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

December 22, 2015

Abstract

As social networks become ever more important, companies and other organizations are more and more interested in how people are talking about them on these networks. The Dutch company OBI4wan delivers a complete solution for social media monitoring, webcare and social analytics. This solution provides answers regarding for example who is talking about a company or organization, and with what sentiment. Some of the questions that OBI4wan wants to answer require another way of storing the social data set, because the normal "relational" way of storing does not suffice. This report compares the column store MonetDB to the graph database Titan, and tries to find an answer to the question: is a distributed solution (Titan) better than a centralized solution (MonetDB) when it comes to answering graph queries on a social network. The LDBC data set and its benchmark are used to answer this question. Because the benchmarks have shown that both MonetDB and Titan have their respective problems, another centralized database solution (Virtuoso) has been added to the comparison. For now, Virtuoso seems to be the best choice (out of the three systems) for loading the LDBC data set into a database and executing the LDBC benchmarks on this database.

Contents

1 Introduction

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. This big pile of data could prove to be useful for companies and organizations, but without any form of organization, it is hard to obtain useful information from this gigantic stream of data. This introduces the need for a system that can perform this "data-to-information" transformation. 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 we focus on in this report 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 7.6 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. The benchmarks can be executed on any cluster that provides the right hardware for the databases under test. For this research, we have the possibility to use the SciLens cluster of the Dutch Centre for Mathematics and Computer Science (Centrum Wiskunde en Informatica, CWI). Finally, section 9 discusses and concludes this research's findings.

1.1 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **Architecture**: what is the best platform to work on, both for storing the data and executing the benchmarks using Titan and MonetDB? What are the differences between platforms? Advantages? Disadvantages?
  - Because the supervisor of this research works with the CWI, and the CWI provides the right hardware for executing benchmarks on Titan and MonetDB, we have chosen this hardware and omitted the investigation of other platform possibilities.
- **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?

2 OBI4wan: Webcare & Social Media Monitoring

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 comparison between centralized- versus decentralized databases is described in section 2.5.

2.1 OBI4wan Twitter data set

The Twitter data set of OBI4wan consists of a collection of tweets, which were posted between 2009 and now. The majority of this tweets data set is obtained by using the Twitter Streaming API, which provides access to the stream of tweets that are continuously posted onto Twitter. Tweets are retrieved from this stream by filtering on a collection of (primarily Dutch) keywords, and then stored by OBI4wan. A small subset of this whole 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 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.

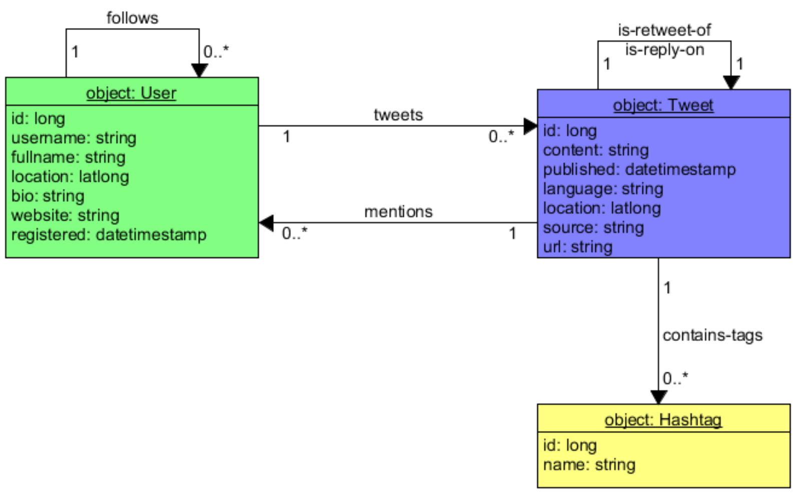


Figure 1: Schema of the OBI4wan data set of Twitter messages.

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 it is likely that the content of these two sets has 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).

