremlin [12], from the TinkerPop stack [11]) and provides support for geo-, numeric range-, and full-text search via an ElasticSearch plugin. The integration of ElasticSearch is an advantage of Titan in the use case of OBI4wan, because the current solution of OBI4wan is built on and uses the full-text search queries of ElasticSearch. However, there is also a possible downside to Titan, that has its roots in the ongoing debate about *shared memory* versus *shared nothing* systems and the resultant way in which multiple processes (nodes) in the same system communicate with each other.

Shared memory versus shared nothing In shared memory systems, nodes can communicate and share data with each other using a shared memory space that is accessible by all nodes in the system. In a shared nothing system, this central memory space is absent, forcing nodes to communicate and share data with each other by exchanging messages. This form of communication can become a problem when a query needs to fetch data from multiple nodes, forcing these nodes to communicate and share data between each other over the network. This is especially true for social graph databases like Twitter, because it is hard to partition a social graph database in such a way that the communication and sharing of data between nodes is kept to a minimum.

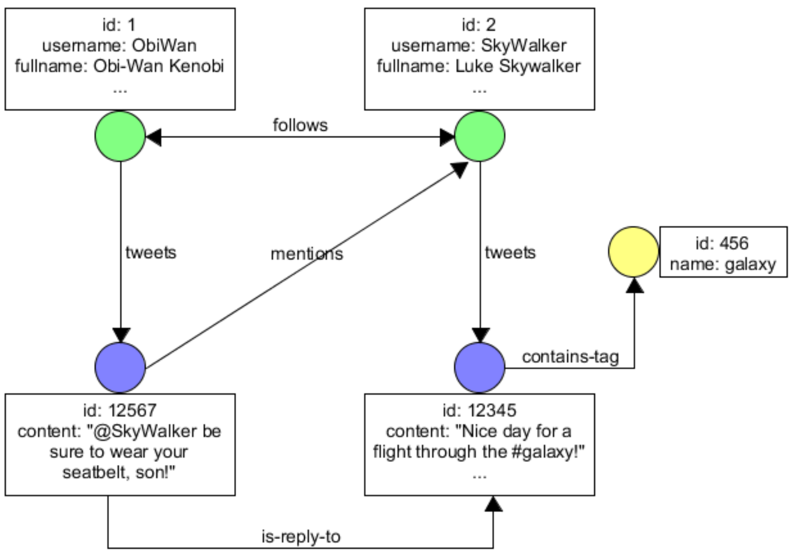


Figure 5: An example data set of Twitter in a graph.

Social data can be related in many different ways, and there is no one-size-fits-all solution to generate efficient partitions to minimize network traffic (also see section 2.2). Furthermore, the amount of data that can be transferred between nodes in one single network message depends on the bandwidth of a system. A large bandwidth means that network messages can contain a relatively large amount of data, resulting in less frequently occurring network requests. The opposite is true when the bandwidth is small, requiring the system to execute relatively many network requests. In the worst case, the response time of the system depends on the speed and capacity of the network, leading to unwanted overhead in the form of latency.

This latency can really become a problem in a situation where the system does not utilize its full potential, resulting in idle time and latency playing the biggest part in the total response time of a system. A completely centralized (non-graph) database obviously does not have these latency problems, but in return lacks the advantages of the scalability of a distributed system.

2.5 Choosing a graph management system

The previous section has described graphs, and how OBI4wan's current data set could be translated into a graph data store. At this point it is known that there is a need for a system that can store and query these graphs. Before diving into the discussion about a centralized versus a decentralized solution, we first need to know what kind of systems we can use. This section could mention a few of these systems, and these systems can be described in more detail in the next section with *related work*. Once the choice for systems (Titan and MonetDB) is discussed, we arrive at the discussion between centralized and decentralized, which is actually a discussion about the difference in architecture between Titan (distributed) and MonetDB (centralized).

2.6 Centralized versus decentralized

This section describes one of the main contributions of this paper. namely benchmarking a graph network on both a distributed system (Titan) and a centralized system (MonetDB), and analyzing which one performs the best (in the use case of OBI4wan).

In this research, the main question to answer is whether a decentralized solution in combination with a dedicated graph database (Titan), graph query language (Gremlin) and text-search capabilities (ElasticSearch) is preferable over a centralized solution (column store MonetDB) in the specific use case of the OBI4wan data set. The LDBC benchmark (see sections 3.5.1 and 8)) will be used in order to test the performance-, scalability- and update[1] level of both solutions. An important question for both solutions is if they are 'future-proof': with an ever increasing amount of data, will the proposed solutions still scale while keeping the required level of performance? In the specific situation of a commercial company like OBI4wan, also the costs and benefits ratio is an important factor, aiming at a situation where as many queries per dollar can be executed while maintaining a respectable response time that satisfies customers.

[1]Databases need to cope with continuous streams of data, which need to be pushed into the database as an update regularly.

3 Related work

At this moment, the related work section roughly describes the following: some graph networks, some graph query languages and other ways to store data in databases (to include MonetDB). Right now, the related work section dives directly into graph databases, without giving the bigger picture of graph networks, what they are and how they are used. This general overview of graph networks should be added to this related work section, or maybe even to a stand-alone section. Especially in the general overview of graph networks, OBI4wan's current situation can be compared to graph networks, to give an impression of how OBI4wan can benefit from using graph networks.

The related work subsections do not yet mention other papers in which the "related work" is used. For example, the subsection about graph databases could mention a few papers in which these graph databases are used, let's say two or three per graph database. This also counts for graph query languages, graph benchmarks, etc.

Research questions:

- **Graph networks**: In the most general form: what are graph networks, what do they look like, how and where are (and could they be) used?
  - This is meant as a very general question about graph networks, serving as a starting point for - for example - the related works section. Later, the paper can narrow down on graph databases and graph query languages (which are included in the world of graph networks), showing how a graph network is stored and queried.
- **Graph databases**: which graph databases are available at this moment ('market research')?
  - Which of these graph databases are suitable to use in combination with the OBI4wan data set?

This section contains an overview of related work in the areas of graph networks, graph databases, graph programming, graph query languages and other related database systems.

3.1 Graph networks

A broad overview of graph networks, without narrowing down on how to store these networks (graph databases) and how to query these networks (graph query languages).

A graph (network) G is a collection of vertices V and edges E, together in the pair of sets $G = (V, E)$ [75]. Vertices from V - say v_1 and v_2 - can be connected to each other through an edge e (with some label l), showing there exists a relationship between these vertices: $e(l) = (v_1, v_2)$. Relationships can also be directed, transforming an edge into what is called an arc from one vertex to another (with some label l): $a(l) = (v_1, v_2)$. Because arcs are directed, they always start in one vertex, and end in another (or the same, resulting in a loop). The two pairs $a_1 = (v_1, v_2)$ and $a_2 = (v_2, v_1)$ are two different pairs, because of the directed nature of arcs.

Graphs can be of multiple types [45], for example simple graphs (only simple vertices and edges), hypergraphs (vertices can be grouped, and edges can be relationships between these groups), nested graphs (vertices can contain graphs themselves) and property (attributed) graphs (vertices are objects with attributes - a social graph is an example of this graph type).

3.2 Graph databases

Graph networks can be stored in a graph database, where data structures of (real-world) objects are modeled as a graph. Each of the graph types mentioned in section 3.1 can be used in a variety of database types, for example web-oriented databases, document-oriented databases and triple stores, all potentially combined with in-memory graph tools for fast query execution [45]. The following subsections provide an overview of a couple of graph databases that are currently used in practice: Titan, Neo4J, Sparksee, FlockDB and RDF databases.

3.2.1 Titan

At this moment, Titan is introduced in this section, without going very deep into the internals of Titan. This more in-depth discussion of Titan can be added to this subsection, but another option is to dedicate a separate section to Titan. Then this 'related work' section could introduce Titan, and then refer to this separate section for more detailed information.

Titan is a cluster-based graph database that is designed for storing and querying graphs with hundreds of billions of vertices and edges [2]. The main focus of Titan is compact graph serialization, graph data modeling and the efficient execution of small, concurrent graph queries (with native support for graph query language Gremlin). Aurelius [8], a team of software engineers and scientists in the area of graph theory and the creators of Titan, published two white papers advocating the scalibility, usage cost and speed of Titan. The first article (Titan Provides Real-Time Big Graph Data [69]) details a benchmark consisting of users concurrently using a Titan graph database hosted by 40 Amazon EC2 m1.small instances [9]. The results of the benchmark show that letting 50.000-100.000 users continuously and concurrently interact with a Titan graph database while maintaining reasonable response times (using an Amazon cluster) costs almost 100.000 dollars per year. The second article (Educating the Planet with Pearson [70]) details a graph database consisting of 6.24 billion vertices (being universities, students, teachers, courses, etc.) and 121 billion edges. The research shows reasonable performance with a load of 228 million transactions in 6.25 hours (10.267 transactions per second and a maximum load of 887 million transactions per day, given that the workload of the transactions is relatively high). Queries with normal workload would then result in a maximum of around 1 billion transactions per day, according to the article.

3.2.2 Neo4j

Besides Titan and Gremlin, there are a lot of other graph database solutions out on the (open source) market. One of the most popular solutions among them is Neo4j, claiming to be "the world's leading graph database" [16]. It uses Cypher [19] as its graph query language (see section 3.4.2). Neo4j claims to be built to perform at scale, supporting graph networks of tens of billions of vertices and edges, combined with hundreds of thousands of ACID[2] transactions per second [17]. However, research has pointed out that Neo4j is relatively powerful compared to other graph database solutions when it comes to read-only operations on the data set, but performs relatively poor when executing edge- and property-intensive write operations on the data set [46] [47]. Another benchmark [48] shows that Neo4j indeed scales when the graph size increases, but mainly for relatively small graphs of less than 32.000 nodes. As shown in the introduction, ten days of Twitter data already contain more than 15 million tweets, which would clearly be to

[2]ACID stands for four rules that databases should be able to satisfy: being **A**tomic (transactions should be executed completely, or not at all), **C**onsistent (each transactions - failed or succeeded - should lead to a consistent state of the database), **I**solated (transactions are independent from each other, and are executed in an isolated environment) and **D**urable (completed transactions cannot be undone).

much for a scalable, distributed Neo4j graph database. The same test shows that Titan is better able to scale with such large graphs and keeps linearly scalable when the graph size increases.

Besides the aforementioned scalability problem, Neo4j has a couple of other (possible) shortcomings which withhold from using this solution in this specific research.

1. Neo4j has a relatively large memory footprint.
2. Query execution in Cypher is relatively slow. Cypher is designed to be a human-readable and understandable language, optimized for reading and not for writing [22]. In other words, the main focus in the development of Cypher has been to create a readable language, leaving a less important role for fast query execution. Benchmark results have indeed shown that Cypher is slower than Titan's query language Gremlin [47] in several situations (such as recommendation engines and friend-of-a-friend queries).
3. Cypher has no built-in query optimization. While it is possible to try and optimize queries yourself (for example by letting queries return only that part of the graph that is needed in further computations), Cypher does not offer built-in query optimization techniques.

However, future work could include Neo4j in benchmarks and test if that solution is indeed not scalable enough to work with large social data sets.

3.2.3 Sparksee

Sparksee (formerly known as DEX) is a graph database management system that is "tightly integrated with the application at code level" [74]. The model used by Sparksee is based on a *labeled attributed multigraph*, where all vertices and edges can have one or more attributes (*attributed*), edges have labels (*labeled*) and can be bidirectional (*multigraph*).

An example visual representation of a Sparksee graph can be found in Figure 6. This graph shows that Sparksee allows various vertex- and edge types in one single graph, in this case actors/directors (represented by the star-icons) and movies (clipboard icons) as vertex types, and 'cast' (green lines) and 'directs' (red lines) as edge types. All vertices have name attributes representing the actor/director- and movie names, and the edges of type 'cast' have an attribute containing the name of the character played by an actor in a movie.

Sparksee uses its API to execute queries on a graph. An example (Java) program with queries is shown below [74].

```
[1] Objects directedByWoody = g.neighbors(pWoody,directsType,
    EdgesDirection.Outgoing);
[2] Objects castDirectedByWoody = g.neighbors(directedByWoody,castType,
    EdgesDirection.Any);

[3] Objects directedBySofia = g.neighbors(pSofia, directsType,
    EdgesDirection.Outgoing);
[4] Objects castDirectedBySofia = g.neighbors(directedBySofia,castType,
    EdgesDirection.Any);

[5] Objects castFromBoth = Objects.combineIntersection(castDirectedByWoody,
    castDirectedBySofia);
```

These lines of code result in the following behavior:

1. The method $neighbours()$ retrieves all vertices that are connected to $pWoody$ via an outgoing edge of type $directsType$. In this line of code, g is the variable that points to the whole graph, $pWoody$ points to the vertex with the name-attribute $Woody\ Allen$ and $directsType$ points to the edge type $DIRECTS$. The resulting set of vertices contains movies which are directed by Woody Allen, and this set is stored in the variable $directedByWoody$.

2. Starting from all vertices in the set $directedByWoody$, find all vertices that are connected to the vertices in this set by any edge of type $castType$. The resulting set of vertices contains actors that have played a role in a movie that was directed by Woody Allen, and this set is stored in the variable $castDirectedByWoody$.

3. Same as line 1; $directedBySofia$ contains all movies which are directed by Sofia Coppola.

4. Same as line 2: $castDirectedBySofia$ contains all actors which have played in a movie that was directed by Sofia Coppola.

5. The final step is to take the intersection of $castDirectedByWoody$ and $castDirectedBySofia$, giving all actors who have played a role in a movie directed by Woody Allen **and** played a role in a movie directed by Sofia Coppola.

The result of this graph traversal is the actress Scarlet Johansson.

Sparksee also provides an implementation for Blueprints, allowing developers to use a Sparksee graph in combination with the tools from the Blueprints framework like the graph query language Gremlin.

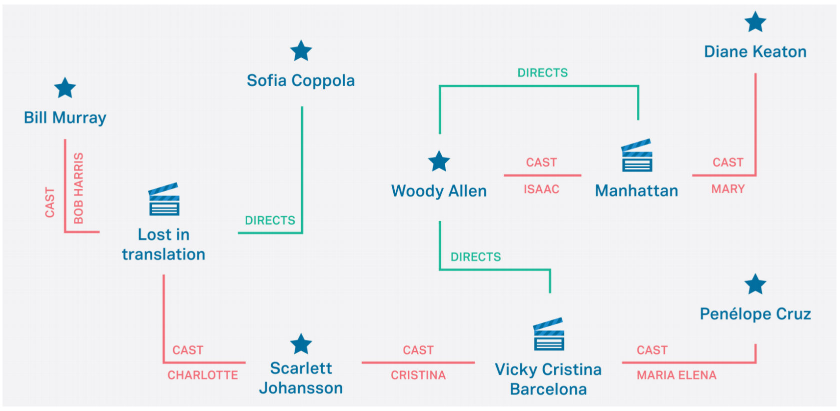


Figure 6: An example of a Sparksee graph.

3.2.4 FlockDB

FlockDB is ”a distributed graph database for storing adjacency lists” [20]. It is used by Twitter to store a variety of social graph types, for example graphs that store follow-relationships between users. Relationships (edges) between two nodes A and B are always stored in two directions: from A to B and vice versa. For example, ’follows’-relationship is stored as (1) A follows B and (2) B is followed by A. This allows for queries in both directions (both queries asking *who follows* A and *who is* A *following* are possible).

Consider the simple graph as shown in Figure 7. The following three lines of code show some of the capabilities of the query language provided by FlockDB [21].

```
[1] flock.select(1, :follows, nil).intersect(nil, :follows, 1).to_a
[2] flock.select(1, :follows, nil).union(nil, :follows, 1).to_a
[3] flock.select(nil, :follows, 1).difference(1, :follows, nil).to_a
```

These lines of code result in the following behavior:

1. The first *select*() function returns the set of all users that user 1 follows. The *intersect*() function is then executed for all users in this set, and returns all users that follow user 1 **and** are followed by user 1.
2. The difference between this line and the previous is that the *intersect*() function is replaced by the *union*() function, meaning that this query returns all users who either follow user 1, or are followed by user 1 (removing any duplicates).
3. The first *select*() function returns the set of all users that follow user 1. The *difference*() function is then executed for all users in this set, and returns those users that are in the first set, but not in the set of users that user 1 follows. In other words, the resulting set contains users that follow user 1, but are not followed back by user 1.

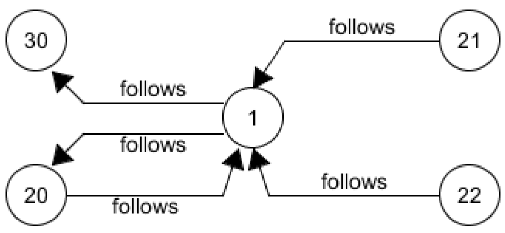


Figure 7: An example graph showing follow-relationships between Twitter users.

3.2.5 RDF

RDF (***R**esource **D**escription **F**ramework*) is a labeled graph data format that is used to represent all kinds of information on the web [23]. Data in RDF is stored in *triples*, containing two entities (subject and object) and a relationship (predicate) between those entities (compared to vertices and edges in graph databases). An example of a triple in RDF (stored in XML-format) is shown below.

```
<rdf:RDF
<rdf:Description rdf:about="http://www.example.com/rdf">
  <si:title>Example.com</si:title>
  <si:author>John Watts</si:author>
</rdf:Description>
</rdf:RDF>
```

This XML document actually stores two triples, with the same subject but with different predicates and objects. The text $rdf : about = "http : //www.example.com/rdf"$ shows the subject of both triples. The two tags $< si : title >$ and $< si : author >$ show the two predicates title and author, respectively. Finally, the objects are shown inside these predicate-tags: "Example.com" and "John Watts", respectively. The graphical representation of these two triples is shown in Figure 8.

A shortcoming of RDF is that it is not possible to store properties (or other metadata) on a predicate between a subject and an object [49]. For example, it is not possible to attach a 'certainty statement' to a predicate. Consider the RDF triple {*ex:Alice foaf:knows ex:Bob*}, where *ex* and *foaf* are abbreviations for the used namespaces[3] *example* and *friend-of-a-friend*, respectively. In standard RDF, it is not possible to attach a certainty statement on the predicate *knows*, indicating how certain it is that *Alice* knows *Bob*. A proposed extension of RDF (*RDF**) tries to provide a solution to this problem, introducing the notion of 'triples about triples'. In other words, the subject or object of a triple can be a triple of itself. This allows to create triples that can both express that *Alice* knows *Bob*, and that this relationship exists with a certainty of *0.5*. To express this, the first triple {*ex:Alice foaf:knows ex:Bob*} is the subject of the triple {*ex:subject ex:certainty 0.5*}. Together - in one combined triple - this looks as follows:

```
<<ex:Alice foaf:knows ex:Bob>> ex:certainty 0.5
```

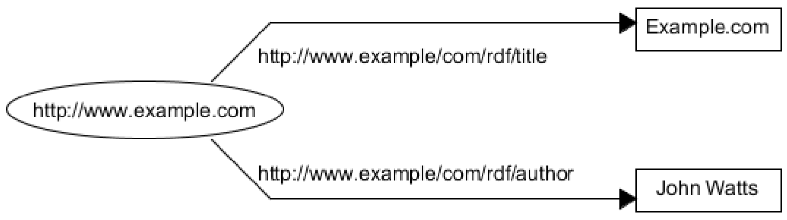

http://www.example.com
http://www.example/com/rdf/title
Example.com
http://www.example/com/rdf/author
John Watts

Figure 8: A visual representation of two RDF triples with the same subject, but different predicates and objects.

3.3 Graph programming

The following section (*graph query languages*) gives an overview of graph query languages that are used to built queries that can be executed on graph systems (described in the section above, *graph databases*). In between those sections about graph systems and graph query languages, a general introduction about graph programming can be given. This section details about graph programming languages, how these languages are structured and how they are different compared to 'normal' programming languages. Languages (systems) that could be mentioned here are Giraph and Pregel. As an example, some famous graph network problems (*Seven Bridges of Konigsberg*) and how they can be solved using graph programming could be described.

3.3.1 Graph language structure

...

3.3.2 Graph programming versus other programming types

...

[3]A namespace in RDF refers to a collection of elements which can be used as subjects, predicates or objects.

3.3.3 Example: solving graph problems

...

3.4 Graph query languages

Where SQL is used as the query language for traditional, relational databases, languages like Gremlin are used to execute queries (or rather traversals) on graph databases. A graph query language is (or should be) capable of supporting a variety of query types [45]:

- **Adjacency queries**: testing whether vertices are adjacent. For example, the two vertices v_1 and v_2 are adjacent if there exists an edge e_i which connects these vertices in a relation $e_i = (v_1, v_2)$.
- **Reachability queries**: testing whether vertices are connected by a path through the graph. For example, the vertex v_2 is reachable from vertex v_1 if there exists a path (a combination of connected edge relationships) between these vertices. The two edge relationships $e_1 = (v_1, v_3)$ and $e_2 = (v_2, v_3)$ show that there exists a path between vertices v_1 and v_2 via vertex v_3.
- **Pattern matching queries**: finding a specific pattern [subgraph] of vertices and edges. For example, the pattern matching query $PMQ = (V, E, R)$ with $V = (v_1, v_2)$, $E = e_1$ and $R = \{e_1(l) = (v_1, v_2)\}$ returns true if there exists a subgraph with vertices v_1 and v_2 connected by a relationship with the label l: $e_1(l) = (v_1, v_2)$.
- **Summarization queries**: summarizing query results, for example finding a maximum, minimum or average. For example, calculating the total value of a certain vertex attribute of a connected graph with vertices v_1, v_2 and v_3 by adding up all values from this numerical attribute per vertex.

3.4.1 Gremlin

Gremlin is a query language for property graphs and part of the TinkerPop [11] framework. Titan is an example graph database that uses Gremlin as its graph query language, but any graph database or framework that implements the Blueprints [4] [15] property graph database modal can in principle make use of Gremlin. It provides solutions for the expression of many types of graph queries (graph traversals) in "a more understandable manner than with traditional programming languages" [13].

An example property graph on which Gremlin can execute graph queries is presented in Figure 9. Gremlin is based on graph traversals, where the starting point of a traversal is either a vertex or an edge. A small example of a program in Gremlin is the following one [14]:

```
[1] g = TinkerGraphFactory.createTinkerGraph()
[2] v = g.v(1)
[3] v.out('knows').filter{it.age > 30}.out('created').name
```

These lines of code result in the following behavior:

[4]Blueprints is a collection of interfaces, that allows developers to "plug-and-play" their graph database backend. It consists of the data flow framework *Pipes*, the graph traversal language *Gremlin*, the object-to-graph mapper *Frames*, the graph algorithms package *Furnace* and the graph server *Rexter*. Find more information at Blueprints website [15].

1. The graph presented in Figure 9 is used in many Gremlin examples, which has promoted this graph into a standard, easy to create graph in Gremlin. The method $createTinkerGraph()$ initiates this specific property graph, and stores it in variable g.
2. The nodes in the graph are numbered with identifiers. The vertex with ID 1 is stored in variable v to use later on.
3. The real graph traversal happens in this line. Starting from vertex v, this line finds all outgoing edges of v that contain the label *knows*. This intermediate set of resulting vertices is then exposed to the filter $age > 30$, reducing this intermediate set to only those vertices of which the *age*-attribute has a value higher than 30. Finally, from this new set of intermediate vertices, all outgoing edges with the label *created* are followed, and the *name*-attributes of this final set of vertices on the end of these edges are printed to the console.

In human language, we now found all projects that were created by people that we (the vertex v) know and are older than 30 years.