Furthermore, FO_x and FR_x are the sets of unique followers and unique friends, respectively, from the OBI4wan user subset $OTOPx$. Formally:

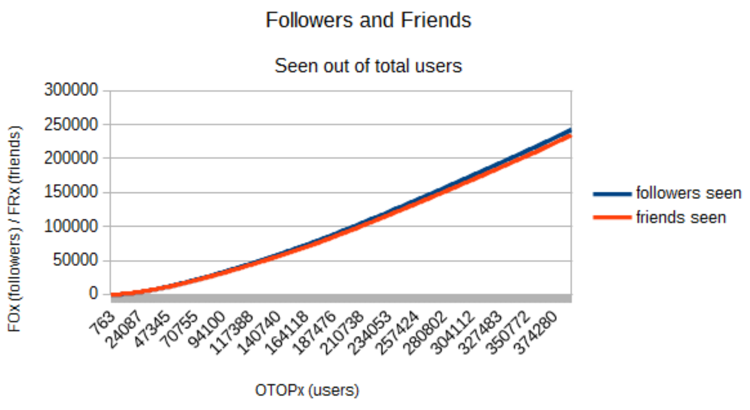
- $FO_x = \{u.Fo|u \in OTOPx\}$
- $Fr_x = \{u.Fr|u \in OTOPx\}$

Over time, the set of followers and the set of friends of a user may change, because the user could unfollow other users (causing a change in the friends set of this user) or the user could be unfollowed by other users (causing a change in the follower set of this user). Also, users may remove their accounts and disappear from the TWitter network, of new users may create an account and appear in the Twitter network. In other words, FO_x and FR_x are not static sets, but change of time. For each user, we can keep track of these changes, by introducing a start- and end-timestamp for each follower and friend of this user. The start-timestamp denotes the moment in time when the friend/follow relation has been established; the end-timestamp denotes the moment in time when this relation has been removed. The end-timestamp has no value when the relation is still active. We can now formalize both FO_x and Fr_x. The formal reasoning is equal for both sets, so the definition below only shows it for FO_x.

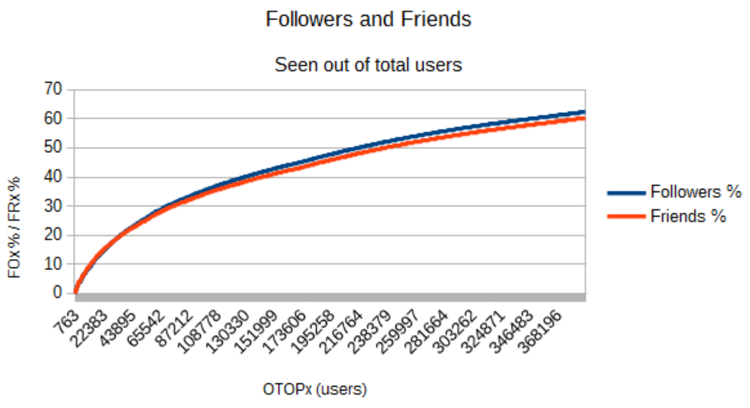
$$FO_x = \{(u, F)|w, u \in OBI \wedge F = \{w, [r_0, ...r_n]\} \wedge r_i = (T_s, T_e)\}$$

In words: (u, F) is a pair of a user u with its set F in the OBI user set. F is defined as a pair of a user w (where $w \neq u$) and an array of timestamp pairs ($[r_0, ..., r_n]$), where each pair r_i is the timestamp pair (T_s, T_e), denoting the start- and end time of a relationship between user u and w (either u following w or the other way around). Using these timestamp sets for each relation, we create an approximation of the real Twitter follower- and friend graphs. It enables us to execute queries that take into account the state of the friend- and follower graphs at various moments in time. An example of such a query is to look for those users who were followed by some other user(s) before they posted a certain tweet (or a set of tweets), and check if this follows-relation is still active some time after the posting of this tweet (or set of tweets). This can tell something about the (negative) impact of the posted tweet(s), measured by number of followers lost.