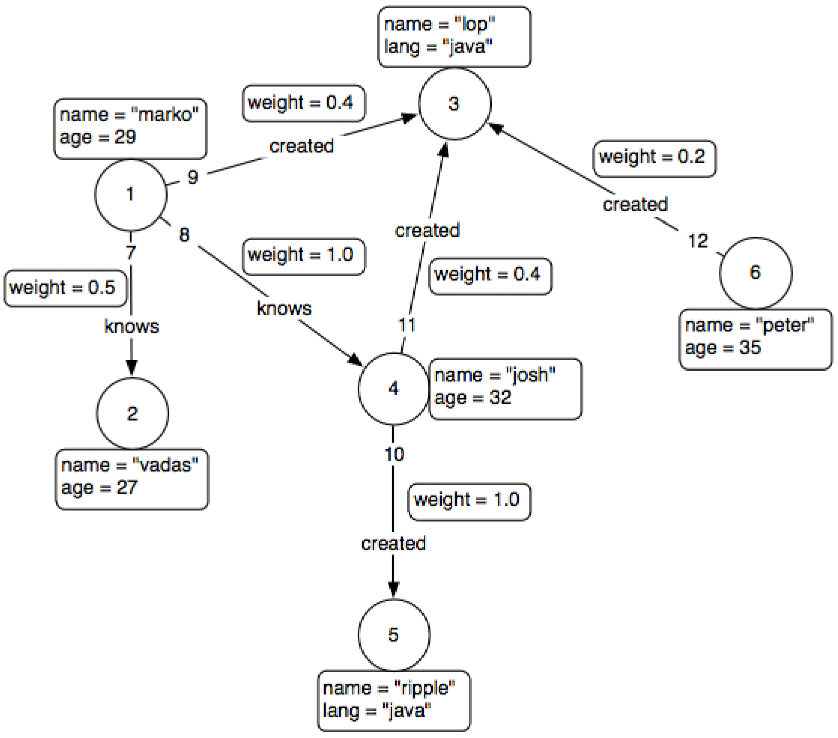


Figure 9: An example property graph database.

3.4.2 Cypher

Cypher is ”a declarative graph query language that allows for expressive and efficient querying and updating of the graph store” [22]. The language is used as the graph query language for the graph database Neo4j [16]. One of the goals of Cypher is that it is a human-readable language for developers as well as operations professionals. Constructs in Cypher are based on English prose, where optimizations in the language are based on better readability and not on easier writing. Cypher is inspired by the relational database language SQL, and uses part of the SQL-slang for the same purposes (for example *WHERE* and *ORDER BY*).

An example graph database that can be queried using Cyper is shown in Figure 10. The following program contains some example lines of codes in the Cypher query language.

```
[1] MATCH (user)-[:friend]->(follower)
[2] WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', Steve']
    AND follower.name =~ 'S.*'
[3] RETURN user, follower.name
```

These lines of code result in the following behavior:

1. The *MATCH* clause retrieves all users (vertices) that have a 'friend'-relationship with another user. The resulting set of users is stored in the variable $follower$; the vertices on the other end of this relationship are stored in the variable *user*.

2. The *WHERE* clause filters the set of users that has been found in the previous line. It only stores users having one of the names from the list **and** starting with the letter *S*.

3. The *RETURN* clause simply returns the earlier created variables, for example printing them to the console.

In other words, these lines of code return pairs of the form $< user, follower.name >$, where *user* is a complete user object, and $follower.name$ is the name-attribute of a $follower$ object. In the case of the graph from Figure 10, the resulting pairs are the following ones:

- $< Node[3][name : "John"], "Sara" >$
- $< Node[4][name : "Joe"], "Steve" >$

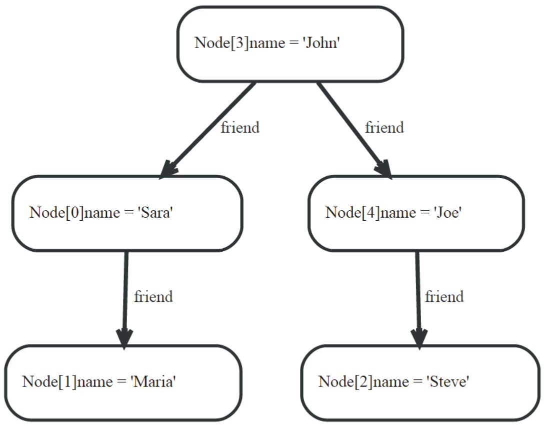


Figure 10: An example property graph database that can be queried by Cypher.

3.4.3 SPARQL

The previously discussed languages are all real graph query languages, in the sense that they are designed to execute queries on real graph databases. SPARQL (***S****PARQL* ***P****rotocol* ***a****nd* ***R****DF* *Query* ***L****anguage*) is designed to execute queries on RDF (see section 3.2.5).

We can use the SPARQL language to execute queries on RDF-triples. The basic syntax of SPARQL is similar to the SQL syntax, working with the *SELECT*, *FROM* and *WHERE* clauses. Consider the following SPARQL query, that can be executed on the RDF triple store that was presented in Figure 8:

```
[1] PREFIX si: <http://www.example.com/rdf/>
[2] SELECT ?name
[3] WHERE { ?website si:author ?name . }
```

The first line defines that a specific namespace for the predicate relationship is used in the query. The third line retrieves all triples of the form <*subject, predicate, object* >, where the subject is stored inside the variable *?website*, the object in the variable *?name* and the relationship (predicate) is *si:author*. Finally, the second line provides the *?name* variable of all entries as the result of the query, in this case the only result is the name *John Watts*.

SPARQL provides a specialized way to execute graph traversals, which are called *property paths* [24]. These property paths can be constructed using the sequence operator (/), which basically represents the hops over vertices that are visited in a graph traversal. An example SPARQL query is presented below.

```
[1] ?x foaf:mbox <mailto:alice@example> .
[2] ?x foaf:knows/foaf:knows/foaf:name ?name .
```

This query retrieves the names of all people who are exactly two (*foaf:knows*) steps away from Alice. The first line retrieves Alice's identity (her name) based on her email address, and saves this name in the variable *?x*. The second line then searches for all people who are exactly two (*foaf:knows*) steps away from Alice, and stores these people in the variable *?name*. The "two steps away from Alice"-part is represented by the path *foaf:knows/foaf:knows*; this path is constructed using the earlier mentioned sequence operator (/). This path first retrieves all people who know Alice (one step away), and subsequently retrieves all people who know people who know Alice (two steps away). This behavior is similar to graph traversals in specialized graph query languages like Gremlin (section 3.4.1) and Cypher (section 3.4.2).

3.4.4 SQL

SQL (*Structured Query Language*) is primarily designed to execute queries on relational database management systems, but with some assumption as to how the data in such a relational database is stored, SQL can also be used to query a graph-based data structure.

In order for SQL to behave like a graph query language, the data in the relational database should be saved in a format that is similar to the triples of RDF. Tables in the database should consist of three columns, two of them being representations of vertices, and the other one being the representation of an edge (relationship between the vertices): $< vertex_1, vertex_2, edge >$. For example, consider the following database table:

vertex1	**vertex2**	**edge**
John	Sara	follows
Maria	Robert	follows
Sara	Maria	follows

This table represents a graph with four users as vertices, and three edges defining (directed) ’follows’- relationships between these users. Figure 11 is a graphical representation of this graph.

Standard SQL queries can now be used in order to find answers on questions like ”give me all users that Sara follows. This query is shown in code below.

```
[1] SELECT vertex2
[2] FROM table
[3] WHERE vertex1="Sara" AND edge="follows"
```

This query returns the only user that Sara follows: *Maria*.

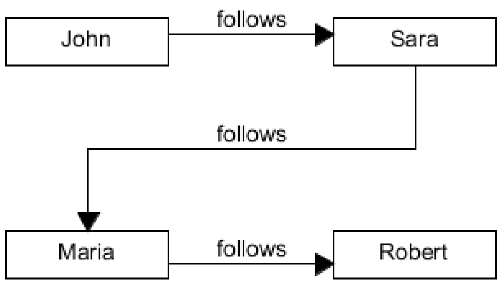


Figure 11: A visual representation of a graph-based table of a relational database.

Graph traversals in SQL can also be imitated by using recursive SQL in combination with the *UNION (ALL)* operator. Recursive SQL starts with a non-recursive part, for example retrieving a vertex in the graph that has specific properties, such as a name. Then, starting from this vertex, the recursive part of the query keeps fetching new vertices until a predefined final destination (another vertex) has been reached. The path through the graph that is obtained in this way generally has to satisfy a predefined condition, for example the absence of cycles. The result is a collection of traversals through the graph. This collection can be further analyzed, for example by searching for the smallest path between two vertices, based on the combined numerical weight on the edges that lie on the path between the start vertex and the destination vertex.

The syntax of recursive SQL with the *UNION ALL* operator is as shown below [72].

```
WITH RECURSIVE <cte_name> (column, ...) AS (
  <non-recursive_term>
  UNION ALL
  <recursive_term>)
SELECT ... FROM <cte_name>;
```

In the above query, the abbreviation *CTE* stands for *Common Table Expression*, which allows to split big, complicated queries into subqueries, to ensure better readability. In the syntax above, *cte_name* is a variable that points to a complete SQL query, of which the output can be used in the *FROM* clause (e.g. *FROM cte_name*) of another query.

Image a recursive SQL query that returns all paths (and their total length) between two vertices A and B, with the goal of finding the shortest path between A and B. This query is split in two parts, based on the syntax that is presented above: a non-recursive term and a recursive term. The non-recursive term selects the starting vertex (vertex A) and all its outgoing edges.

Then, the recursive term follows the outgoing edges from vertex A, stores the vertices on the other end of these edges in an intermediate result, and uses these vertices as an input for the next recursion step (a recursion step has access to the results - in this case a vertex set - of the previous recursion step). This process continues until all paths through the graph that have been found in the recursion have reached the destination vertex. The output is the set of all paths from vertex A to vertex B, with their total, combined length. Finally, the shortest path can be retrieved from this output set.

might give more example code

3.5 Graph benchmarks

A substantial part of this paper writes about running benchmarks on (graph) database systems, for now the battle between Titan (distributed, dedicated graph database) and Monet (column store). This subsection will describe the benchmark system that will be used in this project (the driver from the LDBC), but can (and probably should) also discuss alternative (graph) benchmark systems, such as Graphalytics. This is a big data benchmark for graph-processing platforms.

3.5.1 Linked Data Benchmark Council (LDBC)

The Linked Data Benchmark Council (LDBC for short) is an organization formed by researchers from universities and the industry that offers independent benchmarks for both RDF- and graph technologies [50]. The vision of the LDBC is that without an effort to provide such benchmarks, there will be no good way for end-users to compare graph database systems and no good guidelines for the developers of such systems, which could endanger the future of this technology.

LDBC benchmarks Currently, there are two benchmarks that are being developed and maintained by the LDBC, which are the Social Network Benchmark (SNB) and the Semantic Publishing Benchmark (SPB) [51].

- The SNB focuses on graph data management workloads, and tries to mimic the events of a real social network like Facebook or Twitter. It contains a collection of short read queries, complex read queries and update queries, which are fired on a database management system in a predefined distribution. This distribution is set up in such a way that it contains many short read queries (e.g. looking up a person or a message) and update queries (e.g. adding a new message) in between the complex queries (e.g. looking up friends of friends, introducing the need for deeper graph traversals).

- The SPB focuses on the management and usage of RDF metadata about of media assets or some form of creative works, like productions from the British Broadcasting Corporation (BBC). The SPB contains two separate workloads. The first workload is the *editorial workload*, which mimics the behavior of the (semi-automated) process of inserting, editing and deleting metadata into a database. The second workload is the *aggregation workload*, which simulated the aggregation of content from the database to use it in some external source, for example on a website. The aggregation is performed by a set of (SPARQL) queries.

Because the data collection of OBI4wan is originating from the social network Twitter, this research uses the SNB to compare how queries on this data perform on different database management systems.

LDBC data set In addition to their benchmarks, the LDBC has also developed a data generator called DATAGEN. This data generator can generate data that is similar to the data of a real social network, containing persons, organizations where those persons work for, places where those persons live in, messages created by those persons, tags used in those messages, forums in which those messages are posted, etc [52]. The final data set that DATAGEN produces contains correlations between the attributes of entities. For example, persons who where born in Germany have a first- and last name which occur often in that country. DBpedia has been used to provide such information. Another example is that people who are located in the same place often share the same interests, which may be propagated into the tags they use in their messages. Summarized, DATAGEN tries to produce a social network data set that is as realistic as possible.

DATAGEN can produce data sets of different sizes. The total size of a data set is based on the number of persons that a user can give as an input for data creation, or by a numeric scale factor that represents the total amount of data in gigabytes. The output format of DATAGEN is either CSV or Ntriple (an RDF-related format).

3.5.2 Transactional Processing Performance Council (TPC)

The TPC also provides benchmarks, but for relational database systems instead of RDF/graph databases.

The Transactional Processing Performance Council (TPC) is an organization that defines benchmarks for transactions on databases, such as inventory controlling, money transferring or some kind of reservation processing. There exist some variants of TPC, which are described in full detail on TPC's benchmark overview website [25]. A quick overview of the variants is given in the list below.

TPC-C The TPC-C benchmark simulates the managing of a product or service. Examples of transaction on the database are the insertion of new products/services, checking the status of products/services, monitoring a products stock, etc.

TPC-DI The TPC-DI benchmark (for Data Integration) simulates the unification of data from different sources. In other words, it transforms data that is arriving from different organizations in different formats into one unified, standardized format. An example of data integration is when two (or more) organizations merge together, introducing the need to transform their different data formats into one uniform data format.

TPC-DS The TPC-DS benchmark (for Decision Support) simulates decision making based on a data set in the database. An example of decision making is a medical database that contains data about diseases and its symptoms, that helps doctors to diagnose a patient with a disease based on the patient's symptoms. Another decision support benchmark is provided by TPC-H, which contains more queries than TPC-DS.

TPC-E The TPC-E benchmark is an On-Line Transaction Processing (OLTP) workload, which simulates entering and retrieving data into and from a database. An example is an organization or person that acts as the middleman between a customer and another company, by receiving an order from the customer (entry into a database) and processing that order to send it to the company (processing and retrieving from a database).

TPC-VMS The TPC-VMS benchmark (for Virtual Measurement Single System Specification) is a combination of TPC-C, TPC-E, TPC-H and TPC-DS. It requires to set up three database workloads on one single server, and then choose and execute one of these four benchmarks on all three databases. The result of TPC-VMS is the minimum value of the three benchmark executions.

TPCx-HS The TPCx-HS benchmark (x for Express) provides a way to measure the performance and availability of systems and platforms that use the Apache Hadoop File System API for big data processing.

3.5.3 Graphalytics

Graphalytics is a "big data benchmark for graph-processing platforms" [53]. The main goal of Graphalytics is to provide benchmarks for distributed graph processing platforms, although also traditional graph databases like Titan and Neo4J are supported. Another goal for Graphalytics is to develop a platform that is "future-proof", in a way that it can be used in any (currently unknown) new graph platform. To make this possible, Graphalytics is built in such a way that a new graph platform only needs to write its own platform-specific algorithm implementation in order to execute the benchmarks. The data set that can be used for the benchmark is generated using a subset[5] of the data that is produced by the LDBC's DATAGEN (see section 3.5.1).

The benchmarks provided by Graphalytics use algorithms that mimic real world scenarios. Currently there are five algorithms built into Graphalytics, but given the extendibility of Graphalytics this number may increase in the future. The four algorithms are *general statistics* (vertices and edges count), *breadth-first search* (starting at a root vertex, first visit all its neighbors, then all neighbors of its neighbors, etc.) *connected components* (find all connected entities of a vertex) and *community detection* (find strongly connected groups) and *graph evolution* (predicting graph evolution).

3.6 Other databases

3.6.1 MonetDB

At this moment, MonetDB is introduced in this section, without going very deep into the internals of MonetDB. This more in-depth discussion of MonetDB can be added to this subsection, but another option is to dedicate a separate section to MonetDB. Then this 'related work' section could introduce MonetDB, and then refer to this separate section for more detailed information.

MonetDB is a column store, meaning its technology is built on the representation of database relations as columns (as opposed to row-based representation, see below for a discussion of both the row-based and column-based representations). This principle enables storage of entities of up to hundreds of megabytes swapped into main memory and stored on disk [10]. Furthermore, MonetDB is also designed for multi-core parallel execution on desktops, to reduce the execution time of complex queries [10]. Distributing the processing of queries can be done by using a map-reduce scheme for simple problems or by taking the distribution of processes into consideration at database design time to support more complex cases [10].

Column-based databases differ from row-based databases in a couple of ways, each of them having their own advantages and disadvantages. Row-based databases can efficiently return complete objects (row entries) with all their attributes, making them a good choice for handling OLTP (transaction) workloads, consisting of many small queries that each retrieve or update a handful of rows. Column-based databases are more efficient to return results based on OLAP (analysis) queries, for example a query that retrieves all objects of which the 'datetime' property falls inside a certain range. Row-based databases can imitate the behavior of column-based

[5]Graphalytics only uses person entities and the 'knows' relationship between those persons.

databases slightly by keeping (part of) the database in an index that consists of mappings between row IDs and column values.

Discussion of MonetDB is not thorough enough at this moment. See the explanation of MonetDB that Peter provided during the meeting on Tuesday, March 17, and use this explanation to update this section. TODO

MonetDB will be the 'competitor' of Titan in the benchmark sessions of this project. The advantage of MonetDB is that it is designed to be usable with not only the relational model (with SQL as query language), but also with a variety of emerging models like object-oriented, object-relational [73] and subsequently - in the interest of this research - a graph-oriented model. All data in MonetDB is stored in Binary Association Tables (BATs), consisting of two columns. Various data formats can be mapped into BATs. For example, each column in a relational model can be mapped into a BAT, where the right side of the BAT contains the column value, and the left side contains an identifier [73]. This process of mapping can be summarized as 'vertical fragmentation', which optimizes I/O and memory cache access and allows for data abstraction, thus supporting a variety of data models [73].

3.7 A note on graph standards

The previous sections have shown a wide variety of graph databases and graph query languages, all with different methods to store a graph database and execute queries on it. At the moment of writing, there is no real standard regarding graph databases, graph query languages and their syntax and semantics. Transferring a graph database from one solution to another easily can therefore be a daunting task. Consider the property graph model that is used by Titan. Even this specific type of graph database is not standardized, allowing multiple ways to store such a graph database. For example, multiple properties of the same type can be saved with a vertex in two different ways: (1) each property type occurs only once, with the possibility to store a list of values per property type, or (2) each property type can occur more than once, storing only one value per property.

To clarify these two different methods, consider storing telephone numbers as a property on a vertex. The first storage method would store a list of phone numbers for the *telephone number* property, which would look as follows:

```
Vertex A
Telephone number(s): {012-3456789, 789-6543210}
```

The second storage method would store one value per property, allowing for multiple properties of the same type:

```
Vertex B
Telephone number: 012-3456789
Telephone number: 789-6543210
```

This lack of standardization of the graph database model makes it difficult to perform the same experiment on different graph database systems, because it is not always trivial to transfer the used data set (and the format it is in) from one graph database solution to another. This has consequences when the research that is presented in this paper would be performed again using another graph database solution and -query language.

4 Titan

This section provides a detailed overview of Titan, including a description of its internals.

Research questions:

- **Architecture**: given the data set of OBI4wan, a graph database solution (and its query language), what is the best option: 'shared memory' or 'shared nothing.
  - When choosing a 'shared nothing' (communication-driven) solution, what is the optimal size of a network message? How many data can one message receive given the network bandwidth? How long can you postpone sending network messages without breaking the system?
  - **Architecture**: Titan supports the partitioning (distribution) of a graph. What is the best way to partition a social media graph with Twitter data (e.g. partition on time, or on location, or on personal networks, etc.)?
    - **Note**: this is a big project on its own, and would therefore also fit as a research question for the 'future work' section.
- **Architecture**: Titan claims to be a scalable graph database. Is Titan indeed as scalable as it claims to be? Is Titan 'future-proof', can it continue to scale out indefinitely?
- **Architecture**: what is the TinkerPop framework, and how is Titan related to this framework?
  - Are there any benefits of using Titan in combination with the TinkerPop framework?
  - What are the differences between the various TinkerPop versions that are out on the market (e.g. version 2 and version 3)?
- **Architecture**: how is data stored in a Titan graph database?
  - Because there are multiple storage backends supported by Titan, this question should at least be answered for the storage backend that is actually used. Better would be to describe all possible (useable) storage backends and also use them all separately in benchmarks.
- **Architecture**: how (to what extend) is ElasticSearch integrated into Titan?
  - To answer this question, it is necessary to describe the internals of both ES and Titan. Both descriptions could be provided in this chapter.
- **Architecture**: how can clients interact with a Titan graph database from a remote location? How much latency does this introduce, and is the latency a limiting factor?
- **Architecture**: what is Gremlin, and how Gremlin integrated into Titan. How does the execution of a Gremlin query on a Titan graph database work?
- **Architecture**: is the graph query language Gremlin expressive enough to execute the benchmarks (which were created based on designed business questions)?

This section provides an overview of the Titan distributed graph database. Section 4.1 provides a broad overview of Titan, also discussing the Tinkerpop Stack [11] which elements are integrated with Titan. Section 4.2 details Titan's architecture, describing its internals (the BigTable data model and how this is used by Titan), the data- and indexing backend layers supported by Titan, the query capabilities with graph query language Gremlin and the possibilities to remotely access a Titan graph database.

4.1 Titan overview

In the most general sense, Titan is a graph database engine. An overview of the high-level Titan architecture is shown in Figure 12. Titan's main focus is on "compact graph serialization, rich graph data modeling, and efficient query execution" [27]. For batch loading large amounts of data into Titan and for large-scale graph analytics, Titan uses Hadoop. Furthermore, between the Titan layer and the disk layer sit two more layers: a storage layer and an indexing layer. Both of these layers support a few systems and their adapter implementation, for example storage adapters for Cassandra[6], HBase[7] and BerkeleyDB[8], and indexing adapters for ElasticSearch[9] and Lucene[10]. Titan has implemented the TinkerPop Stack[11] and its Blueprints API into its own TitanGraph API. One of the elements of the TinkerPop Stack is the graph query language Gremlin, which is also used as Titan's query language. A more in-depth explanation of the TinkerPop Stack and its integration into Titan is given in section 4.1.1.

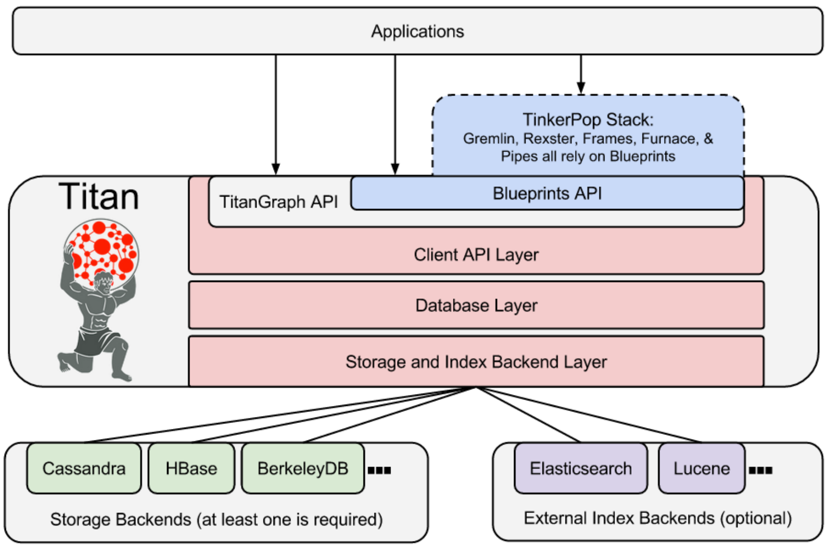

Applications
TinkerPop Stack:
Gremlin, Rexster, Frames, Furnace, &
Pipes all rely on Blueprints
Titan
TitanGraph API
Blueprints API
Client API Layer
Database Layer
Storage and Index Backend Layer
Cassandra
HBase
BerkeleyDB
Storage Backends (at least one is required)
Elasticsearch
Lucene
External Index Backends (optional)

Figure 12: A high-level overview of the Titan architecture.

4.1.1 TinkerPop Stack

The TinkerPop Stack is a platform containing building blocks to develop high-performance graph applications [43]. The version of Titan used in this research (version 0.5.4) is integrated with version 2 of the TinkerPop Stack. The stack consists of six blocks:

Blueprints the foundation of the stack, providing the bridge between a database and the rest of the stack.

Frames providing a "frame" around a Blueprints graph, exposing it as a Java object.

[6]http://cassandra.apache.org/
[7]http://hbase.apache.org/
[8]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html
[9]https://www.elastic.co/
[10]https://lucene.apache.org/
[11]http://tinkerpop.incubator.apache.org/

Pipes	guiding data from input to output.
Furnace	a package of graph algorithms for analysis on graphs.
Gremlin	a graph query language to traverse a graph's content.
Rexster	a server that can be wrapped around a graph, to make it remotely accessible through a REST API.

Note: future versions of Titan (versions 1.x) will be integrated with version 3.x of the TinkerPop Stack. In version 3.x, all building blocks of the TinkerPop Stack have been merged and are referred to as *Gremlin*.

4.2 Titan architecture

The following subsections describe Titan's architecture in more detail. A broad overview of Titan has already been given in the *related work* (see section 3.2.1).

4.2.1 Internals

Internally, Titan stores graphs in the *adjacency list format*. In this format, a graph is stored as a collection of vertices, with one vertex per row of the adjacency list file. Each row consists of the vertex' properties and outgoing edges to other vertices. The adjacency list can be stored in any storage backend that implements the BigTable data model, for example those storage backends that are natively supported by Titan (Cassandra, HBase and BerkeleyDB). The architecture of the BigTable data model is described in the next paragraph. How Titan utilizes this architecture for storing graphs is described in the paragraph after that.

BigTable data model The BigTable data model (shown in Figure 13) is derived from Google's Cloud BigTable data model, and consists of an arbitrary number of data rows which are uniquely defined and sorted by a *key*. Each row can contain multiple *cells*, which further consists of a *column-value* pair. Each of the values in such a *column-value* pair is uniquely defined by the combination of the *key* and the *column* in the key's row. How Titan utilizes this structure to store graphs is discussed in the next paragraph below.

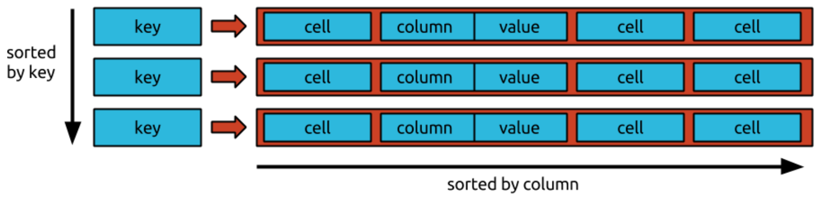


Figure 13: The BigTable data model.

Titan data model Titan uses the BigTable data model to store graphs, as shown in Figure 14. Each row is uniquely defined by a *vertex id* (the row's *key*). Each vertex stores both all of its *properties* and outgoing *edges*. In the BigTable model, both properties and edges are a row's *cells*. Again, each *property* and *edge* (e.g. each *cell*) contains a *column-value* pair, as shown in Figure 15. This figure shows that the *column* and *value* are built up differently for properties and edges.

An **edge's column** is built up by four elements. The label id is unique within a Titan graph and assigned by Titan itself (e.g. each edge has one edge label, which is defined by this id). Appended to the label id is a single direction bit that defines if the edge is an outgoing edge or an incoming edge. The sort key is defined with the edge's label, and can therefore be used in a graph traversal to find all edges with a certain label (for example one could find all edges with the label "knows", which is a relationship that shows that the two vertices on both ends of the edge know each other). The adjacent vertex id refers to the edge's incoming vertex, and is stored as an offset to the edge's outgoing vertex (e.g. the vertex referenced to by this row's vertex id). Finally, the edge id uniquely defines this edge.

An **edge's value** is built up by two elements. The signature key contains the compressed signature properties of an edge. The second element contains any other properties that are attached to this edge.

A **property's column** is built up by just one element, which is the key id that uniquely defines the property. The **property's value** is built up by two elements, the first being the unique property id and the second being the actual property's value.

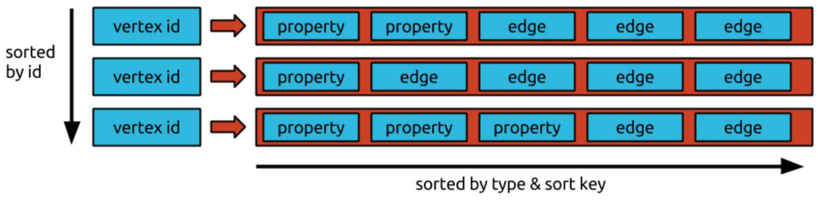


Figure 14: The Titan storage layout, based on the BigTable data model as shown in Figure 13.

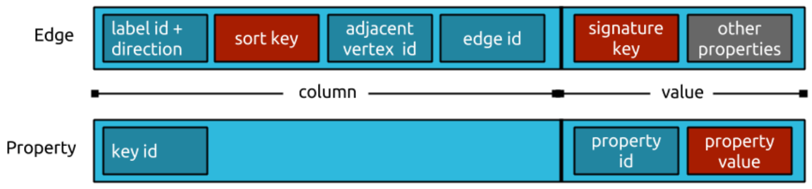


Figure 15: The relation layout inside a BigTable *cell*. shown here for both *properties* and *edges*.

4.2.2 Data storage layer

The data storage layer is the layer between the Titan client and the disk. This layer provides an adapter that tells the Titan client how it should talk to the disk storage. The actual system used in the data storage layer should at least support the BigTable data model. Titan provides native support for a few of these systems, among which are Cassandra, HBase and BerkeleyDB. Any other storage system that supports the BigTable data model can be implemented into Titan by manually writing an adapter for the storage system. The three natively supported systems are globally discussed in the paragraphs below.

Cassandra Apache Cassandra is an open-source noSQL database system, designed to scale over multiple nodes in multiple data centers over multiple geographically separated locations. Nodes in a Cassandra cluster are positioned in a ring. There is no notion of master-slave; each node is homogeneous and equally important. Nodes communicate with each other using the *gossip* protocol. In the gossip protocol, nodes are sending out communication messages to three other nodes in the cluster at a set time interval (for example each second). A communication message contains information about the node itself and about other nodes known by this node. This ensures that each node in the cluster quickly learns about all the other nodes in the cluster, for example if a node is down or if a new node has recently been added.

Figure 16 shows the process of writing new data into a Cassandra cluster. New data is written into two places: a commit log shared by all nodes that contains information about the written data, and a in-memory table called *memtable*. When the configured maximum size of the memtable has been reached, the data that is stored in the memtable is written to (*flushed*) to an *SSTable* (Sorted Strings Table, a key-value store sorted by keys) which is stored on disk. Once data has been written into an SSTable, this table is "closed" and cannot be written to again. Therefore, a node in a Cassandra cluster can maintain multiple SSTables to allow multiple memtable flushes. From time to time, multiple SSTables are written into one larger SSTable, during a process called *compaction*.

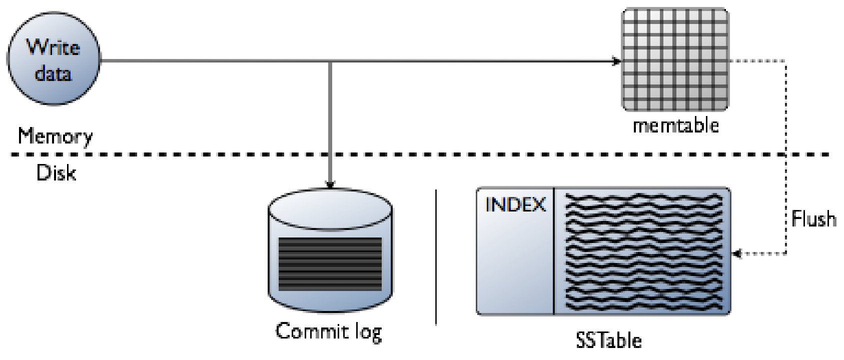


Figure 16: A schematic overview of writing data into Cassandra.

Figure 17 shows the process of reading data from a Cassandra cluster. When a Cassandra cluster receives a request for data, this data could be in multiple places: in the memtable, in one of the SSTables or on one of the larger SSTables stored on the disk. First, Cassandra checks if the memtable contains the requested data. If the data is not present in the memtable, Cassandra checks the *bloom filter* attached to SSTables. These bloom filters can compute the likelihood of the requested data being in an SSTable. If this likelihood is large enough (e.g. higher than a set threshold), then Cassandra checks the sorted keys in the SSTable. If the requested data is found in an SSTable, it is fetched from disk and returned to the user. If the requested data is not found in an SSTable, Cassandra checks the *partition key cache*, which contains the location of the requested data. Using offset information Cassandra jumps to the right location and fetches the data from there (see Figure 17).

Cassandra distributes data automatically over all nodes in a cluster. A tool called the *partitioner* takes care of this distribution. In its default setting, data is distributed randomly over all nodes in the cluster, maintaining an even distribution across the cluster. Data replication is configured by the replication factor, which value defines the number of nodes over which the same piece of data is replicated.

In Titan, Cassandra can run in multiple modes: local server mode, remote server mode, remote server mode with Rexster and Titan embedded mode [34]. In local server mode, Titan and Cassandra run on the same machine and communicate over localhost sockets. In remote

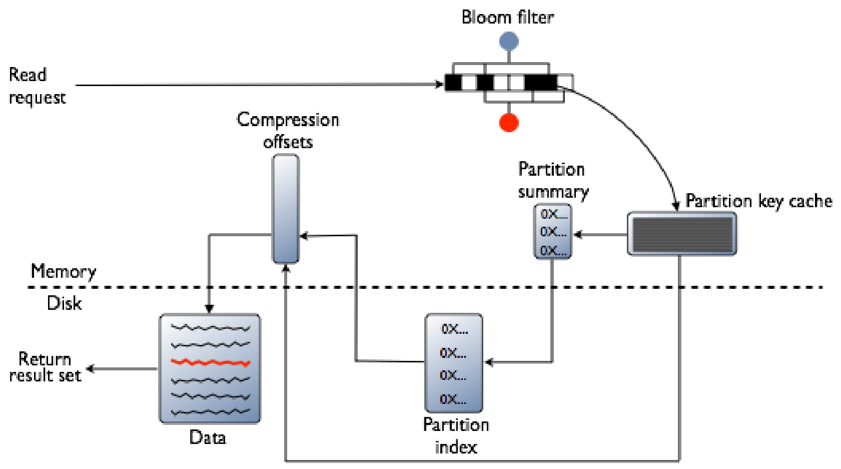


Figure 17: A schematic overview of reading data from Cassandra.

server mode, Titan and Cassandra reside on different machines and are connected to each other using a Titan configuration file that contains the address of the Cassandra machine. In remote server mode with Rexster, a Rexster instance is wrapped around a Titan graph; communication with both Titan and Cassandra is handled by Rexster. In embedded mode, Titan and Cassandra run in the same JVM and communicate with each other using process calls.

HBase Apache HBase is the Hadoop database, storing data in a distributed fashion and enabling scalability [35]. HBase is built upon Hadoop's Distributed File System (HDFS) and provides fast lookups in indexed files that reside on HDFS. Figure 18 shows a schematic overview of the HBase architecture [36]. HBase is based on a master-slave construction, where globally speaking the master coordinates the work which the slaves will execute. The slaves in HBase are called *regions*, and a single region stores part of the HBase tables. Each region consists of two elements: the *memstore* and the *hfile*. When new data is written to HBase, it is first written to the memstore (containing sorted key-value pairs of data). When the memstore is full (or when a signal has been given to clear the memstore), the data from the memstore is written to hfile (also containing sorted key-value pairs) for storage on disk. All new data that is written into HBase is also written into the *write-ahead log* (WAL). Because the WAL contains all data write transactions, it can be used to recover from failure of one of the regions. When the maximum size of a region has been reached, the region is split into two new regions containing parts of the data of the old region. HBase can scale using this splitting strategy.

When a client writes into HBase, it gets in contact with one of the regions. The client will first write to the WAL, registering the write in the HBase system. Afterwards, the data is written into the region's memstore.

When a client sends a read request to HBase, the data to be read could reside in multiple places. Inside a region, the data could be inside the memstore or inside an hfile. Furthermore, recent reads are stored in a *read-cache*. A read request will first search for the data in the read-cache, then in the region's memstore and finally in the region's hfile. If not every data requested is found in the current region (because data could be stored over multiple regions), HBase will use *bloom filters* to find the remaining data. A bloom filter can compute the location of certain data, giving the client handles to retrieve the data from some region [37].

HBase can scale because of the region splits discussed earlier. When a region is split, the two

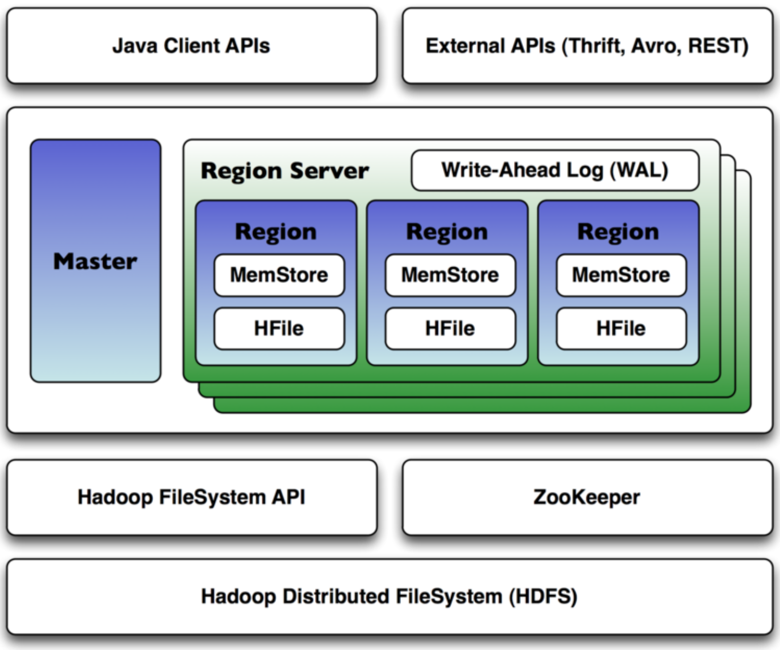


Figure 18: A schematic overview of the HBase architecture.

new regions will be created on the same node as the old region. Later on - for cluster balancing reasons - the HBase master can decide to transfer a region to another node in the cluster. Data replication is defaulted to three locations: new data is written to the local node, to a secondary node and to a tertiary node.

In Titan, HBase can run in multiple modes: local server mode, remote server mode and remote server mode with Rexster. See the details for each of these modes in the Cassandra details mentioned earlier.

BerkeleyDB In contrast to the distributed fashion of Cassandra and HBase, BerkeleyDB is a single-server database system that runs in the same JVM as Titan. Therefore, when using BerkeleyDB a Titan graph cannot be distributed over multiple machines without splitting the graph into multiple, isolated parts. Because a social network is generally not easily partitioned, BerkeleyDB is no good backend solution for this research project.

4.2.3 Indexing layer

The indexing layer is - like the data storage layer - another layer between the Titan client and the disk. Titan has its own indexing backend for composite graph indexes, but needs external indexing backends for mixed indexes. Composite indexes can retrieve vertices by a fixed composition of keys. For example, an index that is composed of two keys can only be used in graph traversals that use both keys - graph traversals with only one of the two keys do not make use of the composite index. In contrast, mixed indexes composed of multiple keys can also be used in graph traversals that use a subset of these keys.

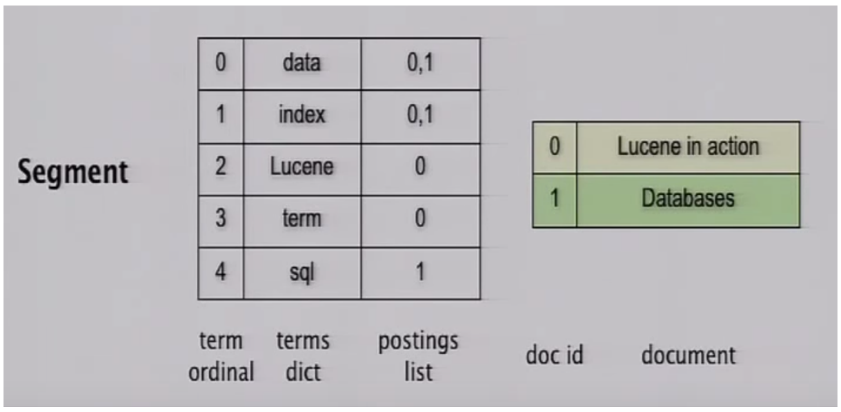
Titan provides native support for three indexing backends: Lucene, ElasticSearch and Solr. These three systems are globally discussed in the paragraphs below.

Lucene A Lucene index consists of a collection of documents. Each document is composed of fields, and each field contains a collection of terms (Strings). Each term is a key-value pair, where the key is the term itself and the value is the documents in which the term occurs. This kind of indexing is known as *reversed indexing*: listing the documents a term is contained in [38]. A normal index would work in the opposite way: listing the terms inside a document. In Titan, a vertex can be seen as a document, a field the set of properties of a vertex, and a term as one of those properties.

Indexes can be separated into multiple segments. For example, when a new document (or in Titan's case, a new vertex) is added to the index, a new segment is created for that document. When the amount of segments in an index gets to big, multiple segments will be merged into one bigger segment. Removing terms from an index is not done immediately. Instead, removed terms will be flagged as 'deleted'. When a segment containing removed terms is being merged with another segments, the removed terms will be removed from the index physically.

Figure 19 shows an example of a segment inside an index [39]. This segment contains two documents, named *Lucene in action* (with id 0) and *Databases* (with id 1). The five terms that occur in both documents are shown in the segment's table. For example, the term *data* occurs in both documents, the term *Lucene* occurs only in the document *Lucene in action* and the term *sql* occurs only in the document *sql.* When a user searches for all documents (in Titan: all vertices) in which the term *data* occurs (in Titan: a property with a certain value), the Lucene index would return the id's of the document (vertices) in which the term *data* occurs, in this case those documents with id's 0 and 1.

In Titan, Lucene can be used for indexing purposes by setting Lucene as the used indexing backend in the Titan configuration.



term ordinal	terms dict	postings list
0	data	0,1
1	index	0,1
2	Lucene	0
3	term	0
4	sql	1

doc id	document
0	Lucene in action
1	Databases

Figure 19: The architecture of the insides of a Lucene Index.

ElasticSearch ElasticSearch is basically a layer around Lucene. It uses Lucene for all indexing and searching purposes, but provides a user-friendly, RESTful API that developers can work with to use Lucene's indexing power. Furthermore, where Lucene is built to be used on a single machine, ElasticSearch supports distributed environments with multiple machines inside a cluster [40].

Figure 20 shows the layout of an ElasticSearch cluster consisting of three nodes. One of the nodes is assigned as the master, and will manage the whole cluster (e.g. add new nodes, remove old nodes, transferring indexes from one node to another, etc.). Each node contains one or more shards. A *shard* is a single instance of Lucene, and holds (parts of the) indexed data. Shards can be replicated over multiple nodes in the cluster. In the figure, the primary shards are indicated by $P1$, $P2$ and $P3$, and each of these shards has two replicas distributed over all nodes in the cluster. These replicas are $R1$, $R2$ and $R3$. When one of the primary shards fails for some reason, one of the replicas can take over the role of the primary shard so data will not be lost.

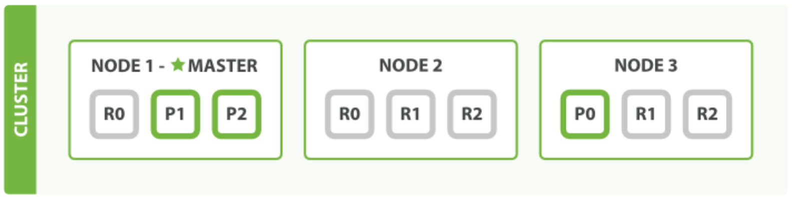


Figure 20: The layout of an ElasticSearch cluster.

In Titan, ElasticSearch can be used for indexing purposes by setting ElasticSearch as the used indexing backend in the Titan configuration.

Solr Like ElasticSearch, Solr is a layer around Lucene, and adds its own functionality on top of that. A Solr cluster consists of a collection of documents, where each document contains fields of a certain type [76]. For example a document about a person could contain a firstname-field (of type String), a lastname-field (also of type String) and a birthday-field (of type Date). All fields are indexed and searchable, which makes it possible to search for all documents (for example all persons) with a certain firstname, or to search for persons who where born before a certain date.

Solr can be used in distributed environments using SolrCloud. The architecture of Solr has much similarities with that of ElasticSearch. although other naming is used. Each single index in SolrCloud is called a *core*, and multiple logically related cores can be combined into a *collection*. Then, collections can be distributed over multiple *shards*, which can be located on multiple machines in the cluster. A normal shard is called a *primary shard*, and any replica that is created based on this shard is called a *replica shard*.

In Titan, Solr can be used for indexing purposes by setting Solr as the used indexing backend in the Titan configuration.

4.2.4 Query capabilities

Titan uses the graph query language Gremlin (which is part of the TinkerPop Stack) as the default query language for Titan graphs. A more detailed description of Gremlin has been given in the related work section of this paper (see section 3).

4.2.5 Remote access

The Gremlin graph query language can be used for local graph traversals, but Titan also supports remote queries using TinkerPop's Rexster. Rexster can be thought of as a 'layer' around a Titan instance, exposing a REST interface with which clients can communicate. Rexster can be implemented on an arbitrary number of machines, creating multiple graph access points to the same graph within one cluster. For example, consider a cluster with two machines, called *machine01* and *machine02*. If Rexster is configured to the same Titan instance on both machines (regardless of which machine contains the Titan instance), then clients can send their graph queries to both *machine01* and *machine02*.

In newer versions of Titan (0.9.0 and above), Rexster is replaced by the Gremlin Server, which is the default for the newest version of the TinkerPop Stack (versions 3).

5 MonetDB

This section provides a detailed overview of MonetDB, including a description of its internals.

Research questions:

- **Architecture**: what is the best way to store Twitter data in a MonetDB data store?
  - How is data stored in MonetDB? How can the current data set be transformed into a format supported by MonetDB? How can the current data set best be partitioned in MonetDB?
- **Architecture**: how does Monet work internally? What is the underlying architecture of Monet (column-store)?
- **Architecture**: how (in what format) does Monet store its data?
- **Architecture**: how can data that is stored in a Monet database be queried (SQL, MAL)? What is Monet's MAL language? Which query language (SQL or MAL) can in theory perform better, given the written queries are optimal?
- **Architecture**: how can clients interact with a Monet database from a remote location? How much latency does this introduce, and is the latency a limiting factor?

This section provides an overview of the MonetDB column store. Section 5.1 provides a broad overview about MonetDB. The architecture of MonetDB is detailed in section 5.2, containing information about query processing with both the SQL query language and MonetDB's own MAL-language, details about how data is stored in MonetDB, and how MonetDB can be remotely accessed.

5.1 Monet overview

MonetDB is a column-store database. Instead of storing data as rows - the way in which data is stored in most relational database systems - MonetDB stores data as columns. For example, see the example data in Table 1. This table shows how a row-oriented database systems would store its data: each row contains one object with all of its properties (defined by the three columns *id*, *firstname* and *lastname*.

id	firstname	lastname
1	John	Smith
2	Ronald	Waterman
3	Jessie	Pinkman

Table 1: Example row-based data store

In a column-oriented data store, data is not stored in rows, but in columns. Each column shown in Table 1 will be stored in its own file, which contains key-value pairs with the key being the column's contents, and the value being references (by id) to the object to which this content belongs. For an example, see Table 2. The contents of this table show how the *lastname* columns would be stored: the lastname itself as the key and a pointer to the object id as the value.