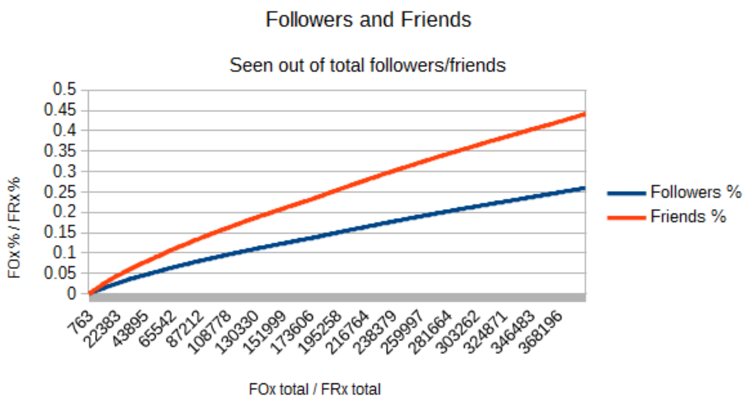
Figure 2a runs over the users in $OTOPx$, and maintains a list of how many of the visited users are also in the (unique) followers- and friends sets. Figure 2b shows the same information, but in percentages of followers and friends out of the users in $OTOPx$ instead of raw numbers. These graphs show that when more users from $OTOPx$ have been visited, also more followers



(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

and friends are visited. These graphs only iterate over the first 400.000 users in $OTOPx$ instead of the total 1 million, so we could assume that when we visit all 1 million users, the number of followers and friends seen out of the total number of users will be close to 100%.

Figure 2c shows the percentage of followers and friends that have been visited out of the total number of users ($OTOPX$), compared to all unique followers and friends as retrieved from the Twitter API. The graph shows that at 400.000 users, 25% of the total set of Twitter followers has been visited, and 45% of the total set of Twitter friends has been visited. These numbers tell something about the completeness of the OBI4wan data set in comparison to the Twitter data set: the OBI4wan data set contains 25% of the complete set of Twitter followers and 45% of the complete set of Twitter friends, given the first 400.000 users in $OTOPx$. So, the OBI4wan data set for the follower- and friend graphs is not the complete data set. This information has to be taken into account as a side note for further analysis on the OBI4wan data set.

2.2 ElasticSearch architecture

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.

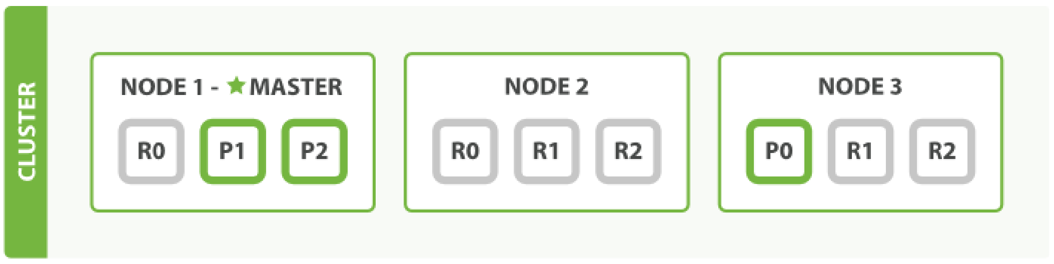


Figure 3: An ElasticSearch cluster with three nodes and three shards per node.

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.

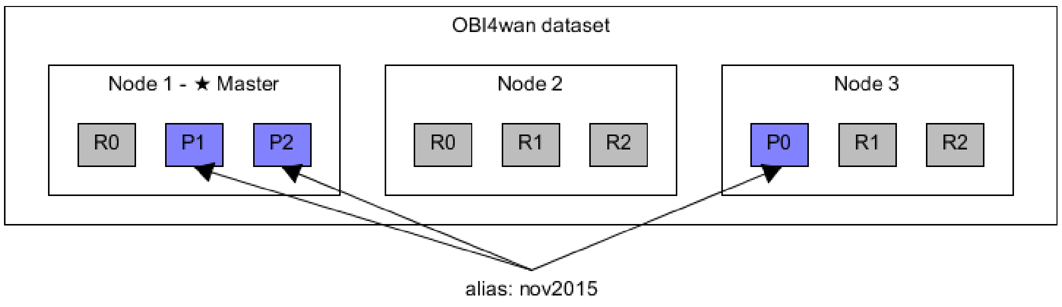


Figure 4: An ElasticSearch cluster as it is used by OBI4wan.