5.2 Monet architecture

The following subsections describe MonetDB's architecture in more detail. A broad overview of MonetDB has already been given in the *related work* (see section 3.6.1).

Smith	1
Waterman	2
Pinkman	3

Table 2: Example column-based data store

5.2.1 Query processing

Queries are processed by MonetDB using three software layers: the parsing layer, the optimization layer and the interpreter layer. The first layer (parsing) provides adapters that can translate queries from a high level language (like SQL, or SPARQL) into MonetDB's own query language, the *MonetDB Assembler Language* (MAL). The second layer (optimization) is a collections of optimization modules. For example, this layer ensures that the MAL programs that were created in the first layer are optimized to be as efficient as possible. Finally, the third layer (interpreter) uses various modules to interpret the optimized MAL program, and use the right interpreter for the right MAL statements.

As has been stated earlier, MonetDB uses its own language MAL for query execution. However, because MonetDB contains parsers that can transform higher level query languages into MAL, it is also possible to use SQL (or another supported language) with MonetDB. For this project, MonetDB is uses in combination with the SQL query language. The two next paragraphs provide some general information about using SQL and MAL with MonetDB.

SQL MonetDB supports SQL since 2002, and is largely based on the SQL'99 [67] and SQL'03 [68] definitions. However, the actually supported definitions depend largely on the requirements of MonetDB's user base. Therefore, some SQL definitions that are defined in the '99- and '03 definitions are not supported in MonetDB, and some of the language constructs supported by MonetDB are not contained into the '99- and '03 definitions. A full overview of SQL definitions supported by MonetDB and how to use them can be found in MonetDB's SQL Reference Manual[12].

MAL While users can use MonetDB with high-level query languages like SQL and SPARQL, the actual execution of queries on a MonetDB instance is performed using the *MonetDB Assembler Language* (MAL). It is possible to program queries in MAL, although it is not encouraged to do so [41]. A full overview of MAL (including for example its syntax, interpreters, optimizers and other modules) can be found in the MonetDB Internals section on MonetDB's website[13].

5.2.2 Data storage

All data (all columns) in MonetDB is stored in what are called *Binary Association Table*s (BATs). A BAT consists of $< surrogate, value >$ tuples, where the *surrogate* (the *head*) is a virtual id, which can be thought of as the array index of a BAT. The *value* (the *tail*) is the actual content of a column, in the example sketched in Table 2 the values of the BAT would be last names. Figure 21 shows the architecture of BAT tables, and how they interoperate with higher level languages like SQL, XQuery and SPARQL [65]. Every query that is fired onto a MonetDB database is transformed into the *MonetDB* Assembler Language (MAL), which can communicate with BAT tables. For example, a query in one of the higher level languages could ask for all objects of which the year of birth is equal to 1927. In MAL, such a query would be transformed into a

[12]https://www.monetdb.org/Documentation/SQLreference
[13]https://www.monetdb.org/Documentation/MonetDB/Introduction

MAL statement: $select(byear, 1927)$. The $select()$ function consists of two arguments, the first being the BAT table to search in, and the second being the value to search for. In this case, the BAT table $byear$ contains two values which are equal to 1927, namely those with IDs 1 and 2. These results are returned in yet another BAT table. Using the returned IDs, the names of the people to which these IDs belong can be fetched from the $name$ BAT table using these IDs.

BATs are stored in two memory arrays: one for the surrogate (head) and one for the value (tail). In case the value stored in the tail array are of variable width, the tail array is split into two arrays: one containing offset values, and the other containing the actual content pointed to by the offset values in the first array. When the values in the head array are densely packed and ascending (e.g. when the values are just indexes: 0, 1, 2, ...),then the head array can be omitted and retrievals from the tail array can be made by just referencing to an index value [66].

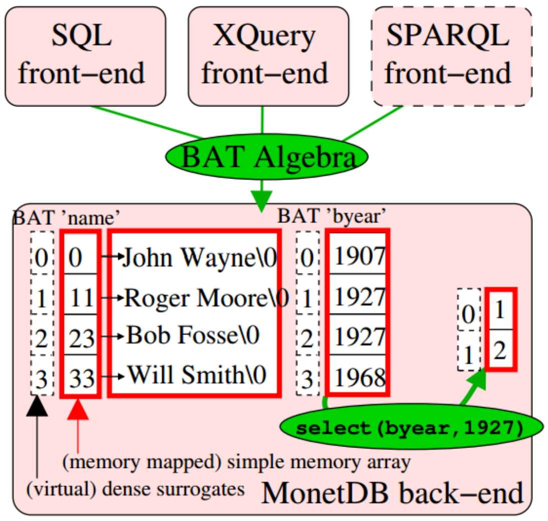


Figure 21: The architecture of MonetDB's BAT tables.

5.2.3 Remote access

MonetDB is running locally on a machine, but can be accessed remotely by setting up a MonetDB server. A MonetDB server can be started by initializing a MonetDB daemon (managing a database farm), using the $monetdbd$ command. The command $monetdbdcreate < local - dbfarm - filepath >$ creates a new MonetDB database farm on the local file system. After setting up the database farm, it can be started using the command $monetdbstart < local - dbfarm - filepath >$.

When the database farm is running, new databases can be added to the farm using the $monetdb$ command (without the trailing 'd'). For example, to create a new database called $twitter$ one would issue the command $monetdbcreatetwitter$. The new database is created in maintenance mode, so that the user can first setup the database before making it accessible. When all initialization has been performed, the new database can be started with the command $monetdbstarttwitter$. From then on out it is accessible remotely through the MonetDB server.

There exist various language bindings to connect to a MonetDB server instance, either locally or remotely. At the moment of writing, MonetDB supports JDBC, ODBC, PHP, Perl, Ruby and Python interface libraries. For example, a connection to a remote MonetDB database can be setup in JDBC using the following command:

```
Connection con = DriverManager.getConnection(
  "jdbc:monetdb://127.0.0.1:54321", "monetdb", "monetdb");
```

The first argument to the *getConnection*() function is the (remote) address and port number at which the MonetDB database runs. The second and third argument are the username and password used for authentication.

6 Test data generation

In order to benchmark both Titan and MonetDB, we need to have a data set. A good source of data is the data generator of LDBC. The advantage of this data set is that is represents a social network, and is therefore similar to the Twitter data set from OBI4wan.

Also interesting to write about here, or in the 'related work' section: other projects that (dynamically) create social networks to use in graph comparison studies like this.

Research questions:

- **Architecture**: what is the best platform to work on, both for storing the data and execute the benchmarks using Titan and MonetDB (for example a cluster on Amazon, a server at the CWI, multiple servers from OBI4wan)?
  - What are the differences between the platforms? Advantages? Disadvantages? If not, what can we do to overcome this problem (e.g. write plugins, use workarounds)?
- **Data**: the LDBC data generator can generate a data set that is a representation of a real social network. What does this data set look like? Which entities are created, what properties do they have and what relationships between entities exist?
- **Data**: how is the data set from the LDBC data generator generated (e.g. how does the data generator work internally)?
- **Data**: how can output from the LDBC data generator be used as input for both database solutions (Titan and Monet)? What is the best format into which the LDBC data can be translated (if multiple formats are supported by the database solution)?
- **Data**: how long does it take to load data of different scale factors into the database solutions? Does this time increase linearly with the data size?

In section 6.1, the need for extra test data apart from the real Twitter data from OBI4wan is detailed, along with some examples of programs or data stores that can deliver this test data. The LDBC DATAGEN is the test data generator that is used in this research. Section 6.2 describes how test data is generated using the LDBC DATAGEN. Section 6.3 shows how the output from the LDBC DATAGEN is transformed so that it suits the input format required by the databases under test. Section 6.4 shows the process of loading the transformed test data into the databases under test.

6.1 In need of test data

In order to compare how well database systems perform on the storage of graph data and the querying on that graph data, we first need to create such data. The Twitter data collected by OBI4wan is the primary source of graph data, but it is not complete in its raw format. For example, the Twitter data does contain information about how many followers and friends a user has, because this number is attached to each tweet. However, who these followers and friends are (e.g. the actual followers- and friends relations) are not contained in the raw tweet data, which means that this data has to be fetched separately. Because manually retrieving data about a user's followers and friends using the official Twitter API is a very long process, another data set that does contain all necessary information for querying graphs is needed. There are various sources that provide this kind of data, for example the collection of real social network data at the Stanford Network Analysis Platform (SNAP)[14], or (social) network data generators like

[14]http://snap.stanford.edu/data/index.html#socnets

Lancichinetti-Fortunato-Radicchi (LFR)[15] and LDBC DATAGEN[16]. Among other data, the two network data generators LDR and LDBC DATAGEN contain person vertex data and relationship edge data between this vertex data, filling the missing data gap of the raw Twitter data set as collected by OBI4wan.

6.1.1 Stanford Network Analysis Platform (SNAP)

The Stanford Network Analysis Platform (SNAP) contains a collection of categorized data sets. One of them is the *social network* category, which contains data from various social networks like Twitter, Facebook and Google+. The data sets consist of various files, containing vertices with attributes and edges (relations) between those vertices [26].

6.1.2 Lancichinetti-Fortunato-Radicchi (LFR)

The Lancichinetti-Fortunato-Radicchi (LFR) method is a benchmark for community detection, and tries to create a social graph network that resembles real social network graphs as closely as possible [54]. The algorithm for generating graphs is centered around communities. A vertex is given a degree and is assigned to a community based on an exponent from the power law distributions γ and β, respectively. Power law distributions are chosen because research has shown that real world social graph networks also seem to be following them when it comes to degree distribution of vertices [54] [56] . Furthermore, $1 - \mu$ of edges from a vertex are connected to another vertex within the community, and μ edges from that vertex are connected to another vertex not in the community [55].

6.1.3 LDBC DATAGEN

The LDBC DATAGEN generates a collection of people and their attributes (for example their name, gender and birthday), plus their activity in a social network (for example the posts and comments that these people post in certain forums, the places they live in, the organizations they work for and the knows-relations [one persons *knows* another person] they have with each other) [52]. The social network is built in such a way that attributes of an entity are correlated and also influence the relationships with other entities.

6.1.4 Choosing the secondary data source

The final choice for LDBC DATAGEN - and not for another data generator - needs argumentation.

6.2 Generating test data

The LDBC DATAGEN generates a social network based on the persons in that network, and their attributes. The person's attributes are correlated, and will also influence the relationship between a person and other entities. For example:

- the place where a person lives influences that persons name and the interests of that person.
- a person can only comment on another person's post if they have become friends earlier, and persons can only become friends if they are both registered to the social network.

[15]https://sites.google.com/site/andrealancichinetti/files
[16]https://github.com/ldbc/ldbc_snb_datagen

- posts and comments are influenced by real world events. For example, when a political election takes place, people will talk about this election in their messages more often than on average (e.g. the topic is trending).
- people who are "similar" (e.g. similar interests, living in the same place, working for the same organization, etc.) have a higher chance of being friends.

The creation process of a social network consists of three phases. Phase one is the *person generation*, in which all persons and their attributes are created. Phase two is the *friendship generation*, in which friendship relations between persons are created (based on the persons attributes as mentioned earlier). Finally, phase three is the *person activity generation*, in which all posts and comments created by persons are created.

The LDBC DATAGEN can output social networks of different sizes. The total size of the network is based on a given number of persons, starting year and total number of years, or on a scale factor which references the approximate data size of the network in gigabytes (e.g. SF1 contains 1GB of data, SF3 contains 3GB of data, etc.). The output of the LDBC DATAGEN is split into CSV-files that contain vertices and their attributes, and CSV-files that contain edges (relationships) between vertices, potentially containing edge labels such as the creation date of the edge.

When executing a benchmark to test the quality of a database system, it is necessary to test the system with different data sizes in order to test how scalable the system is. The LDBC DATAGEN option to generate a data set with a specific size therefore comes in handy.

6.3 Translating test data to database input

6.3.1 LDBC DATAGEN output format

The LDBC DATAGEN can output data in CSV format or in Ntriple (more verbose) format. We will use the CSV format, as the extra data from the Ntriple format will not be used in this research. The CSV-files are split into files containing vertices and their attributes, and files containing the edges between vertices, potentially containing edge labels. Examples of both file types can be found in Figures 22 and 23.

```
16492674424341|David|Smith|female|1985-11-21|2012-09-30T06:55:10.913+0000|
11.6.91.153|Firefox|
```

Figure 22: Output CSV-file containing information about a vertex (in this case a person). The attributes are ID, first name, last name, gender, birthday, creation (registration) date, IP address and used browser.

```
16492674424341|16492674424863|2012-09-30T08:13:06.697+0000|
```

Figure 23: Output CSV-file containing information about an edge (in this case a "knows"-relation between two persons). The attributes are ID person A, ID person B, creation date (of relation).

6.3.2 Translation scripts

Different databases generally use different data formats for loading data into databases. The same is true for the two database systems used in this research: Titan and Monet. So, before we can use the data that is output from LDBC DATAGEN, we need to translate this data into a format that is compatible with the respective database systems. The following paragraphs show this translation process for both Titan and Monet.

Titan supports a variety of data input/output (IO) formats, which are the SequenceFile format, special Titan formats, GraphSON format, EdgeList format, RDF format and Script format. A short description of these formats is given in the list below.

SequenceFile A *SequenceFile* is the native binary file format that is used by Hadoop. The *FaunusVertex* and *FaunusEdge* objects provided by Titan both implement Hadoop's *Writable* interface, which means they can be captured by a *SequenceFile* [28].

Special Titan Titan supports both Cassandra and HBase as its database backend. Special Titan IO formats can be used to read and write from one of these back-ends: *TitanCassandraInput/OutputFormat* and *TitanHBaseInput/OutputFormat* [29].

GraphSON The *GraphSON* format is a vertex-centric input format: each line contains one vertex with its attributes and a list of all vertices that have a relation with this vertex [30].

EdgeList The *EdgeList* format is a plain text file, where each line contains one edge. Each line consists of the source vertex ID, the sink vertex ID and an edge label. Note that this format only stores the ID of a vertex, and no other attributes. Also, an edge can have only one edge label when using the *EdgeList* format [31].

RDF The *RDF* format consists of triples with a subject and an object, connected to each other by a predicate. It is similar to the *EdgeList* format, but vertices in *RDF* point to the vertex location, which can hold all attributes and other information for that vertex [32].

Script The *Script* format is a special format used by Titan. The user has to define its own script that will be used when loading data into Titan. An input file is read line by line, and the script's content parses each line and creates vertices and edges based on the user-defined format.

The raw CSV-files that are output by the LDBC DATAGEN are not directly usable with one of the Titan IO formats, which makes the user-defined *ScriptInputFormat* the best choice for writing data into a Titan database. The output format depends on the back-end that is used for the Titan database, and can be either *TitanCassandraOutputFormat* or *TitanHBaseOutputFormat.*

6.3.3 Translating LDBC DATAGEN output into Titan Script IO format

The user-defined input format that is used in this research is detailed in a BitBucket repository[17]. A Pig script is used to translate the LDBC DATAGEN output format to the user-defined format that is used by Titan's *Script IO.* All relations types, property (attribute) types, entity types and data types that are present in the LDBC DATAGEN output are put into separate enumerations, resulting in smaller output files. The exact relation-, property-, entity- and data types that the Pig script has to take into account must be written down in a special schema input file. In the output files, each line contains exactly one vertex with all of its attributes and relations to

[17]https://bitbucket.org/RenskeA/test/wiki/Home

other vertices. The total number of output files is equal to the total number of vertex types - all vertices of the same type are put into the same output file.

The example output line in Figure 24 is built up as follows.

- One the first line, we have the tuple (0,1,173). The 0 means this tuple is an ID (data type), the 1 means that it is a person (entity type) and the 173 is the actual ID.
- The next item is a quadruple (4,6,2,Zheng). The 4 means that this quadruple is a property, the 6 means that it is the "first name" property (property type), the 2 means that the property is a string (data type), and "Lei" is the actual property value.
- The following five items (continuing on the second line in Figure 24) are also quadruples containing vertex attributes: the gender (female), birthday (1989-05-06), creation date (2010-02-28T07:31:07.363+0000), IP address (27.50.135.189) and used browser (Internet Explorer).
- The first item on the third line is (3,1). The 3 means that this is a relation, the 1 means that the relation is a friend (relation type). The following list contains the persons' friends, for example the tuple (0,1,3298534892049), where the 0 means this tuple is an ID, the 1 means that it is a person and 3298534892049 is the actual ID.
- The last line also contains relations, but now of the enumeration type 4, which is a like. The next tuple contains the vertex where this relation points to (with ID 206158956528). The last quadruple is an edge label of the date type, with the value 2012-03-16T04:18:17.605+0000.

```
(0,1,173)|(4,6,2,Zheng)|(4,7,2,Lei)|(4,8,2,female)|(4,9,1,1989-05-06)|
(4,1,1,2010-02-28T07:31:07.363+0000)|(4,2,2,27.50.135.189)|(4,3,2,Internet Explorer)|
(3,1)|(1,3,{((0,1,3298534892049)),...})|
(1,4,{((0,0,206158956528),(4,1,1,2012-03-16T04:18:17.605+0000)),...})
```

Figure 24: An example output line created by a Pig script that translates the output from LDBC DATAGEN into a user-defined output format.

Monet Loading the LDBC DATAGEN output into Monet is relatively simple. Monet supports SQL's *COPY INTO* statements, which can use a CSV-file as input for a SQL database table. The database table to which the content of the CSV-file is copied must have the same number of columns and their respective data types as the content in the CSV file. For example, to copy the line from Figure 22 into a Monet database table, this table must have six columns (for each of the six attribute values): an *ID* column of type *BIGINT*, a *firstname* column of type *VARCHAR(x)*, a *lastname* column of type *VARCHAR(x)*, a *gender* column of type *VARCHAR(6)*, a *birthday* column of type *DATE* and a *creationdate* column of type *TIMESTAMP*. The resulting *COPY INTO* statement has the following syntax:

```
COPY INTO <tablename> FROM '<filename>';
```

6.4 Loading test data into databases

When the data is in the right input format for a database system, the next step is to start the actual loading process. Loading data into Monet with the *COPY INTO* statements does not require any extra work other than just executing the statement, but Titan requires more work. The next subsection will describe the process of loading data into Titan using the database's *ScriptInputFormat* in combination with Hadoop.

6.4.1 Loading data into a Titan database

Section 6.3.2 has shown the process of translating the LDBC DATAGEN data into a user-defined format that can be used to load the data into a Titan database. This process has created output files for every entity type, in this case for persons, posts, comments and tags. Titan's *Script IO* format uses these files as input and reads them line by line. A user-defined script parses each line and performs all necessary operations. The script that is used for this research takes the following steps for each line in the input files.

1. Each vertex has an ID, but ID's are not unique over all vertex types. Therefore, we use a special ID for each vertex in Titan. Under the radar this special ID is just another vertex attribute which is indexed, and is of the form "¡vertex type¿-¡vertex id¿" (for example: "person-12345"). The first step of the script is to create a new Titan vertex with its ID as first attribute.

2. Next, the script iterates over the next tuples/quadruples, which can be of type 'relation', 'label' or 'property'.

    (a) A **relation** always exists between two vertices. The outgoing vertex is already known, because it is the same vertex which line is currently processed. The other vertex is retrieved by its ID, and the function $addEdge(direction, type, ID)$ is used to add the new edge. If the new edge contains an edge label, this label is attached to the new edge object using the function $setProperty(key, value)$.

    (b) A **label** specifies the vertex type (person, post, comment or tag). It adds this label as a normal vertex property.

    (c) A **property** is an attribute that is attached to a vertex, using the function $addProperty(key, value)$.

7 OBI4wan data usage

The real Twitter data comes from the OBI4wan archives. This section can be used to describe this data, and how to translate it to input for both database systems (Titan and MonetDB). The starting point is to use the exact same data format as the format from the LDBC data generator, because then the same scripts for transforming and loading data into the databases can be used.

Research questions:

- **Data set**: given the architecture of the OBI4wan data set, what is the fastest, most efficient, best way to transform the data set in its current format (stored in ElasticSearch) to a graph-based format that can be used by property graph databases (Titan)?
  - Is it possible to transform the current data set to a graph format, and still use the same ElasticSearch full-text search queries that are used now (e.g. is only a graph database enough, or is there still a need for an extra ElasticSearch store besides the graph database)?
  - In what way is ElasticSearch integrated into Titan? Can ElasticSearch still be used in the same way as it is used now, or is this not possible in Titan (e.g. is ElasticSearch fully implemented, or only partly)?
- **Architecture**: what is the best way to store Twitter data in a MonetDB data store?
  - How is data stored in MonetDB? How can the current data set be transformed into a format supported by MonetDB? How can the current data set best be partitioned in MonetDB?
- **Architecture**: what is the best platform to work on, both for storing the data and execute the benchmarks using Titan and MonetDB (for example a cluster on Amazon, a server at the CWI, multiple servers from OBI4wan)?
  - What are the differences between the platforms? Advantages? Disadvantages?
  - If not, what can we do to overcome this problem (e.g. write plugins, use workarounds)?

This section provides information about how the Twitter data from OBI4wan is used in the benchmarks executed in this research. Section 7.1 shows the structure of the Twitter data as it is stored by OBI4wan. Section 7.2 shows the process of translating OBI4wan data into an input format that is accepted by the databases under test. Section 7.3 shows how a missing part of the OBI4wan data - the friends- and followers set of the top users from the OBI4wan data set - is constructed, using the Twitter API to retrieve this data (also see section 2.1.1). Section 7.4 provides an analysis about three graphs that were generated from OBI4wan data and the extra friends- and followers data, in this way providing information about the structure of the OBI4wan data set. Finally, section 7.2 describes the process of loading the translated OBI4wan data into the databases under test.

7.1 OBI4wan data format

The Twitter data archive of OBI4wan consists of gzipped archives, where each month is split into three parts to reduce the size of the individual archive files. For example, in November three archive files are created: one for November 1st until November 10th (named *2014nov1.gz*), one for November 11th until November 20th (*2014nov2.gz*) and finally one for the rest of the month, from November 21st until November 30th (*2014nov3.gz*).

Each line in the archive contains exactly one tweet in JSON format. An example tweet is shown in Figure 25. The attributes contained in the JSON are explained below.

id	message ID given by Twitter.
user	accountname of the user that posted the tweet.
content	message content
published	timestamp from the moment the message was posted on Twitter
sentiment	sentiment value (enumeration) given to the message based on its content
sourcetype	platform on which the message was posted
language	language of the message's content
friends	number of this user's friends at this message's publish timestamp
followers	number of this user's followers at this message's publish timestamp
statuscount	total number of messages posted by this user at this message's publish timestamp
loc	location from which this message was posted
source	application from which this message was posted
hashtags	hashtags used in this message (tags preceded by the official hashtag symbol (#)
mentions	users mentioned in this message (names preceded by the official mention symbol (@)
inreplytoid	ID of the parent message to which this message is a reply (or -1 if this message is an original status update)
posttype	message type, being STATUS (original message), RETWEET or REPLY
url	exact URL that leads to this message on Twitter

```
2014nov3/test/535648671644536832 {"id":"535648671644536832", "user":"kimber33",
"content":["@ter808 so I'm catching up on #TheVoice & someones singing I wanna
dance w somebody, to get saved by America... Clearly my vote goes to her"],
"published":"2014-11-21T04:19:29.000Z", "indexed":"2014-11-21T04:22:21.479Z",
"sentiment":1, "sourcetype":"twitter", "language":"EN", "rank":1, "friends":295,
"followers":45, "statuscount":7376, "loc":"Rhode Island",
"source":"Twitter for iPhone", "hashtags":["thevoice"], "mentions":["ter808"],
"inreplytoid":"-1", "accountname":"" ,"posttype":"STATUS",
"url":"http://twitter.com/kimber33/status/535648671644536832/","host":"twitter.com"}
```

Figure 25: Example tweet from the OBI4wan archive, in JSON format.

7.2 Translating OBI4wan data to database input

Before the OBI4wan Twitter data can be put into Titan and Monet, it needs to be translated into a format that is supported by these two database systems. The most efficient solution is to reuse the translation scripts that have been used to translate the data from the LDBC DATAGEN into a format that is readable by the two database systems. The task that then remains is to translate the OBI4wan Twitter data into the same format as the LDBC DATAGEN output format. In general, the LDBC DATAGEN data output format can be considered as an efficient format, because it consists of vertex-files (including vertex properties) and edge-files (including edge properties), which is equal to the kind of format graph databases use to store data.

A Python script is used to translate the OBI4wan Twitter data into the LDBC DATAGEN data format. The process of this translation is described in the list below.

1. The Python script accepts an OBI4wan gzipped archive file, which contains tweets from a certain time frame. The lines in this archive file are read one by one, with each line containing the JSON representation of a tweet. The contents of this line are described in section 7.1.

2. The tweet's JSON is being processed by two functions, being *create_vertices*() and *create_edges*(). Based on the tweet's JSON, these two functions will create the same files as those that are output by the LDBC DATAGEN: files containing vertices and their attributes and files containing edges, potentially containing edge labels. The inner workings of both functions are discussed in the next steps.

3. The function *create_vertices*() creates one file per vertex type (in this case persons, posts, comments and tags), each file containing one vertex and its attributes per line. For each vertex type, the information for a vertex is contained in the tweet's JSON. For example, a vertex of type 'post' can retrieve the post's publish timestamp, language and content directly from the tweet's JSON.

4. The function *create_edges*() creates one file per edge type, each file containing one edge and potentially its attributes per line. Again, all information that is needed for the edges is contained in the tweet's JSON. For example, an edge of type 'comment_post' (e.g. a comment that is a reply on a post) can retrieve the comment ID (which is the tweet's ID) and the post to which the comment is a reply (which is the tweet's 'inreplyto' ID) from the tweet's JSON.

When all tweets from the OBI4wan Twitter archive are processed, all vertex- and edge files have been filled with content originating from tweet's JSON. However, there is one piece of information that is not contained in the tweet's JSON, and that is the followers/friends (sub)graph. Each tweet contains the number of followers and friends that the user who posted the tweet has at the moment of posting, but who those followers and friends are is not known. The next section shows how this information has been retrieved, using the official Twitter API.

7.3 Creating the friends/followers (sub)graph

Tweets from the OBI4wan archive contain information about the number of friends and followers of the user who posted the tweet, but no information about who these friends and followers are. To get this information, we need to use the Twitter API again with the username or ID of a user in order to receive a list of friends and followers of that user. The Twitter API allows to retrieve those lists for one user per call, returning at most 5000 friends or followers per user. When this information is stored for all users that occur in the OBI4wan Twitter data set (e.g. all users who have posted one or more tweets that have been collected by OBI4wan), a friends/followers graph can be constructed.

The API call to receive a user's friends/followers is rate-limited at 15 calls per 15 minutes (e.g. 1 call per minute). So, with one authentication token, it is possible to receive at most 5000 friends and 5000 followers for one user per minute[18]. With each extra authentication token, the amount of allowed calls doubles. For example, with 20 authentication tokens, one can call the Twitter API 20 times per minute, increasing the rate at which the followers/friends graph can be created.

The total number of unique users per OBI4wan Twitter archive file is 3.5 million on average. With 20 authentication tokens, and the assumption that only the first 5000 friends and followers of a user will be retrieved, it still takes 122 days (3.000 hours, 175.000 minutes) to retrieve the first 5000 friends and followers for all users. This is a very long time, especially considering the fact that the followers and friends of a user can change over time. With this rate, a user's friends and followers are only updated once every 122 days. When real-time and up-to-date information is required, this method of creating a friends/followers graph is not good enough, but for statistical purposes this data set is still interesting to use. For this research the friends/followers graph is needed because some benchmark queries depend on the information. However, it is not required that this information is up-to-date for benchmarking purposes, so also an out-of-date friends/followers graph is good enough. For this research, only the top 1 million users have been taken for the friends/followers graph, sorted descending by how many tweets each user has posted in an OBI4wan Twitter archive.

7.3.1 Program execution

The execution of the program that retrieves a user's friends and followers is described below.

1. The program accepts an input file with on each line a Twitter username, in this case the file contains a total of 1 million usernames.

2. Before retrieving followers and friends using the Twitter API, the program sets up the environment that is required to actually use the Twitter API. This setting up requires another input file, which contains authentication tokens from Twitter users that are attached to the application. These tokens can be used to make calls to the Twitter API. The more tokens are in the input file, the more requests one can make to the Twitter API per time interval.

3. Based on a Twitter username, the program uses the *Twitter4J* library[19] to retrieve the followers and friends of the user, with a maximum of 5000.

4. After the friends and followers of a Twitter user have been retrieved, they are written to an output file in JSON format. The JSON contains an attribute pointing to the Twitter user, and two lists containing the IDs of this user's friends and followers. An example line in the output file is shown in Figure 26.

[18]In order to let users with a lot more friends and followers not become a bottleneck, only the first 5000 friends/and followers are retrieved for each user.

[19]http://twitter4j.org/en/

```
{"name":"esrayucel571","followers":[3032153789,2810764899,2857458725,...],
"friends":[252601950,365260141,1689115598,...]}
```

Figure 26: Example line from the friends/followers JSON file.

7.3.2 Translating friends/followers JSON to user relations file

The last step of translating the OBI4wan archive data format to the LDBC DATAGEN data format is to translate the friends/followers JSON output to the *person_person* edge relation file (e.g. person A *knows* person B). This translation is relatively simple: for each line in the friends/followers JSON, we need to output a line for each relation between the current user and one of his followers, and a line for each relation with one of his friends. The example line as shown in Figure 26 would translate into the *person_person* relations file as shown in Figure 27.

```
3032153789|esrayucel571|knows
2810764899|esrayucel571|knows
2857458725|esrayucel571|knows
esrayucel571|252601950|knows
esrayucel571|365260141|knows
esrayucel571|1689115598|knows
```

Figure 27: Example lines from the *person_person* relations file.

7.4 OBI4wan graph analysis

To obtain a better understanding of the OBI4wan data, we can analyze it. Detecting communities is one of the most relevant methods of analyzing (social) graphs [58]. A *community* can be defined as a set of vertices which have many connections among them, but relatively few connections with other vertices in the graph. Among others, community detection can provide information about how the network is structured, how vertices (for example persons) interact with each other and which role these vertices (persons) have in a network (for example, some person could be the connecting link between two communities, some person could be the center of a community [containing links to all other vertices in the community,] etc.).

There exist a variety of community detection algorithms, of which *Scalable Community Detection* (SCD) seems to be the most promising, having the fastest execution time compared to other algorithms like Walktrap [59], Infomap [60], Oslom [61], Link Clustering Algorithm (LCA) [62] and BigClam [63].

The SCD has been used in [64], where the structure of communities in real graphs (like Amazon and YouTube) have been compared to communities in graphs generated by graph generators, such as LFR[20] and LDBC DATAGEN[21]. Using SCD, the following five structural indicators have been calculated: (global) clustering coefficient, triangle participation ratio (TPR), bridge ratio, diameter and conductance. These five structural indicators are now discussed shortly.

[20]https://sites.google.com/site/andrealancichinetti/files
[21]https://github.com/ldbc/ldbc_snb_datagen

Clustering coefficient	The clustering coefficient is a number between 0 and 1, where 0 means that there is no clustering at all in a graph (or community), and 1 means that the graph (or community) contains only disjoint clusters (which are not connected to each other but only contain connections to other vertices within the cluster). The clustering coefficient is defined more formally by the probability that given two edges that share a vertex, there is a third vertex that closes a triangle (e.g. edges A and B can share a vertex X, edge A is also connected to vertex Y, edge B is also connected to vertex Z, and a third edge C between Y and Z closed the triangle). See Figure 28
TPR	Given a set S and a vertex x, the triangle participation ration (TPR) is defined by the number of triangles in S that x is contained in.
Bridge ratio	Given a set S with vertices (an induced subgraph of a community) and an edge $e \in S$. Edge e is called a *bridge* if e disconnects the set S (e.g. the community) from the rest of the graph. The bridge ratio is calculated by finding the set of neighbours of every vertex in S - thereby retrieving all edges between vertices in S and between vertices in S with vertices outside S - and calculating how many of those edges are *bridge* edges.
Diameter	The diameter of a graph (or community) is defined by the maximum number of steps that is needed to travel from one vertex to another in that graph. In other words, it is the "longest shortest path" between two vertices in a graph.
Conductance	A community's conduction can be thought of as the level of isolation of a community from the rest of the graph [64]. Another way of thinking about conductance is to consider a random walk that starts at some vertex in a community. When the community is well-connected, it will take longer for the random walk to step out of the community, because the chance of traversing to another vertex inside the community is higher than the chance of traversing to a vertex outside the community. The longer the random walk will stay in the community, the more isolated the community is and the higher the value for conductance of this community will be. More formally: given a graph G and a set of vertices S which is a subgraph (community) of G, the conductance can be calculated by dividing the number of edges leaving S by the summed number of neighbours for every vertex in S.

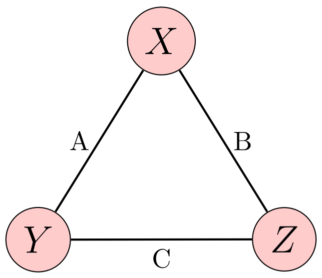


Figure 28: Example of a graph triangle

The implementation of SCD can be found on their GitHub repository[22], and works as follows:

1. Given a graph in unique EdgeList format (the same edge will appear only once), the SCD algorithm first finds all communities in that graph. If the EdgeList of the graph is contained

[22]https://github.com/DAMA-UPC/SCD

in the file *network.dat*, then finding the graph's communities is done by executing the following command:

```
./scd -f ./network.dat
```

The SCD algorithm is based on finding triangles in the graph and creating communities based on those triangles. However, there exist vertices which do not participate in triangles. The SCD algorithm puts these vertices in their own community of size one, which may pollute the final results. Therefore - after the communities have been detected by SCD - all communities with size less than three (e.g. all communities with vertices which are not participating in any triangle) are removed from the community set.

2. Once the communities of the graph have been found, SCD's *CommunityAnalyzer* can be executed to calculate all structural indicators discussed earlier. Given the EdgeList of the graph in file *network.dat*, the calculated communities in *communities.dat* and an output file to write the results to in *output.csv*, the following command will calculate the structural indicators:

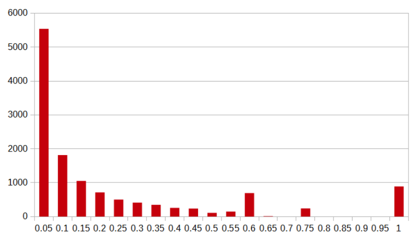
```
./communityAnalyzer -f network.dat -p communities.dat -o output.csv
```

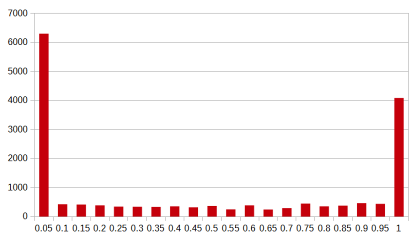
The *SCD*- and *CommunityAnalyzer* algorithms output a CSV-file, where each line contains the structural indicator results for one of the communities of the input graph. The next three subsections present the results of the *CommunityAnalyzer* algorithm for three graphs: the friends/followers graph in section 7.4.1 - of which the construction and contents has been discussed earlier, the "mentioned" graph in section 7.4.2 (containing a relation between person A and person B if person A has mentioned person B in a tweet) and the "retweeted" graph in section 7.4.3 (containing a relation between person A and person B if person A has retweeted a tweet that was posted by person B).

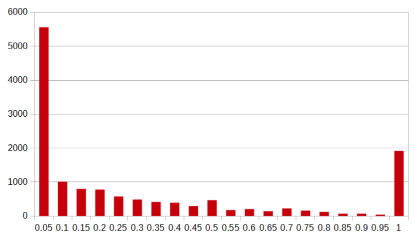
7.4.1 Friends/followers graph statistics

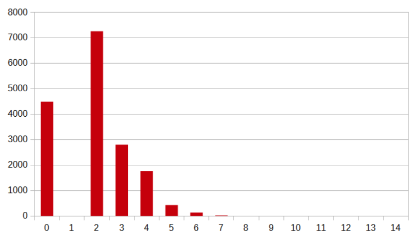
First, we will analyze the results from executing SCD's algorithm on OBI4wan's friends/followers graph, as shown in the histograms of Figure 29.

Clustering coefficient The clustering coefficient in Figure 29a shows that the majority of the graph's cluster has a relatively low clustering coefficient, meaning that they contain few closed triangles. There is a small peak at the extreme right of the histogram, showing those communities with a clustering coefficient (close to) 1. These communities are (quasi-)cliques, where all vertices are connected to each other. The overall low clustering coefficient means that the (OBI4wan) Twitter network does not contain many clusters in which the vertices (persons) are highly connected. Another explanation for the small clustering coefficient value of communities should become clear when looking at the following example. Consider a user A, which is contained in the OBI4wan data set, called D_{OBI}. Then consider another user B, not in D_{OBI}. User A and user B could both have relationships with other users (the set U_{other}) not in D_{OBI}, and users in U_{other} could have relationships among each other, forming a cluster, called C. However, because user B and users in U_{other} are not contained in D_{OBI}, their followers- and friends relationships are not known, preventing the recognition of cluster C or resulting in a much lower clustering coefficient value compared to when the data set D_{OBI} would have been fully complete.


(a) Clustering coefficient


(b) TPR


(c) Bridge


(d) Diameter

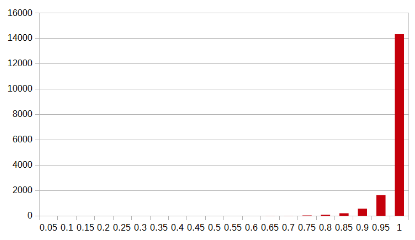

(e) Conductance

Figure 29: Distribution of the statistical indicators for the OBI4wan friends/followers graph (1 million users).

TPR The triangle participation ratio (TPR) in Figure 29b shows two peaks at the extremes, and the rest of the communities is divided over the intermediate ranges. This shows that many communities either have zero (or very few) vertices that close triangles, or (almost) all vertices close triangles.

Bridge ratio Figure 29c shows that most of the clusters have a relatively small bridge ratio. The communities with a bridge ratio of zero (on the left extreme of the histogram) are the communities that do not have a bridge edge. The communities with a bridge ratio of one (on the right extreme of the histogram) are the communities that consist only of bridge edges. A low bridge ratio value means that there are not many communities which contain an edge which disconnects the community from the rest of the graph. This might be an indication that communities have many connections (edges) to vertices outside the community, meaning that communities are unlikely to be completely cut off from the rest of the graph.

Diameter The diameter value in Figure 29d shows that most of the communities have a diameter of either zero or two. Generally, a (sub)graph can have a diameter of zero if (1) the graph does not contain any vertices or has just one vertex, or if (2) the graph contains vertices but no edges. The SCD results of the OBI4wan followers/friends graph show that no communities consist of zero vertices, and also all communities contain more than zero edges. However, in the SCD algorithm - given that S is the set of vertices in a community - the diameter is calculated by taking the community's actual diameter, and divide it by the *log* of the size of S, plus one (e.g. $\log(|S|) + 1$). When the result of this calculation is a number between 0 and 1 (which is the case if the actual diameter value is low), and the algorithm levels down the result, then diameter values of zero are possible. This might be an indication that communities have a small actual diameter value, and are thus well-connected.

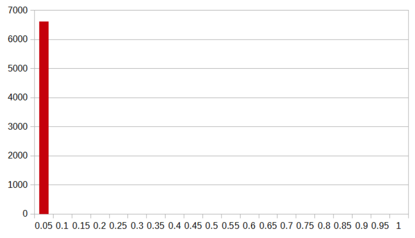
Conductance Figure 29e shows the conductance of communities. For the majority of the communities, the conductance is high (close to 1), meaning that most of the communities are not very well isolated from each other.

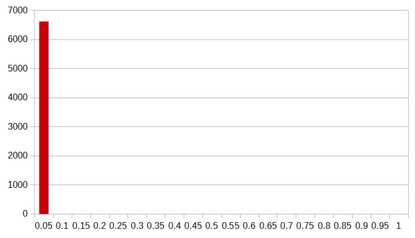
7.4.2 Mentioned graph statistics

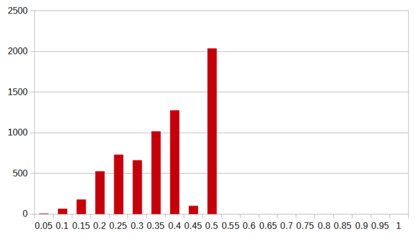
Next, we will analyze the results from executing SCD's algorithm on OBI4wan's "mentioned" graph, as shown in the histograms of Figure 30.

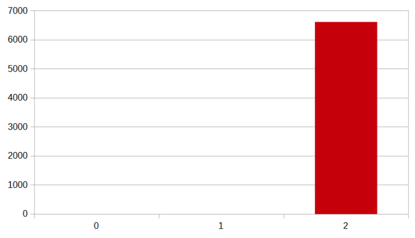
Clustering coefficient The clustering coefficient in Figure 30a shows that the value for every community is equal to zero. This is due to the fact that the number of triangles in each community is also equal to zero. This means that given a person A, a person B and a person C, there is no community in which person A mentions person B, person B mentions person C and person C mentions person A. This could be normal behavior in the Twitter network, as usually people just mention other people in their tweets when they have a (public) conversation with each other.

TPR Again, because there are no triangles in each of the communities (see the details about the clustering coefficient above), the TPR for all communities is equal to zero. This is shown in Figure 30b.


(a) Clustering coefficient


(b) TPR


(c) Bridge


(d) Diameter

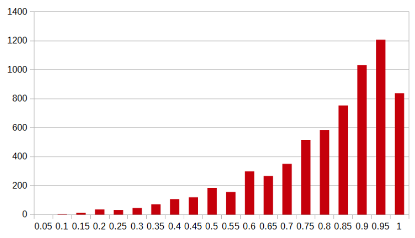

(e) Conductance

Figure 30: Distribution of the statistical indicators for the OBI4wan mentioned graph (all tweets).

Bridge ratio Figure 30c shows that all communities have a bridge ratio equal to or lower than 0.5, with a majority of the communities having a bridge ratio of exactly 0.5. A bridge ratio of 0.5 means that the number of bridges (counting a bridge twice, once for each vertex connected to the bridge) is equal to half of the number of edges that connect members of the community with each other. The relatively low bridge count value indicates that many of the communities do not have many bridges going outside the community. This can be explained by looking at how people use mentions on Twitter. Most of the mentions occur in a conversation between people, for example a conversation between person A, B and C. In this conversation, these people will mention each other a lot, but are not likely to mention any other person D (where D is not equal to A, B or C).

Diameter Figure 30d shows the diameter of all communities is equal to 2. In other words, it never takes more than two steps to travel from one person in a community to another. Why is this?

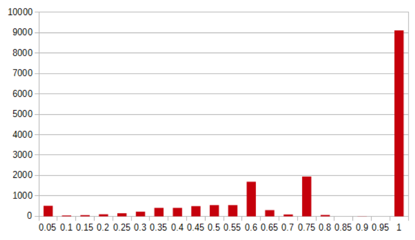
Conductance Figure 30e shows the conductance of communities. Many of the communities have a conductance that is relatively high, with the majority having a conductance between 0.9 and 0.95. This means that most of the communities have a relatively bad isolation - changes are high that you will often travel from one community to another when randomly walking through the graph. This can be explained by looking at the following example. Person A could be in a community together with person B and person C, where the three persons mention each other a lot during a conversation. However, changes are high that person A is involved in other conversations with other people as well, meaning that there are also a lot of edges originating from person A and arriving at persons from other communities.

7.4.3 Retweeted graph statistics

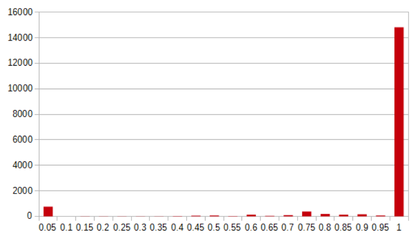
Finally, we will analyze the results from executing SCD's algorithm on OBI4wan's "retweeted" graph, as shown in the histograms of Figure 31.

Clustering coefficient The clustering coefficient in Figure 31a shows that the majority of the clusters have clustering coefficient equal to 1, with the other communities having a clustering coefficient somewhere between 0 and 1, rather evenly distributed over those other communities (except for two relatively larger peaks at ranges 0.55-0.60 and 0.70-0.75). Almost all communities with a clustering coefficient of one consist of three vertices, with a diameter of two, a TPR of one and a bridge ratio of zero. These communities could consist of three persons A, B and C, where person A has retweeted a tweet from person B, and person C has retweeted person A's retweet, indirectly also retweeting person B's tweet. This closes the triangle between A, B and C (TPR of 1), introduces a "longest shortest path" (thus the diameter) from person C to B via A of length 2, and contains no connections to other nodes (bridge ratio of 1). See Figure xx for a graphical representation of this community. The communities with a smaller clustering coefficient have the same kind of structure, but contain more vertices.

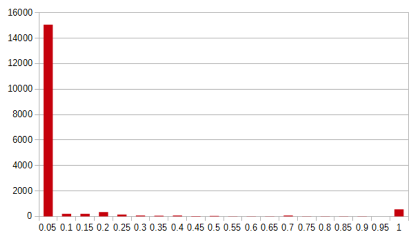
TPR Figure 31b shows that the majority of the communities have a TPR of 1. These communities are already explained above in the clustering coefficient paragraph. There also exist some communities with a TPR of 0, meaning that there exist no triangles in these communities. These communities could consists of a person A, who has retweeted tweets from many other persons. Such a community contains only outgoing edges from person A to other persons, but no edges between these other persons.



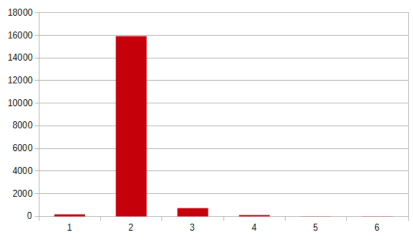
(a) Clustering coefficient



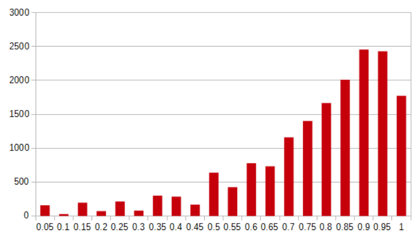
(b) TPR



(c) Bridge



(d) Diameter



(e) Conductance

Figure 31: Distribution of the statistical indicators for the OBI4wan retweeted graph (all tweets).

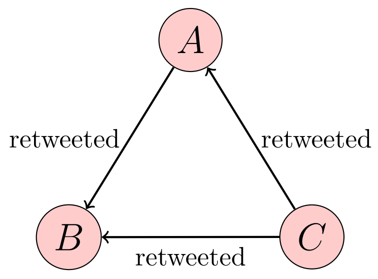


Figure 32: An example community of three persons, where person A has retweeted a tweet from person B, and person C has retweeted person B's retweet, thereby also indirectly retweeting person A's tweet.

Bridge ratio Figure 31c shows that most of the communities have bridge ratio equal to zero, meaning that these communities do not have any bridges. This type of community has been discussed in the paragraph about the clustering coefficient.

Diameter Figure 31d shows that most of the communities have a diameter equal to 2. Again, this type of community has been discussed in the paragraph about the clustering coefficient.