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 properties 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*).

```
{"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 built 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

In addition to the analyses that OBI4wan already does on its data set, there are a lot more possible questions that could be asked to this data set. A few examples of such questions are shown in the list below.

- *Find all Twitter users who are talking about OBI4wan (using "OBI4wan" as a keyword in one of their tweets) and are no more than two follows-relation steps away from the official OBI4wan Twitter account.* This provides insights in which users are talking about OBI4wan, and whether or not they are already connector to the OBI4wan Twitter account. All users who are one follows-relation away from the OBI4wan Twitter account are already connected directly, all users who are two follows-relation steps away are connected to OBI4wan via someone else.
- *Find all Twitter users who are currently following the OBI4wan Twitter account, and recently (within a given period) mentioned the OBI4wan Twitter account (using the @ sign) in one of their tweets/replies, excluding retweets of one of OBI4wan's tweets.* This provides insights in which users are talking about OBI4wan, proactively mentioning the OBI4wan Twitter account.
- *Find all Twitter users who were following the OBI4wan Twitter account before a certain tweet* X *on datetime* Y *(created by the OBI4wan Twitter account), but do not follow OBI4wan anymore in a given period that lies beyond the datetime of the tweet.* This provides insights in how good the tweets posted by the OBI4wan Twitter account are. If OBI4wan has posted a tweet after which many users are unfollowing the OBI4wan Twitter account, then this sort of tweets is apparently not beneficial in promoting the OBI4wan brand.
- *Find all original (hash)tags ([hash]tags that have not been used before a specific datetime) that Twitter users following the OBI4wan Twitter account have attached to tweets in a given period, and give a list of the* n *most occurring (hash)tags as a result.* Tweets that contain tags which have not been used before, are tweets which talk about something new, for example a new development in social networks. This provides insights for OBI4wan about new developments in its branch.
- *Find the top* n *(hash)tags that were used in combination with the keyword "OBI4wan" in a tweet that was posted by Twitter users following the OBI4wan Twitter account.* This provides insights in the topics that users who follow the OBI4wan Twitter account (e.g. the users that are interested in OBI4wan) talk about.
- *Find how many of the tweets posted in a given period by Twitter users following OBI4wan that are replies on tweets that were created by the OBI4wan Twitter account are positive/neutral/negative.* This provides insights in how people are talking about OBI4wan, especially with what kind of sentiment.

2.4 In need of graph query functionality

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 (not specifically taking into account an OBI4wan use case):

```
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. Internally, 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.

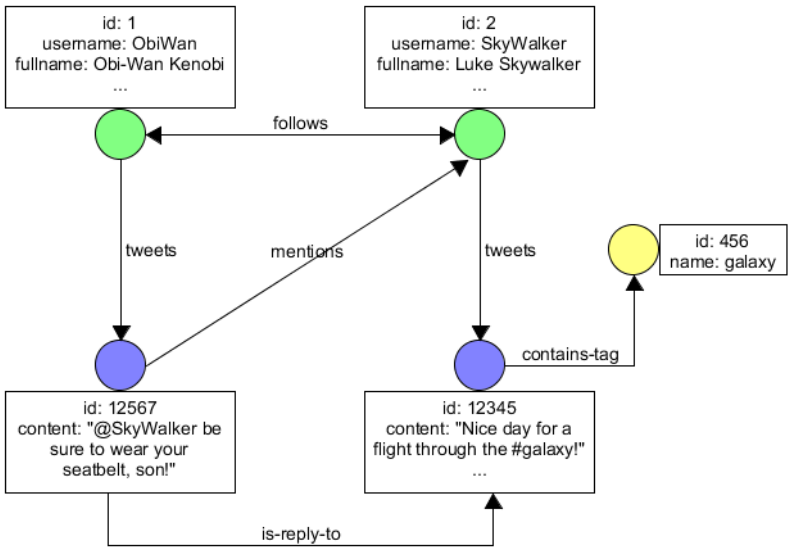


Figure 5: An example data set of Twitter in a graph.

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. 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 Centralized versus decentralized

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 3.4.1 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.

2.6 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **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?

[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

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

3.1 Graph networks

A graph (network) G is a collection of vertices V and edges E, together in the pair of sets $G = (V, E)$ [80]. 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 [49], 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 [49]. 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

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 [74]) 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 [75]) 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 3.3.2). 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 [50] [51]. Another benchmark [52] 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 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 [51] 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" [79]. 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 [79].

[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).

```
[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.

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].

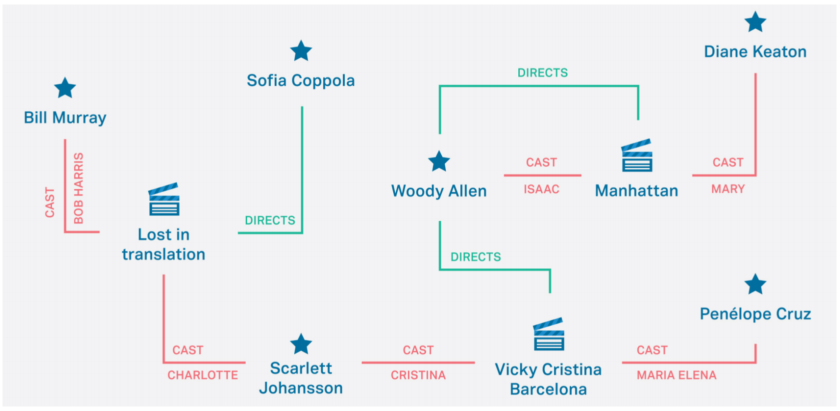


Figure 6: An example of a Sparksee graph.

```
[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.

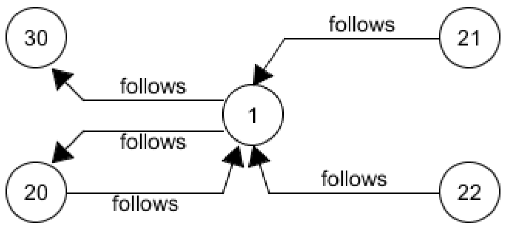


Figure 7: An example graph showing follow-relationships between Twitter users.