Conductance Figure 31e shows that the majority of the communities has a conductance higher than 0.5, with peaks at the range from 0.85 to 0.95. This shows that communities are not very well isolated. This can be explained by considering a person A, who could be part of many communities in which one of person A's tweet has been retweeted by some people (where each community is based on another of person A's tweets). There could be a lot of other persons similar to person A, meaning that communities are not isolated but connected to each other through such persons.

7.5 Loading OBI4wan data into databases

The previous sections have discussed the translations of OBI4wan Twitter archive data into the same format that is output by the LDBC DATAGEN. After this process is complete, the scripts that were used to load the LDBC DATAGEN data into Titan and Monet can be reused to load the translated OBI4wan data into those databases.

8 Benchmarks setup and execution

In order to say something about the performance of both Titan and MonetDB, benchmarks can be used. A benchmark for graph databases is the LDBC benchmark, which can be used for both the LDBC data (using the original benchmark) and the OBI4wan data (using a custom version of the benchmark). This section describes how to set up this benchmark for both database systems, how to execute it and analysis of the results.

Research questions:

- **Benchmarks**: How can one benchmark be executed on both a graph database (Titan) and a column database (MonetDB) without introducing any kind of advantage/disadvantage for one of the database types (e.g. because of how benchmarks can be expressed in queries, you do not want to lose performance or precision because the query language lacks expression)?
- **Data**: which part of the data that is output by the LDBC data generator is useful to use in benchmarks to represent the Twitter social network? Which queries of the LDBC driver are useful to use in benchmarks to represent the Twitter social network?
- **Business questions**: Which business questions are interesting to answer for OBI4wan?
  - How can one best design such business questions? Which scenarios have to be covered by the business questions?
  - How can the business questions be transformed into executable benchmarks?
- **Architecture**: is the graph query language Gremlin expressive enough to execute the benchmarks (which were created based on designed business questions)?
- **Driver**: how can the drivers for both database solutions be validated?
- **Benchmarks**: how to interpret the results from the LDBC driver benchmarks? How should differences in execution times per query for both database solutions be interpreted?

In this section, the setup and execution of the LDBC benchmarks is described. Section 8.1 shows how the LDBC driver for both Titan and MonetDB has been implemented, giving detailed information for each query handler that has been written. Section 8.2 describes how the database-specific implementations have been validated for correctness, using validation sets from the LDBC validation GitHub repository[23]. Section 8.3 also shows database-specific implementations, but now for the real Twitter data set and accompanying queries from OBI4wan. Section 8.4 shows further benchmark setups, for example the values of various properties that can be specified when running an LDBC benchmark. Section 8.5 shows which benchmarks have been run for both Titan and MonetDB, and details the hardware specifications of the machines on which the two database backends were implemented and the benchmarks were run. The results from these benchmarks are analyzed in section 8.6.

8.1 Creating the LDBC Driver

The full LDBC benchmark consists of 14 complex queries, 7 short queries and 8 update queries. These queries are based on the social network data that is output by the LDBC DATAGEN, which structure resembles that of Facebook, meaning it also contains entities like forums on which users post messages, organizations that people are connected to, interests the person has,

[23]See https://github.com/ldbc/ldbc_snb_interactive_validation

etc. Not all of these entities are used on Twitter, which is the social platform from which the OBI4wan data is collected. In order to let the LDBC data resemble the data structure of Twitter, some entities and consequently some queries are taken out from the LDBC benchmark set. After stripping down the LDBC data set and benchmark queries, the following data and queries remain:

Vertices	person, post, comment, tag
Relations	comment_hasCreator_person, comment_hasTag_tag, comment_replyOf_comment, comment_replyOf_post, person_knows_person, person_likes_comment, person_likes_post, post_hasCreator_person, post_hasTag_tag
Complex queries	1, 2, 4, 6, 7, 8, 9, 13, 14
Short queries	1, 2, 3, 4, 5, 7
Update queries	1, 2, 3, 6, 7, 8

For the LDBC benchmark to work, each of the remaining queries needs a query handler. This query handler takes care of receiving data input for the queries (delivered by LDBC substitution parameters during a benchmark run), building up queries for the respective database system (with the Gremlin language for Titan, and SQL for Monet), executing these queries on the database system and processing the results that the database system returns. During the construction of the query handlers, a lot of example has been drawn from existing implementations for Titan and Virtuoso[24], which can be found in LDBC's SNB implementation GitHub repository[25]. The next two subsections will discuss the query handlers for both Titan and Monet.

8.1.1 LDBC driver for Titan

...

Often used constructs Some of the constructs used when constructing query handlers for Titan come back regularly. These constructs are discussed below. In the details about the various query handlers, often a reference to a "global construct" that is discussed here is given.

PipeFunction In Gremlin, a *PipeFunction* is often used in order to process the results (or intermediate results) of a query. Gremlin's $filter()$ and $order()$ functions can use such a *PipeFunction* in order to further process (intermediate) results. Consider an imaginary Gremlin query that contains the following vertices in its result:

```
Vertex A: {id: 123, firstname: "John", lastname: "Carter"}
Vertex B: {id: 456, firstname: "Sara", lastname: "DeVries"}
Vertex C: {id: 789, firstname: "John", lastname: "Waldorf"}
```

On this set of results, we could use a $filter()$ function, with as argument the name of a *PipeFunction*, for example $filter(HAS\_FIRST\_NAME)$. The implementation of the HAS_FIRST_NAME function could look as follows:

[24]https://github.com/openlink/virtuoso-opensource
[25]https://github.com/ldbc/ldbc_snb_implementations/tree/master/interactive

```
final PipeFunction<Vertex, Boolean> HAS_NAME = new PipeFunction<Vertex, Boolean>() {
  public Boolean compute(Vertex v) {
   return v.getProperty("firstname").equals(friendFirstName);
  }
};
```

Each *PipeFunction* always contains the *compute*() function, which iterates over all the vertices that are in the (intermediate) result set. In this specific case, it check for each vertex in the set if it's $firstname$ property equals the content of the variable $friendFirstName$. If it does, $True$ is returned and the vertex remains in the set, otherwise $False$ is returned and the vertex is removed from the set.

Getting vertex

All vertices in the Titan graph contain a custom vertex ID property, called *iid*. The syntax of the *iid* consists of the vertex type, followed by a dash and then the actual vertex ID: $< type > - < id >$ (for example $person - 12345$). The following function retrieves a $Vertex$ object by its ID and type (t is a variable that points to an instance of the Titan graph).

```
public Vertex getVertex(long id, String type) {
  String vertexId = String.format("%s-%d", type, id);
  Iterable<Vertex> vertices = t.getVertices("iid", vertexId);
  if (vertices.iterator().hasNext()) {
   return vertices.iterator().next();
  }
  return null;
}
```

If the $getVertices$() function contains vertices, the next object in the set is returned. Because all vertex IDs are unique, this set can only contain one instance - or zero if the vertex has not been found, in which case *null* is returned.

Friend-of-Friend

Some of the LDBC queries require to retrieve all people who are friends or friends-of-friends of some person. "Friends-of-friends" are those persons who are two "knows" relation steps away from some person - they are the friends of the friends of some person. The basic logic behind a function that needs to retrieve all friends and friends-of-friends, is to retrieve all people who are one or two "knows" relation steps away from some person. The following function achieves this:

```
public Set<Vertex> getFoF(long rootId, TitanDb.TitanClient client) {
  Set<Vertex> res = new HashSet<Vertex>();
  Vertex rootVertex = client.getVertex(rootId, "person");
  GremlinPipeline<Vertex, Vertex> gp = (new GremlinPipeline<Vertex, Vertex>(rootVertex));
  gp.both("knows").aggregate(res).both("knows").aggregate(res).iterate();
  res.remove(rootVertex);
  return res;
}
```

Starting at some person vertex retrieved by the function $getVertex$(), the *GremlinPipeline* traverses all friends and friends-of-friends of this person through the double "knows" relationship construct. Gremlin's *aggregate*() function collects and stores all of the (intermediate) resulting vertices in the variable *res*, which is a *Set* with *Vertex* objects. In a

GremlinPipeline, iterating over all (intermediate) results - in this case iterating over all friends of the start person - must be done manually with the *iterate*() function.

Adding vertices Adding a vertex to a Titan graph can be done by using the *addVertex*() function and storing the result in a *Vertex* object. Vertex attributes can then be added to the new vertex by calling the *setProperty*() function on the new *Vertex* object, which accepts two parameters: the key and the value of the new property. The following method can add a new vertex to a Titan graph.

```
public Vertex addVertex(long id, String type, Map<String, Object> properties) {
  Vertex v = t.addVertex(null);
  if (v == null) {
   return null;
  }
  v.setProperty("iid", getNewVertexId(id, type));
  v.setProperty("type", type);
  for (Map.Entry<String, Object> p : properties.entrySet()) {
   v.setProperty(p.getKey(), p.getValue());
  }
  return v;
}
```

This function receives the ID, type and properties (attributes) of the new vertex, and creates a new vertex with the given ID and properties.

Adding edges Adding a new edge between two vertices in a Titan graph can be done by using the *addEdge*() function. This function accepts four parameters: optionally an ID used "under-the-hood" by Titan, the outgoing vertex, the incoming vertex and an edge label. When creating a new edge, the reference to this new object can be stored in an *Edge* object. On this new *Edge* object the function *setProperty*() can be called, which adds a new property (a key and a value are accepted parameters) to the edge. The following method can add a new edge to a Titan graph.

```
public void addEdge(Vertex vOut, Vertex vIn, String label, Map<String, Object> properties) {
Edge e = t.addEdge(null, vOut, vIn, label);
for (Map.Entry<String, Object> entry : properties.entrySet()) {
e.setProperty(entry.getKey(), entry.getValue());
}
}
```

This function receives the outgoing vertex, the ingoing vertex, an edge label and edge properties, and creates a new edge with this data.

Complex queries

Query 1 This query requires to store the number of steps from one person to another via the 'knows' relationship, with a maximum of three steps. Storing these number of steps can be achieved by creating a *Set* of *Vertex* objects for each step number, e.g. all vertices that are reached after 1 step, 2 steps and 3 steps. To store the intermediate steps, we can use Gremlin's *store*() function. Finally, a *filter*() is used to only store those vertices of which the *firstname* property is equal to the value given by the substitution parameters, and the final result set is ordered ascending by the person's last name and then the person's identifier.

Query 2 Two *PipeFunction*s are used in this query. The first is a *filter*() function which filters out all vertices from an intermediate result set that have a date that is larger than the maximum date given as a substitution parameter. The second is an *order*() function which order the final result descending by creation date and ascending by message identifier.

Query 4 This query finds all tags attached to some post A that were not used in other posts before post A - with post A's creator being a friend of some start person that is given by the substitution parameters. To produce correct results, we need to find to sets of tags: set A which contains all tags that were used in some person's posts before date X, and set B which contains all tags that were used in the post that was created at date X. The final query then finds all tags from posts that were created by the start person's friends (set B), except those tags that were used in earlier created posts (set A). Excluding a set of tags is possible using Gremlin's *except*() function, which accepts a set (of tags) as parameter. The number of times a tag occurs in the result set can be calculated by using Gremlin's *groupCount*() function.

Query 6 All friends and friends of friends of some start person can be retrieved by using the special *getFoF*() function as discussed in the "often used constructs" paragraph. Starting from this *Set* of *Vertex* objects (persons), the query finds all posts of these persons that contain the tag given by the substitution parameters. Next, all tags that co-occur with this tag on the found posts need to be retrieved. We can use Gremlin's *back*() function to return to an earlier position in the graph traversal, while keeping the already found posts and their tags in memory. This makes it possible to first find all posts with the given tag from the substitution parameters, and the find all other tags on these posts using the *back*() function and the *hasNot*() function to exclude the tag from the substitution parameters. In code, this construct looks as follows:

```
.as("post").out("hastag").has("name", tagName)
.back("post").out("hastag").hasNot("name", tagName)
```

Finally, Gremlin's *groupCount*() function can be used to count the number of posts that contain both the tag from the substitution parameters and some other tag.

Query 7 This query retrieves all persons who have liked a post created by some start person that is given by the substitution parameters. In addition, the query checks whether a person who has liked a post is friends with (knows) the start person. For this check, a *Set* with *Vertex* objects that contains all friends of the start person is stored. If a result person is contained in this set of friends, then we know that this person is friends with the start person.

For each person, only the most recent post that this person has liked must be returned. This can be done by first collecting all posts that a person has liked, and then compare the creation date of these posts to return the most recent one.

Query 8 This query is relatively straightforward. Starting at some start person given by the substitution parameters, the query retrieves all comments on messages that were created by the start person. These comments are stored in a *comment* variable, and the person who posted the comment is stored in a *commenter* variable. Finally, the results are sorted descending by the comment creation date.

Query 9 All friends and friends of friends can again be retrieved using the special $getFoF()$ function that has been discussed earlier. The resulting *Vertex* objects can be stored in a *Set* called $friends$. Starting at those friends, a graph traversal retrieves all messages that were created by these friends which were posted before a given date. All posts that were posted after this date can be filtered out by using Gremlin's $filter()$ function. Finally, the results are sorted descending by the message creation date.

Query 13 When finding the shortest path between two vertices, one of the challenges is to not visit the same vertex twice in one graph traversal. This would introduce loops in the traversal, which never leads to the shortest path in a graph. In other words, during each traversal iteration, we need to skip those vertices that have been visited before. This can be done in Gremlin by storing all vertices that have been visited before in a *Set* of *Vertex* objects, which could be called x. The code for such a traversal is shown below:

```
start.as("x").both("knows").except(x).store(x).loop("x", ...)
```

Gremlin's *loop*() function can contain a total of three parameters. The first parameter is a reference to the point where the next iteration of the loop needs to start, in this case at the point that is called x. The second parameter is the stop condition; when this condition has been reached, Gremlin breaks out of the loop. The third and final parameter can emit intermediate results to the console. For this query, the second parameter with the stop condition is important. We need to continue searching for the shortest path until the end vertex for which is searched has been found. This can occur when either the end vertex is contained in *Set x*, or when the current vertex is equal to the end vertex. In code, this looks as follows (where *bundle.getObject*() retrieves the current *Vertex* object):

```
!x.contains(end) && !bundle.getObject().equals(end);
```

With this stop condition, the traversal stops as soon as the shortest path has been found. The length of this path can be returned as the result of this query.

Query 14 This query is similar to query 13. Again, the shortest path between two vertices must be found, but there are two differences compared to query 13. We now need to find *all* shortest paths and the vertices that are on that path, and we need to calculate a weight for each path. The strategy for this query is to split the finding of the shortest paths from calculating the weight of these paths. In other words, we will first find all shortest paths, and afterwards calculate a weight for each of these shortest paths.

Finding the shortest path works similar as in query 13, with a few differences. A *Stack* is kept in memory, which contains the shortest path found until now at the top. If the graph traversal finds another path that is longer than the current shortest path, this path is not taken into account for the final result. Another difference is that not only the length of the shortest path is needed for the final result, but also all vertices which are on this shortest path. Outputting all vertices on a path can be done by using Gremlin's *path*() function, which outputs the path list.

Finding the weight of all shortest paths can be done by looping through all found shortest path. For each path, all consecutive vertices on the path are considered. So, in each iteration (starting at an index of $i = 1$), vertices i and $i - 1$ are considered. Two consecutive persons add to the weight of the path when one of the persons has replied to a post or comment of the other person. In the first case (reply to a post), 1 is added to the total path weight, in the second case (reply to a comment), 0.5 is added to the total path weight. The final path weight is the sum of all replies on posts or comments by each pair of consecutive persons in a shortest path.

Short queries

Query 1 The first short query just retrieves one *Vertex* object (a person) from the graph. This can be done by using the *getVertex*() function which has been discussed earlier.

Query 2 This query can be split into two parts. In the first part, the 10 last messages created by a start person that is given by the substitution parameters are retrieved and stored in a *message* variable. The returned messages are sorted descending by their creation date using Gremlin's *order*() function.

In the second part, for each found message the root post in the conversation is retrieved. Retrieving the root post in a conversation can be found by starting at some message in the conversation, and traverse back up to the conversation root by following the "replyOf" relationship. When a message does not have this relationship, it means it is the root message in the conversation and that the traverse has completed. In Gremlin, this can be achieved by using the following piece of code:

```
message.as("start").out("replyof").loop("start", true, true).as("rootMessage");
```

This loop keeps traversing up through the "replyOf" relationship until it does not exist anymore, and at that moment the current message in the conversation - which is the root message - is stored in the variable *rootMessage*.

Query 3 This query finds all friends of a start person as given by the substitution parameters, and the date at which the friendship started. This required information of both the friend vertex and the friendship edge. In the Gremlin graph traversal, both entities need to be stored. Starting from the start person, first the friendship ("knows") edge is stored by using Gremlin's *bothE*() function, which retrieves all edges originating from or arriving at the start person. Next, starting at this set of edges, we can use Gremlin's *bothV*() function to retrieve all vertices which are at the end of the set of edges, in this case being all friends of the start person. Finally, the results are ordered descending by the friendship creation date using Gremlin's *order*() function.

Query 4 Like short query 1, this query just retrieves one *Vertex* object (a comment or a post), which means that we can use the earlier mentioned *getVertex*() function again. Because the substitution parameter only gives an ID to search for, we do not know whether the message is a comment or a post. This problem can be solved by first trying to find a comment which has the given ID. When this does not return a *Vertex* object (e.g. when the *getVertex*() function returns *null*), the vertex must be a post an we search for the given ID in combination with "post" as the vertex type.

Query 5 This query is the same as short query 4, but returns other attributes from a message (post or comment) than query 4.

Query 7 The result for this query can be found in two steps. First, all comments that are replies to the start message given by the substitution parameters must be found, ordered descending by their creation date using Gremlin's *order*() function. Then we need to know if the author of the reply knows the author of the start message. We can figure this out by first retrieving all friends of the author of the start message, store these friends in a *Set*, and then check if an author of a reply is contained in this set of friends. The final challenge of this query is that the content of a message should be returned. For comments, there exists only one type of message content (only text), but the content of a post could be textual (*content* attribute) or an image (*imagefile* attribute). We can solve this challenge by first trying to retrieve the *content* attribute of a message. If this returns *null*, then we know that the message is a post and contains an *imagefile* attribute, which can then be retrieved.

Update queries The update queries either add new vertex to the graph, a new edge, or both. For example, the addition of a new comment requires the addition of a new vertex (the comment itself) and the addition of new edges (to the person who created the comment, and to any tags that are contained in the comment). Adding vertices and edges in Titan can be done by using the special $addVertex()$ and $addEdge()$ function, which have been discussed in the previous "often used constructs paragraph".

8.1.2 LDBC driver for Monet

Often used constructs

Substitution parameters The LDBC benchmark provides substitution parameters for each query. For Monet all complex- and short queries are specified in separate SQL-files, which are loaded into each query handler as a String. In order to insert the substitution parameters into those queries, the queries contain placeholder at the positions where substitution parameter values should be inserted at runtime. These placeholder always have the same syntax: $@ < placeholder > @$. For example, if the substitution parameter for a query contains a person's ID, the placeholder in this query is $@Person@$. The function that transforms a SQL-file to a String is shown below.

```
public static String file2string(File file) throws Exception {
  BufferedReader reader = null;
  try {
    reader = new BufferedReader(new FileReader(file));
    StringBuffer sb = new StringBuffer();

    while (true) {
      String line = reader.readLine();
      if (line == null) {
       break;
      }
      else {
        sb.append(line);
        sb.append("\n");
      }
    }
    return sb.toString();
  } catch (IOException e) {
   throw new Exception("Error openening or reading file: " + file.getAbsolutePath(), e);
  } finally {
    try {
      if (reader != null) {
         reader.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
    }
  }
}
```

This function opens the file containing the SQL-code, reads through it line by line, and appends each line to a *StringBuffer* object. This returns a String, potentially containing placeholders. Finally, these placeholders can be replaced by the substitution parameters by calling Java's *replaceAll*() function on the String.

Date formatting All date/time attributes are stored in the Monet database as a TIMESTAMP. The date values that are given by the substitution parameters are a Java *Date* object, and need therefore be formatted before they can be used in SQL statements. There are two places where date formatting is needed: from the substition parameters to SQL strings, and from a SQL result to a long variable.

- **From substitution parameters to SQL strings**: Date/time objects from the substitution parameters are a Java *Date* object, and should be translated into a SQL *Timestamp* object. Furthermore, all *Timestamp*s must be of the GMT time zone. The following code will transform the substitution parameters into the right format:

  ```
  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
  TimeZone gmtTime = TimeZone.getTimeZone("GMT");
  sdf.setTimeZone(gmtTime);
  ```

  This creates a *SimpleDateFormat* object. Finally, the *format()* function can be called on this object in order to format a substitution parameter into the correct format.

- **From SQL result to long**: Date/time objects are given as a SQL TIMESTAMP object when they are part of a SQL result. In order to parse such a TIMESTAMP into a long variable, we can use the following function:

  ```
  public static Long getTimeStampFromSql(ResultSet queryResult, int columnIndex) {
    Calendar calendar = Calendar.getInstance();
    calendar.setTimeZone(TimeZone.getTimeZone("GMT"));
    Timestamp ts;
    try {
      ts = queryResult.getTimestamp(columnIndex, calendar);
      return ts.getTime();
    } catch (SQLException e) {
      e.printStackTrace();
      return new Long(-1);
    }
  }
  ```

  This function creates a *Calendar* object which is set to the GMT time zone. Then, the SQL TIMESTAMP is retrieved from the SQL result with the *getTimestamp*() function, and the resulting long timestamp value will be formatted into the GMT time zone.

Timestamp difference Complex query 7 needs to calculate the difference between two timestamps. The function shown below accepts two timestamps, and returns the number of minutes between these two timestamps.

```
CREATE FUNCTION GetTimestampDifference(Timestamp1 TIMESTAMP, Timestamp2 TIMESTAMP)
RETURNS INT
BEGIN
  RETURN
   (Timestamp1 - Timestamp2) / 60000;
END;
```

Friendship test A couple of queries need to know whether two persons are friends or not. In SQL terms, this means that we need to know if the two persons that are given as an input to this function exists in the *person_person* relationship table. The code for this function is shown below:

```
CREATE FUNCTION AreTheyFriends(ThisPersonId1 BIGINT, ThisPersonId2 BIGINT)
RETURNS INT
BEGIN
  RETURN
   SELECT 1 FROM person_person pepe WHERE pepe.personid1 = ThisPersonId1 AND
    pepe.personid2 = ThisPersonId2;
END;
```

Latest likers Complex query 7 needs to retrieve the latest persons that liked a message from some start person. This data can be retrieved by matching the person from the *message_person* relation (*messagehasCreatorperson*) with the *person_message* relation (*personlikesmessage*). The code for this function is shown below:

```
CREATE FUNCTION GetLatestLikers (ThisPersonId BIGINT)
RETURNS TABLE (personid BIGINT, creationdate TIMESTAMP)
BEGIN
  RETURN
   SELECT person_message.personid, MAX(person_message.creationdate) as creationdate
    FROM person_message, message_person
    WHERE
      message_person.personid = ThisPersonId AND
      message_person.messageid = person_message.messageid
    GROUP BY person_message.personid
    ORDER BY MAX(person_message.creationdate) DESC
    LIMIT 20;
END;
```

Latest messages Short query 2 needs to retrieve the 10 latest messages for a certain start person, and also the root message in the conversation of those messages. The best strategy for this query is to first retrieve the 10 latest messages for the start person, and only search for the conversation root message for those 10 messages instead of for *all* found messages created by the start person. Retrieving these latest messages can be done by using the function displayed below:

```
CREATE FUNCTION GetPersonsLastMessages(ThisPersonId BIGINT)
RETURNS TABLE (id BIGINT, imagefile TEXT, content TEXT, creationdate TIMESTAMP)
BEGIN
  RETURN SELECT messages.id, messages.imagefile, messages.content, messages.creationdate
  FROM messages, message_person
  WHERE
    message_person.personid = ThisPersonId and
    message_person.messageid = messages.id
    ORDER BY messages.creationdate DESC
LIMIT 10;
END;
```

This function accepts a person ID as input parameter, and returns a table with the 10 latest messages (order descending by creation date) of that person. For each message the id, image file, content and creation date are returned.

Conversation root Short query 2 needs to find the root message in a conversation. The following function can return this root message, given the ID of a message in a conversation:

```
CREATE FUNCTION GetMessageConversationRootId (ThisMessageId BIGINT)
RETURNS BIGINT
BEGIN
  -- get the type of the current message's parent
  DECLARE ThisMessageType STRING;
  SET ThisMessageType = GetMessageType (ThisMessageId);

  -- if the type of this message is 'post',
  -- then we have reached the root of the conversation
  IF (ThisMessageType = 'post') THEN
    RETURN ThisMessageId;
  END IF;
  -- get the id of the current message's paren
  SET ThisMessageId = GetMessageParentId (ThisMessageId)

  -- recursive call to walk to the root of the conversation
  RETURN GetMessageConversationRootId (ThisMessageId);
END;
```

The function is setup as a recursive function, that keeps returning itself until the current message in the conversation is of type "post" and therefore the root in the conversation. This "post" message is returned by the function.

Complex queries

Query 1 Retrieving all persons who are at most three "knows" relation steps away from some start person given by the substitution parameters can be expressed in SQL by using UNION ALL. Separate SELECT queries can retrieve all persons at 1 step, 2 steps and 3 steps away, which can then be combined using the UNION ALL statement. This results in one dynamic table, from which the persons with a first name given by the substitution parameters can be retrieved.

Query 2 This query retrieves the content of a message, not knowing if this message is a comment or a post. In the latter case, the content could be both textual (*content* attribute) or an image (*imagefile* attribute). For this query, both attributes are being retrieved, and the one that is not *null* is taken as the message's content.

Query 4 This query retrieves those tags from messages that have not been used in messages that were posted earlier. In SQL, this can be expressed using the NOT EXISTS statement. The first SELECT statement selects all tags that occur in posts between two dates from some start person as given by the substitution parameters. Then, the NOT EXISTS part of the query retrieves all tags that occur in posts *before* the timeframe of the first SELECT query, excluding those from the final result.

Query 6 Retrieving the friends and friends of friends of some start person as given by the substitution parameters is expressed in SQL by two separate SELECT queries which are combined using the UNION statement. The first SELECT query selects all directs friends of the start person, and the second SELECT query selects all friends of friends of the start person. Using UNION instead of UNION ALL makes sure the final set does not contain duplicate persons. This "friends-of-friends" set is then used for the rest of the query, which retrieves posts and tags based on some tag given by the substitution parameters.

Query 7 In the SELECT part of the query, two functions are used. The first is $GetTimestampDifference()$, which accepts two TIMESTAMP objects and calculates the difference between them in minutes. The second is $AreTheyFriends()$, which accepts two BIGINTSs as person ID's and checks whether the two persons are friends (e.g. are bound by the "knows" relationship). Another function is used to dynamically create a table: the function $GetLatestLikers()$. This function accepts a BIGINT as a start person's ID, and returns a table with all persons who have liked a message that has been created by the start person. This last function has been introduced, because a dynamic table that is directly written down in a SQL query does not support the use of the ORDER BY statement. This workaround (putting the dynamic table in a separate function) solves this support problem.

Query 8 In order to retrieve the comments on messages of some start person given by the substitution parameters, we need to use two instances of the $message\_person$ table (which is the relation $messagehasCreatorperson$). The first instance is responsible for the binding between the start person and the messages this person has posted, and the second instance is responsible for the binding between a reply message on one of the start person's messages and the creator of this reply. In SQL, multiple instances of the same table can be instantiated by giving both instances a different alias, for example $mepe1$ and $mepe2$:

```
from message_person mepe1, message_person mepe2, ...
```

Query 9 The first part of this query retrieves the friends and friends of friends of some start person given by the substitution parameters, which works the same as in complex query 6. The next step - retrieving all messages (both posts and comments) from this resulting set of persons is straightforward: binding the persons to the messages they have created using the $message\_person$ table (relation $messagehasCreatorperson$).

Query 13 The algorithm for finding the shortest path in a SQL database system (specifically Monet) has been based on the paper "Using the Monet database system as a platform for graph processing" [57]. This paper has proposed a solution for shortest path graph traversal in Monet using a couple of temporary tables which support the shortest path finding process.

These extra tables are called *mpath*, *mpathtotal* and *mtemp*, all containing one column of type BIGINT representing the ID of a vertex. The algorithm (in pseudo-code) is shown below.

```
//initialization phase
insert $source$ into $mpath$
insert $source$ into $mpathtotal$

//the loop
while $sink$ not found and $mpath$ not empty {
    //insert all neighbours of the current front into the $mtemp$ table
    INSERT INTO $mtemp$ $(\pi_{sink}(edges \bowtie_{source=node} mpath))$

    empty $mpath$

    //insert all node values of the $mtemp$ table which are not present in the $mpathtotal$
    //table into the $mpath$ table
    INSERT INTO $mpath$ $(\pi_{mtemp.node}(\sigma_{mpathtotal.node=NULL}(mtemp ⟕ mpathtotal)))$

    copy all values from $mpath$ to $mpathtotal$
    empty $mtemp$

    //The termination criteria
    if $mpath$ is empty: terminate //entire graph connectivity is found
    if $sink$ in $mpath$: terminate //sink is reached
}
```

In the initialization phase, the source vertex (the vertex at which the graph traversal starts) is inserted in both the *mpath* and *mpathtotal* tables. Then, a while loop is started which only returns if the sink has been found (the vertex that needs to be found in the graph traversal) or when the *mpath* table is empty (in which case no new vertices have been found in the current iteration, meaning that all vertices have been visited without finding the sink vertex).

In each iteration of the while loop, the following steps are executed:

1. Find all neighbours of the nodes in *mpath*, continuing the graph traversal to the next series of vertices. Store these new vertices in *mtemp*.
2. Delete all entries from *mpath*.
3. Insert all vertices in *mpath* which are in *mtemp*, but not in *mpathtotal*. This ensures that no vertex is visited twice in one graph traversal, eliminating the possibility of cycles in the traversal (*mpathtotal* contains all vertices which have been visited at some time during the current graph traversal).
4. Insert all vertices from *mpath* into *mpathtotal*. Because the previous step has ensured that all vertices currently in *mpath* are not present in *mpathtotal*, there will be no duplicates in *mpathtotal* after this step.
5. Delete all entries from *mtemp*.
6. The last step is to check if the graph traversal has terminated, because the sink vertex has been found or because all vertices of the graph have been visited (e.g. no new vertices have been found in this iteration, leaving *mpath* empty).

Query 14 This query is similar to query 13 (finding the shortest path between two vertices), but with a few additions. First, in this query not only the length of the shortest path must be calculated, but also all shortest paths and all vertices that are on those paths must be retrieved. Furthermore, a weight has to be assigned to each path. This weight is calculated by taking each consecutive pair of vertices (persons) from a path, and add 1 to the total weight if one of the persons has replied to a post of the other person, and add 0.5 to the total weight if one of the persons has replied to a comment of the other person.

In order to achieve this in Monet, we will again use the extra tables that were introduced in query 13, but with the addition of two columns for *mpath*, *mpathtotal* and *mtemp*. The first of these new columns is called *cost* (data type DOUBLE) and holds the cost of a path. The cost of a path is calculated after all shortest paths have been found, so it will remain at a default value during the shortest path finding phase of the algorithm. The second new column is called *pathString* (data type String) and holds a comma-separated String of all the vertices which are currently on the path. The while-loop of query 13 is also executed in this query, but with a few additions at some steps, as shown below:

1. Each new node that has been found in this step is added to the current *pathString* column value, using SQL's CONCAT structure:

   ```
   CONCAT(CONCAT(mpath.pathString, '-'), edges.personid2) AS pathString
   ```

   Here, the current value of the path String (*mpath.pathString* is concatenated with the newly found vertex ($edges.personid2$). The CONCAT structure is used twice, because it only supports concatenation of two Strings, while we need the concatenation of three Strings.
2. No additions to deleting all entries from *mpath*.
3. Like in query 13, all entries from *mpath* which are not in *mpathtotal* are inserted into *mtemp*. Checking if an entry is present in *mpath* but not in *mpathtotal* is done by comparing the *node* column from both tables (which was the only column in the extra tables from query 13). All other information (*cost* and the current *pathString*) are then automatically transferred to the *mtemp* table.
4. No additions to inserting all entries from *mpath* to *mpathtotal*.
5. No additions to deleting all entries from *mtemp*.
6. No additions to check fro graph traversal termination.

When a shortest path has been found, we can retrieve all shortest path of that length from *mpathtotal* by retrieving all entries from that table who's current value of the *node* column is equal to the sink node. Each of these entries has found the sink node in the same iteration (e.g. with the same path length) as the first shortest path that was found, meaning that all shortest paths will be retrieved from the table.

The next step is to calculate each path's weight. This can be done by looping through the set of paths, and calculating a weight for each pair of consecutive vertices in each path (e.g. starting at an index i of 1, calculating the weight of vertex i and $i-1$ in each iteration of a for loop). The following piece of code shows how to check if one of the person vertices has created a reply on one of the other person's messages, which would add the value of 1 to the current weight of the path:

```
-- reply of p2 to post of p1
SELECT COUNT(*) AS weight
FROM post_person pope, comment_post copo, comment_person cope
```

```
WHERE
  pope.personid = Source AND
  pope.postid = copo.postid AND
  copo.commentid = cope.commentid AND
  cope.personid = Sink
UNION ALL
-- reply of p1 to post of p2
SELECT COUNT(*) AS weight
FROM post_person pope, comment_post copo, comment_person cope
WHERE
  pope.personid = Sink AND
  pope.postid = copo.postid AND
  copo.commentid = cope.commentid AND
  cope.personid = Source
```

The two SELECT statements will find all replies of a person to a post of the other person, and counts the number of replies (e.g. adding the value of 1). The structure for replies on comments works in a similar way, but then the COUNT(*) value is multiplied by 0.5 to ensure a weight of 0.5 for each reply on a comment. UNION ALL is used instead of UNION to allow for duplicate weight values.

Short queries

Query 1 Retrieving a person based on an ID given by the substitution parameters is simply done by using a SELECT on the person's ID.

Query 2 This query uses the *GetPersonsLastMessages*() function which has been discussed in the "often used constructs" paragraph in order to get the 10 latest messages created by a start person as given by the substitution parameters. Of these 10 messages, also the root message in the conversation is retrieved. The benefit of using a function to retrieve the 10 latest messages and returning them in a temporary table is that we need to find the root message in the conversation for only those 10 messages, and not for all messages created by the start person. Also, the *GetMessageConversationRootId*() function is used to retrieve the root message in a conversation, given the ID of a message in a conversation.

Query 3 All information to retrieve a person's friends and the date at which they became friends can be retrieved from the *person_person* table (relation *personknowsperson*). Given a start person by the substitution parameters, this person is person A in the relation table, the friend is person B in the relation table, and the date at which they became friends is in the *creationdate* table.

Query 4 A message could have textual content (*content* property) or an image (*imagefile* property) as its content. The SQL query tries to retrieve both, and the query handler determines which one contains the actual content value.

Query 5 Retrieving a message's author based on the message ID can be done by using the *message_person* table (relation *messagehasCreatorperson*). The message ID is the message-part of the table, the person ID is the person-part of the table. Getting the author's name involves retrieving a person entry from the *persons* table by the person's ID.

Query 7 Determining whether two persons (where one person is the creator of a message, and the other person is the creator of a reply on that message) are friends can be achieved by using the earlier discussed $AreTheyFriends()$ function. Retrieving a message as given by the substitution parameters and getting the comments on that message is straightforward.

Update queries The update queries add new vertices and/or new edges to the SQL tables, which just requires a simple INSERT INTO statement. Because posts and comments are combined in a message table in the Monet database (the type of the message - post or comment - is determined by a column in this *messages* table), an insert of a post or comment needs an INSERT INTO both the *posts/comments* table and the *messages* table. The same is true for edges, for example an INSERT INTO the *comment_post* table also needs an INSERT INTO the *comment_message* table.

8.2 Validating the LDBC driver

Custom database implementations must be validated for correctness before the LDBC benchmarks start. Currently, there exist two validation sets on the LDBC SNB Interactive validation GitHub repository[26]. Both validation sets contain the output from the LDBC DATAGEN (including substitution- and update parameters), and a file (called *validation_params.csv*) which contains the expected results of the benchmark run.

The LDBC benchmark configuration contains a special property called *validate_workload*. When this property is set to *true*, the data set that is loaded into the database under test is checked for validity. In addition, the property *validate_database* must be set to the location of the *validation_params.csv* file. Finally, the *ldbc.snb.interactive.parameters_dir* should point to a directory which contains the substitution- and update parameter files. If preferred, the update parameters could be put into another directory and pointed to by the *ldbc.snb.interactive.updates_dir* property. See the earlier mentioned LDBC validation repository on GitHub for more information and for the exact configuration files.

8.2.1 Stripping down the validation set

In order to make the LDBC data created by the LDBC DATAGEN look more like Twitter data, the output from the generator has been stripped down as discussed in section 8.1. The consequence is that also the expected result set from the *validation_params.csv* file has to be stripped down, so that it only contains those results that can be output by the stripped down data set. Otherwise, the validation set would incorrectly mark results as wrong because the data that should have been returned simply does not exist in the stripped version of the LDBC DATAGEN data set.

A script has been written that strips down the validation set to contain only those results contained in the stripped down LDBC DATAGEN data set. The script accepts the original *validation_params.csv* file, and outputs the stripped down version of this file. The script iterates over all LDBC queries (complex, short and update), and only keeps those query result parameters which are supported by the stripped down LDBC DATAGEN data set. All LDBC queries which are not touched by the scripted are assumed to be non-existent in the stripped down LDBC DATAGEN data set and are removed completely.

For example, complex query 1 only supports the three parameters *ID*, *firstname* and *lastname* in the stripped down version. The following piece of code calls a function which accepts the

[26]https://github.com/ldbc/ldbc_snb_interactive_validation

substitution parameters, the expected result, the result parameters to keep and the query that is currently processed:

```
set_updated_validation(params, result, [0, 1, 2], 'query1')
```

Then, a new list with results is created, called *new_results*. This list will contain only those result properties which are at an index given as an input to the function (the *keep* input parameter). Finally, the results are written to the stripped *validation_params.csv* file. The function with the code containing the aforementioned is shown below:

```
def set_write_query(params, result, keep, query):
  query_name = params.pop(0)
  new_results = []
  for index, item in enumerate(result):
    if index in keep:
      new_results.append(item)
      new_results.insert(0, query_name)
  write_to_file(new_results, result)
```

8.2.2 Validating the Titan and Monet drivers

With all configuration files setup correctly, the validation run of the LDBC benchmark can be executed by using the following command:

```
java -cp target/jeeves-0.2.jar com.ldbc.driver.Client
  -P validation/validate.properties -P <any-other-properties-file>
```

The *jeeves-0.2.jar* file contains the Java executable with the LDBC benchmark code. The class from this executable that is executed is *com.ldbc.driver.Client*. Then, the validation configuration properties file is given as an input parameter. Finally, any other extra properties file can be given as an input, for example a properties file containing information that is needed for connection to the database under test (e.g. the database address and port number, login information, etc.).

When the validation run is successful, a ”success”-message is shown, together with all the executed queries. See the example output below.

```
***
Successfully executed 28 operation types:
    4 / 4         LdbcQuery10
    4 / 4         LdbcQuery11
    ...
***
  Missing handler implementations for 0 operation types:
  ***
  Unable to execute 0 operations:
  ***
  Incorrect results for 0 operations:
  ***

Client  Database Validation Successful
```

If one (or some) of the queries did not return the expected result, then the validation run is marked as "failed". The queries for which the actual result is not equal to the expected result are stored in two files: a file containing the actual result for the query, and a file containing the expected result for the query. These two files can then be used by the developer in order to find out what went wrong, so that any errors can be fixed.

When the validation run has been successful, the actual LDBC benchmark can be run on the database under test.

8.3 Custom OBI4wan benchmarks

to write

8.3.1 LDBC benchmark adaptation

to write

8.3.2 Custom OBI4wan benchmark

to write

8.4 Benchmark set-up

Before the LDBC benchmark can be executed on both Titan and MonetDB, various configuration files and property files have to be setup correctly. The LDBC validation configurations as shown on the LDBC validation GitHub repository [42] have been taken as a starting point for the configuration of Titan and Monet. Some of the configuration parameters have the same values for Titan and Monet, others are slightly different.

The following subsection shows the general configuration for both Titan and Monet, and afterwards two subsections show the specific configuration for Titan and Monet.

8.4.1 Setting up the general benchmark configuration

Both Titan and Monet have a properties file that is called *workload.properties*. This file contains information about the actual workload that is used in the LDBC benchmarking process, the location of the database implementation class, the parameter directories for the substitution parameters and the update parameters, the amount of short queries compared to the amount of complex queries, the frequency of each of the complex queries, and the complex-, short- and update queries that should be enabled for the benchmark. The paragraphs below discuss these configuration parameters in more detail.

workload **value**: com.ldbc.driver.workloads.ldbc.snb.interactive.LdbcSnbInteractiveWorkload

Contains the path of the workload class that will be used in the LDBC benchmark run. Apart from the *SNB Interactive Workload*, other (future) versions of the LDBC Social Network Benchmark (SNB) could contain other types of workload, such as the *Business Intelligence Workload* or the *Graph Analytics Workload*. More information about the various types of SNB workloads can be found on the LDBC website[27].

[27]http://ldbcouncil.org/benchmarks/snb

database **value**: com.ldbc.driver.workloads.titan/monet.db.TitanDb/MonetDb

Contains the path to the class containing the specific database implementation, for Titan or for MonetDB. This database implementation contains code to connect to the Titan- or MonetDB database, code to maintain this connection over multiple query executions and some other code needed for database operations. In other words: this configuration parameter points to the database under test.

ldbc.snb.interactive.parameters_dir **value**: *substitution parameters directory*

Contains the (local) filepath to the directory containing the substitution parameters for the current workload. The LDBC DATAGEN outputs these substitution parameters together with the rest of the benchmark data. The substitution parameters are variables inserted into the benchmark queries at runtime.

ldbc.snb.interactive.updates_dir **value**: *update parameters directory*

Contains the (local) filepath to the directory containing the update parameters for the current workload. Like the substitution parameters, the update parameters are output by LDBC DATAGEN together with the rest of the benchmark data. The update queries use the update parameters to insert new records in the database under test.

ldbc.snb.interactive.short_read_dissipation **value**: 0.2

Contains the *short reads random walk dissipation rate*, a value between 0 and 1. A value closer to 1 means fewer random walks and therefore fewer short reads. The value 0.2 is the default value.

ldbc.snb.interactive.LdbcQueryX_freq **value**: *default*

Contains the frequency of complex queries upon benchmark execution time. A higher number means that more update queries are executed in between the complex queries. This value should be set for all uses complex queries. For the benchmark runs in this research, the default values have been chosen.

ldbc.snb.interactive.LdbcQueryX_enable **value**: *true* or *false*

Defines whether or not a certain query is executed during benchmark runs or not. For this research, all queries that support the stripped down data set are executed. See section 8.1 for an overview of the supported queries. This value should be set for all complex-, short- and update queries.

Another type of configuration file that is set for both Titan and Monet are the files *ldbc_driver_monet.properties* and *ldbc_driver_titan.properties*. These files configure the LDBC driver itself. The paragraphs below show these driver configuration options in more detail, also showing the used setting for the benchmark runs in this research.

status **value**: 5

Time interval (in seconds) at which to show information about the current benchmark run. An example of such an information line is shown below:

```
5668 [WorkloadStatusThread-1443340504544] INFO
com.ldbc.driver.runtime.WorkloadStatusThread - Runtime [00:00.191.000 (m:s.ms.us)],
Operations [4], Last [00:00.041.000 (m:s.ms.us)],
Throughput (Total) [20.94] (Last 0s) [20.94]
```

This line contains information about the time in milliseconds into the benchmark run (here: 5668), the elapsed time from the benchmark data point of view (e.g. looking at the timestamps contained in the benchmark data, here: Runtime [00:00.191.000 (m:s.ms.us)]), the number of operations executed up until this point (here: Operations [4]), the execution time of the last operation (here: Last [00:00.041.000 (m:s.ms.us)]) and the total- and last operation throughput (here: Throughput (Total) [20.94] (Last 0s) [20.94]).

thread_count **value**: 24
The number of threads used by the LDBC benchmark. The number of threads defines how many operations can be thrown at the database under test concurrently. In this case, the LDBC benchmark can run 24 threads, meaning that 24 different operations can be executed concurrently. For testing purposes, this value could be set to 1 in order to execute operations one at a time.

name **value**: LDBC
Name that is given to the benchmark run.

results_dir **value**: *results directory*
Local filepath to the directory that will contain the results of the LDBC benchmark run. This directory will eventually contain three result files: (1) a file containing the actual benchmark results (e.g. for each query the number of times it has been executed, the minimum-, maximum and mean execution times, etc.), (2) a file containing the execution log, only if the configuration parameter shown below (*results_log* is set to *true*) (with - in order - all the queries that have been executed) and (3) a file containing the configuration for the benchmark run.

results_log **value**: *true*
If set to *true*, the benchmark outputs the benchmark log file as shown above (see the paragraph about the *results_dir* configuration parameter).

time_unit **value**: MILLISECONDS
Time unit in which to measure benchmark execution times. This configuration parameter can be set to one of the following values: *NANOSECONDS*, *MICROSECONDS*, *MILLISECONDS*, *SECONDS* or *MINUTES*.

time_compression_ratio **value**: 1.0E-4
When this configuration parameter is set to *1*, then all operations in the benchmark are executed exactly at their timestamp value. A value between 0 and 1 speeds up the benchmark, a value bigger than 1 slows down the benchmark (e.g. a value of 0.5 executes the benchmark operations two times faster; a value of 2 executes the benchmark operations two times slower).

validate_workload **value**: *false*
If set to *true*, the benchmark is run in validation mode, to check whether the current query implementation is correct for the database under test. If set to *false*, the "default" benchmark is executed.

workload_statistics **value**: *false*
If set to *true*, extra statistics about the benchmark workload are calculated, such as the operation mix.

spinner_wait_duration value: 0

Contains a time value (in milliseconds) specifying how long the benchmark should wait between each iteration in a *busy wait loop*. When set to a value larger than zero, this configuration parameter can take some load off the CPU.

ignore_scheduled_start_times value: *false*

If set to *true*, the benchmark does not take into account the actual query start times, but just executes all queries as fast as possible. If set to *false*, the actual start times are considered when running the benchmark.

In addition to the configuration parameters described above, Titan and MonetDB have some database-specific configuration parameters that must be set for a correct benchmark run. These parameters are discussed in the next two paragraphs.

8.4.2 Titan-specific configuration parameters

Titan contains one additional configuration file, that specifies some parameters for the Titan graph database. This file is called *titan.properties*, and its content is shown below:

```
storage.backend=cassandra
storage.hostname=192.168.64.205

index.search.backend=elasticsearch
index.search.hostname=192.168.64.205
index.search.elasticsearch.client-only=true

cache.db-cache=true
cache.db-cache-clean-wait=20
cache.db-cache-time = 180000
cache.db-cache-size = 0.25
```

The first two sets of configuration parameters contain address information about the storage- and indexing backend used, in this research Cassandra and ElasticSearch. The final set of configuration parameters contain cache settings, namely (1) enabling the cache, (2) setting the time to wait before cleaning the cache after the maximum caching time has expired, (3) setting the maximum cache time and (4) setting the maximum cache size as a percentage of the heap space.

8.4.3 MonetDB-specific configuration parameters

MonetDB also contains one additional configuration file, that contains authentication credentials for the MonetDB database instance. This file is called *monet.properties*, and its contents are the username and password for the MonetDB database instance.

8.5 Running the benchmarks

The LDBC benchmarks are executed on machines of CWI's SciLens cluster. The CWI is the Dutch centre for mathematics and computer science (Centrum voor Wiskunde en Informatica). The SciLens cluster is built to handle massive amounts of data for research purposes. The cluster consists of various machine types with different configurations, named *diamonds*, *stones*, *bricks*, *rocks* and *pebbles*[28]. The LDBC benchmarks in this research are using two of these machines types: the *bricks* machines for running the LDBC benchmark on a Titan graph database, and the *diamonds* machines for running the benchmark on a MonetDB database. In Titan's case, some of the benchmarks use multiple *brick* instances in a distributed setting, while MonetDB only uses a single-machine architecture with just one *diamond* machine. The LDBC benchmark driver is located on another *bricks* machine, which is not the same machine (or machines) on which the Titan graph database is located. The specifications of the *diamonds*- and *bricks* machine are listed in Table 3.

Machine name	**diamonds**	**bricks**
CPU	GenuineIntel 96 cores 2.4GHz (2.9GHz turbo)	GenuineIntel 32 cores 2.0GHz (2.8GHz turbo)
Memory	1024GB (1TB)	256GB
Disk	4x2TB	4x2TB (HDD) 8x128GB (SSD)
Network	2x10GB/s (ethernet) 4x40GB/s (infiniband)	1GB/s (ethernet) 40GB/s (infiniband)

Table 3: Machine specifications of the *diamonds* and *bricks* machines of CWI's SciLens cluster.

The following to subsections provide more information about running the LDBC benchmark on Titan and MonetDB.

8.5.1 Running the Titan benchmark

The Titan graph database is located on one or more *bricks* machines from the CWI SciLens cluster. For this research, version 0.5.4 of Titan (with Hadoop version 2) has been used.

8.5.2 Running the Monet benchmark

The MonetDB database is located on a single *diamonds* machine from the CWI SciLens cluster. For this research, version 11.21.5 of MonetDB has been used.

8.6 Analyzing benchmark results

The next subsections will contain the results of the LDBC benchmark runs for both Titan and MonetDB. For every data scale factor, graphs show the number of queries executed in total (*count*), the mean execution time per executed query in milliseconds (*mean*) and the min- and max execution times per executed query in milliseconds (*min/max*).

8.6.1 LDBC benchmark, SF1

The results of the LDBC benchmark on scale factor 1 (SF1) can be found in the following three figures in appendix A:

- **Complex**: see Figure 33
- **Short**: see Figure 34

[28]See https://www.monetdb.org/wiki/Scilens-configuration-standard

- **Update**: see Figure 35

9 Discussion & Conclusion

Research questions:

- **Future**: based on the research that has been conducted for this master thesis, what future work could be performed in order to extend on this research? How would future work relate to this research?

9.1 Discussion

9.2 Conclusion

9.3 Future work

References

[1] http://www.monetdb.org/Home

[2] http://thinkaurelius.github.io/titan/

[3] http://www.obi4wan.nl/

[4] http://www.elastic.co/products/elasticsearch

[5] http://www.elastic.co/guide/en/elasticsearch/guide/current/index.html

[6] http://www.elastic.co/guide/en/elasticsearch/guide/current/match-query.html

[7] http://lucene.apache.org/core/

[8] http://www.thinkaurelius.com/

[9] http://aws.amazon.com/ec2/instance-types/

[10] http://www.monetdb.org/Home/Features

[11] http://www.tinkerpop.com/

[12] http://github.com/tinkerpop/gremlin/wiki

[13] http://github.com/tinkerpop/gremlin/wiki/The-Benefits-of-Gremlin

[14] http://www.github.com/tinkerpop/gremlin/wiki/Getting-Started

[15] httpw://www.github.com/tinkerpop/blueprints/wiki

[16] http://www.neo4j.com/

[17] http://www.neo4j.com/product/

[18] http://www.sparsity-technologies.com/

[19] http://www.neo4j.com/developer/cypher-query-language/

[20] http://www.github.com/twitter/flockdb

[21] http://www.github.com/twitter/flockdb/blob/master/doc/demo.markdown

[22] http://www.neo4j.com/docs/stable/cypher-introduction.html

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

[24] http://www.w3.org/TR/sparql11-property-paths/

[25] http://www.tpc.org/information/benchmarks.asp

[26] http://snap.stanford.edu/data/index.html#socnets

[27] Aurelius. "Titan Documentation", http://s3.thinkaurelius.com/docs/titan/0.9.0-M2/ (2015).

[28] http://s3.thinkaurelius.com/docs/titan/0.5.4/sequencefile-io-format.html

[29] http://s3.thinkaurelius.com/docs/titan/0.5.4/titan-io-format.html

[30] http://s3.thinkaurelius.com/docs/titan/0.5.4/graphson-io-format.html

[31] http://s3.thinkaurelius.com/docs/titan/0.5.4/edgelist-io-format.html

[32] http://s3.thinkaurelius.com/docs/titan/0.5.4/rdf-io-format.html

[33] http://s3.thinkaurelius.com/docs/titan/0.5.4/script-io-format.html

[34] http://s3.thinkaurelius.com/docs/titan/0.5.4/cassandra.html

[35] http://hbase.apache.org/

[36] https://www.altamiracorp.com/blog/employee-posts/handling-big-data-with-hbase-part-3-architecture-overview

[37] https://www.mapr.com/blog/in-depth-look-hbase-architecture#.VgJmC_mqpBc

[38] http://lucene.apache.org/core/3_0_3/fileformats.html#Index%20File%20Formats

[39] https://www.youtube.com/watch?v=T5RmMNDR5XI

[40] https://www.elastic.co/guide/en/elasticsearch/guide/current/intro.html

[41] https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

[42] https://github.com/ldbc/ldbc_snb_interactive_validation

[43] http://www.tinkerpopbook.com/

[44] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D., *A Comparison of a Graph Database and a Relational Database - A Data Provenance Perspective*, ACMSE '10, 2010

[45] Angles, R., *A Comparison of Current Graph Database Models*, 2012

[46] Jouili, s., Vansteenberghe, V., *An empirical comparison of graph databases*, 2013

[47] Holzschuher, F., Peinl, R., *Performance of Graph Query Languages - Comparison of Cypher, Gremlin and Native Access in Neo4j*, EDBT/ICDT '13, 2013

[48] Labute, M.X., Dombroski, M.J., *Review of Graph Databases for Big Data Dynamic Entity Scoring*, Lawrence Livermore National Laboratory, 2014

[49] Hartig, O., *Reconciliation of RDF* and Property Graphs*, University of Waterloo, 2014

[50] Boncz, Peter, Irini Fundulaki, Andrey Gubichev, Josep Larriba-Pey, and Thomas Neumann. "The linked data benchmark council project." Datenbank-Spektrum 13, no. 2 (2013): 121-129.

[51] Angles, Renzo, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. "The linked data benchmark council: A graph and RDF industry benchmarking effort." ACM SIGMOD Record 43, no. 1 (2014): 27-31.

[52] Erling, Orri, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. "The LDBC Social Network Benchmark: Interactive Workload." In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 619-630. ACM, 2015.

[53] Capota, Mihai, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. "Graphalytics: A Big Data Benchmark for Graph-Processing Platforms." (2015).

[54] Van Laarhoven, Twan, and Elena Marchiori. "Network community detection with edge classifiers trained on LFR graphs." (2013)

[55] Lancichinetti, Andrea, Santo Fortunato, and Filippo Radicchi. "Benchmark graphs for testing community detection algorithms." Physical review E 78, no. 4 (2008): 046110

[56] Clauset, Aaron, Cosma Rohilla Shalizi, and Mark EJ Newman. "Power-law distributions in empirical data." SIAM review 51, no. 4 (2009): 661-703

[57] Steven Woudenberg. "Using the Monet database system as a platform for graph processing." Inf/Scr-10-15, Utrecht University, Faculty of Science (2010)

[58] Prat-Pérez, Arnau, David Dominguez-Sal, and Josep-LLuis Larriba-Pey. "High quality, scalable and parallel community detection for large real graphs." In *Proceedings of the 23rd international conference on World wide web*, pp. 225-236. ACM, 2014.

[59] Pons, Pascal, and Matthieu Latapy. "Computing communities in large networks using random walks." In *Computer and Information Sciences-ISCIS 2005*, pp. 284-293. Springer Berlin Heidelberg, 2005.

[60] Rosvall, Martin, and Carl T. Bergstrom. "Maps of random walks on complex networks reveal community structure." *Proceedings of the National Academy of Sciences* 105, no. 4 (2008): 1118-1123.

[61] Lancichinetti, Andrea, Filippo Radicchi, José J. Ramasco, and Santo Fortunato. "Finding statistically significant communities in networks." *PloS one* 6, no. 4 (2011): e18961.

[62] Ahn, Yong-Yeol, James P. Bagrow, and Sune Lehmann. "Link communities reveal multiscale complexity in networks." *Nature* 466, no. 7307 (2010): 761-764.

[63] Yang, Jaewon, and Jure Leskovec. "Overlapping community detection at scale: a nonnegative matrix factorization approach." In *Proceedings of the sixth ACM international conference on Web search and data mining*, pp. 587-596. ACM, 2013.

[64] Prat-Pérez, Arnau, and David Dominguez-Sal. "How community-like is the structure of synthetically generated graphs?." In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pp. 1-9. ACM, 2014.

[65] Abadi, Daniel, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. *The design and implementation of modern column-oriented database systems.* Now, 2013.

[66] Manegold, Stefan, Martin L. Kersten, and Peter Boncz. "Database architecture evolution: mammals flourished long before dinosaurs became extinct." *Proceedings of the VLDB Endowment* 2, no. 2 (2009): 1648-1653.

[67] Mattos, Nelson M. "Sql99, sql/mm, and sqlj: An overview of the sql standards." *IBM Database Common Technology* (1999).

[68] Eisenberg, Andrew, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. "SQL: 2003 has been published." *ACM SIGMoD Record* 33, no. 1 (2004): 119-126.

[69] Rodriguez, M.A., *Titan Provides Real-Time Big Graph Data*, ThinkAurelius, 2012

[70] Rodriguez, M.A., *Educating the Planet with Pearson*, ThinkAurelius, 2013

[71] Siciliani, T., *Lambda Architecture for Big Data*, DZone (java.dzone.com), 2015

[72] Kolodziejski, M., *Get to know the power of SQL recursive queries*, Verbatelo.com, 2013

[73] Boncz, P.A., *Monet - A next-Generation DBMS Kernel For Query-Intensive Applications*, 2002

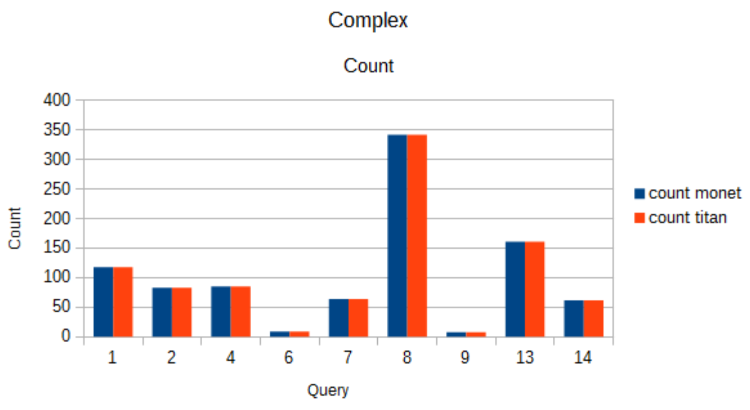
[74] Sparsity Technologies, *User Manual Sparksee*, 2015

[75] Ruohonen, K., *Graph Theory*, 2013

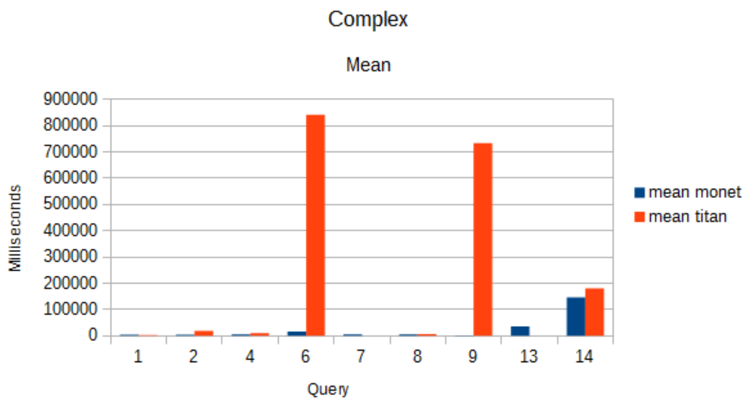
[76] Apache Solr. "Apache Solr Reference Guide - Covering Apache Solr 5.3." (2015)

A LDBC benchmark results

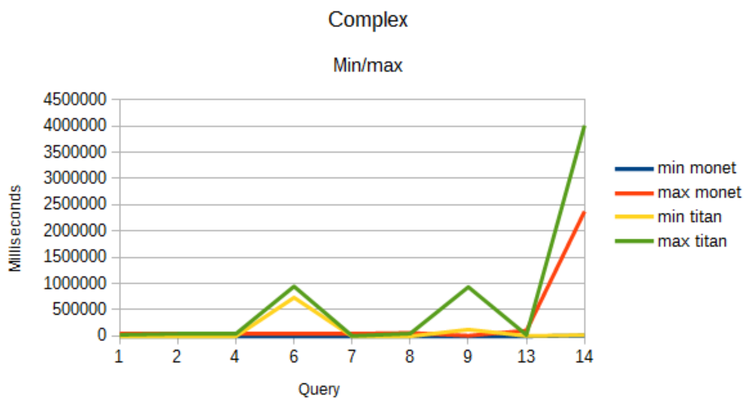
This appendix contains the results of the LDBC benchmarks on various data scale factors, visualized in graphs. Per scale factor, the number of executed queries per query type (*count*), the mean execution time per query type in milliseconds (*mean*) and the min- and max execution time per query type in milliseconds (*min/max*) are shown.



(a) Count

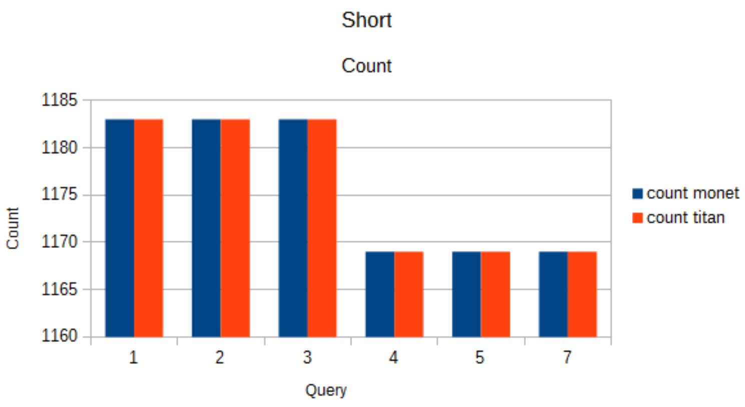


(b) Mean

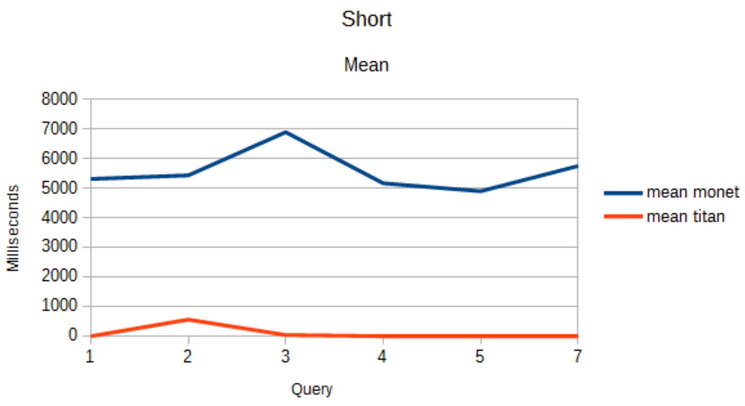


(c) Min/max

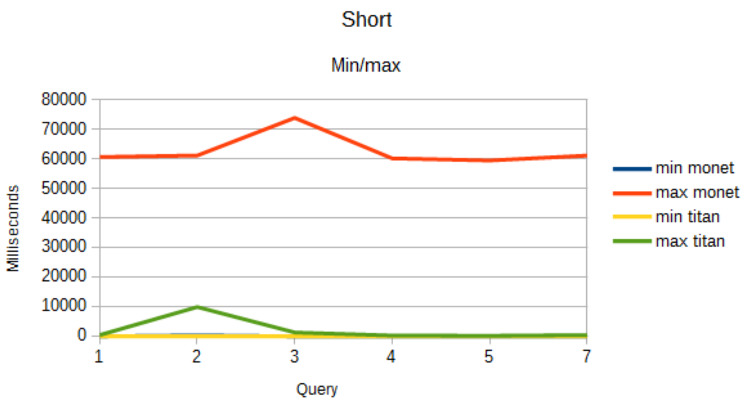
Figure 33: LDBC benchmark on SF1 data, complex queries



(a) Count

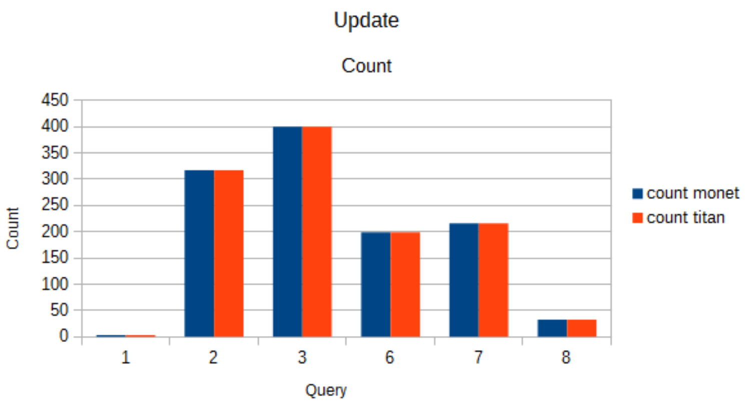


(b) Mean

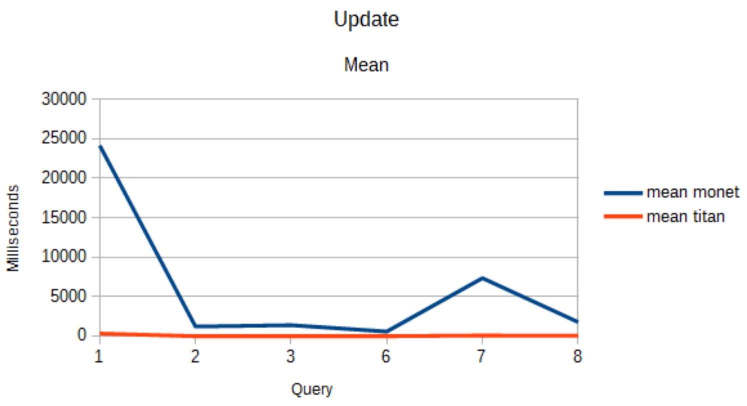


(c) Min/max

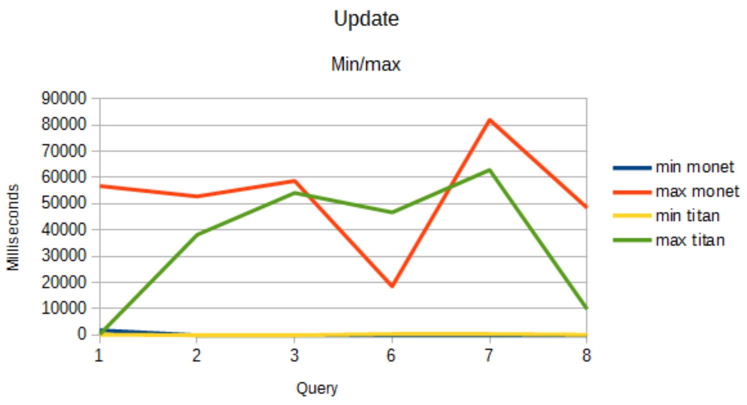
Figure 34: LDBC benchmark on SF1 data, short queries



(a) Count



(b) Mean



(c) Min/max

Figure 35: LDBC benchmark on SF1 data, update queries

B Planning

- **Total workload**: 20 weeks (mid-March to end-July)

Titan

- **Total workload**: 6 weeks (mid-March to end-April)
- Studying and at the same time implementing Titan. Detailed description of what Titan is, how it works (especially internally, by researching the source code and documentation), and how to work with it (translating data into graph format, and querying this graph). **Workload**: 3 weeks
- Study of the integration of ElasticSearch in Titan. How is ElasticSearch integrated (to what level, e.g. which features of ElasticSearch are useable in Titan, and where are they attached to Titan exactly [researching the source code and documentation]), and how can they be used. **Workload**: 2 week
- Based on the integration of ElasticSearch in Titan, study how the current OBI4wan-data in ElasticSearch can be translated to a (graph) data format supported by Titan, preferably in the easiest way possible. Also implement this transition. **Workload**: 1 week
- Titan also supports the distribution of a graph over multiple nodes. Study how to distribute the created graph database with OBI4wan-data over multiple nodes, then implement this distribution. Results from benchmarks (created later on) should show if distributing the data results in optimized execution (faster response times). **Workload**: 2 weeks

MonetDB

- **Total workload**: 5 weeks (end-April to end-May)
- Study of MonetDB, but not necessarily as thorough as the study on Titan. MonetDB is more of a base reference in the benchmarks, and not the main focus of the research. Also implement MonetDB: set up a Monet database and query it. **Workload**: 2 weeks
- Study how the current OBI4wan-data in ElasticSearch can be translated to a data format supported by MonetDB (preferably in the easiest way possible). Then do this transition. **Workload**: 1 week
- MonetDB also supports the distribution of a graph over multiple cores on the same machine. Study how to distribute the created graph database with OBI4wan-data over multiple cores, and execute this distribution. Benchmarks (created later on) should show if distributing the data results in optimized execution (faster response times, compared to both the single-server and distributed solution of Titan). **Workload**: 2 weeks

Gremlin

- **Total workload**: 5 weeks (end-May to end-June)
- Detailed study of the graph query language Gremlin. As there is no grammar (or detailed description of the syntax and semantics) of Gremlin yet, we will create one in this project. See (Angles, Barceló, 2013) for a potential starting point. Become acquainted with Gremlin by using it (see Gremlin documentationo). **Workload**: 4 weeks

- With the grammar in place, study the complete feature set of Gremlin, and detect shortcomings with respect to the benchmarks (and thereby practical queries) that will be executed (e.g. is Gremlin powerful enough to translate the designed benchmarks into queries?). **Workload**: 1 week
- When Gremlin is found not complete and powerful enough for some benchmarks, study the possibility of extending Gremlin with a plugin to add the missing requirements. If possible, implement these plugins. **Workload**: 1 week (optionally)

Benchmarks

- **Total workload**: 4 weeks (end-June to end-July)
- Think of (and design) questions (10-20) that OBI4wan would like to ask to the data in the graph database. This should be done from a business perspective. These questions must be translated into queries, which could fall into categories introduced in (Angles, 2012): adjacency queries, reachability queries, pattern matching queries and summarization queries. **Workload**: 1 week (coming up with questions also job for Frank)
- The execution of benchmarks (and the storage of [graph] data) for both Titan and MonetDB can be done on various platforms: a cluster on Amazon, or a (big, non-distributed) local server at OBI4wan or at CWI. Research the two possibilities, and choose the best option (or use both and compare). **Workload**: 1 week
- When a platform has been chosen, execute the benchmarks. The benchmarks can scale in size by picking a smaller or larger part of the complete OBI4wan dataset (specifying a timeframe). Not only execute the 'question-benchmarks' (based on the questions designed from the business point of view), but also benchmark the loading of data into a Titan graph. How fast is this? **Workload**: 2 weeks