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 [53]. 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
```

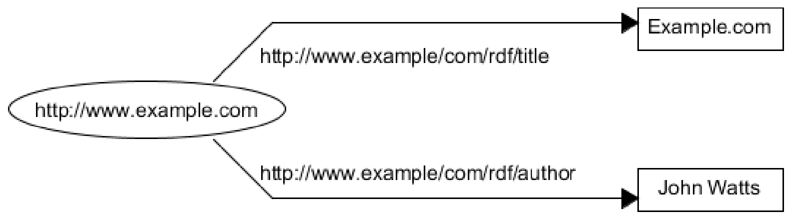

Example.com
http://www.example/com/rdf/title
http://www.example.com
http://www.example/com/rdf/author
John Watts

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

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

3.3 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 [49]:

- **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.

3.3.1 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:

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.

[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].

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.

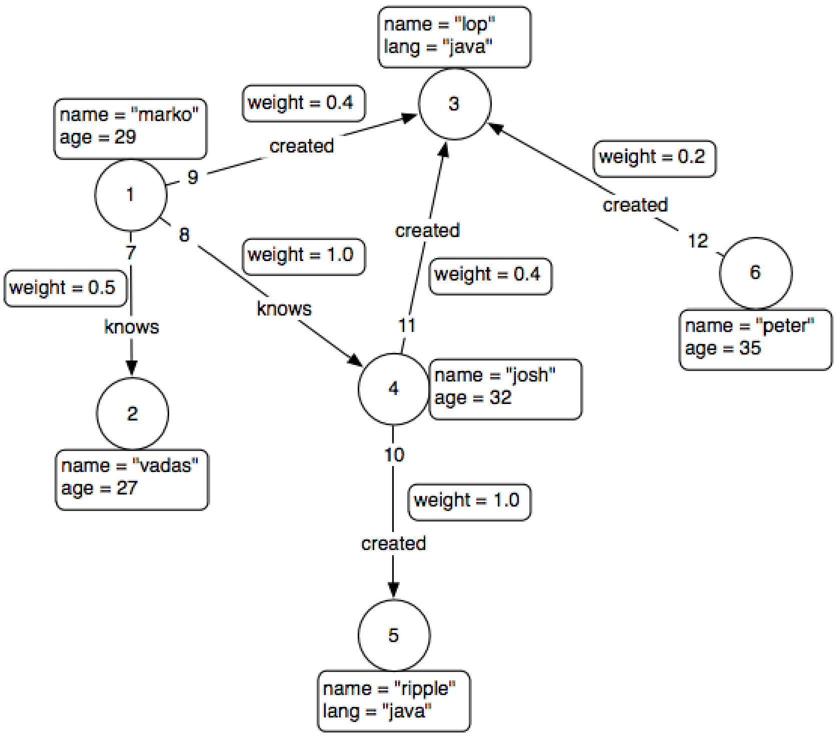


Figure 9: An example property graph database.

3.3.2 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:

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

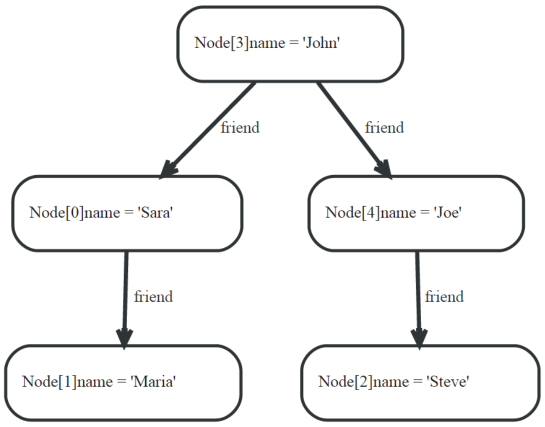


Figure 10: An example property graph database that can be queried by Cypher.

3.3.3 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 3.3.1) and Cypher (section 3.3.2).

3.3.4 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*.

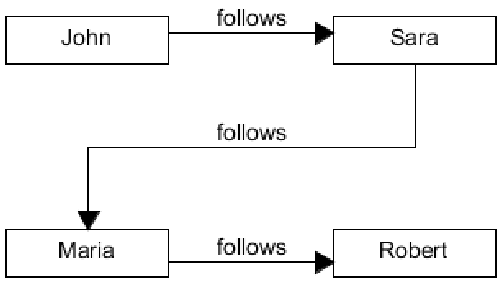


Figure 11: A visual representation of a graph-based table of a relational database.

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 [77].

```
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

3.4 Graph benchmarks

Graph benchmark systems are used to test how well graph database systems performs given a data set and queries that will be executed on this data set. The following subsections describe a couple of these systems: the Linked Data Benchmark Council (LDBC) in section 3.4.1, the Transactional Processing Performance Council (TPC) in section 3.4.2 and Graphalytics in section 3.4.3.

3.4.1 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 [54]. 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) [55].

- 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 [56]. 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).

3.4.2 Transactional Processing Performance Council (TPC)

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.

3.4.3 Graphalytics

Graphalytics is a "big data benchmark for graph-processing platforms" [57]. 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 3.4.1).

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).

3.5 Other database systems

3.5.1 MonetDB

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 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 [78] 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

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

the left side contains an identifier [78]. 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 [78].

3.5.2 Virtuoso

Virtuoso is an object-relational SQL database [44], able to store data in either a row-based format or - like for example MonetDB - a column based format. The database is implemented as a server, which contains a Data Management layer and a Web & Internet layer. The former is responsible for managing the data and querying it, using for example the SQL query language; the latter is to expose this data layer to the web and thereby to its users.

Virtuoso is not used as one of the databases-under-test in this research, but it has been used in other research projects that implemented database- and query handlers to use Virtuoso in LDBC benchmarks - specifically the Social Network Benchmark. These handlers are open source and can be found in LDBC's GitHub repository[6]. The query handler implementations that can be found in this repository have been taken as a starting point to create query handlers for MonetDB, which is one of the databases-under-test in this research project.

3.6 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.

[6]https://github.com/ldbc/ldbc_snb_implementations

3.7 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **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?
- **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?

4 Titan

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[7], HBase[8] and BerkeleyDB[9], and indexing adapters for ElasticSearch[10] and Lucene[11]. Titan has implemented the TinkerPop Stack[12] 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.

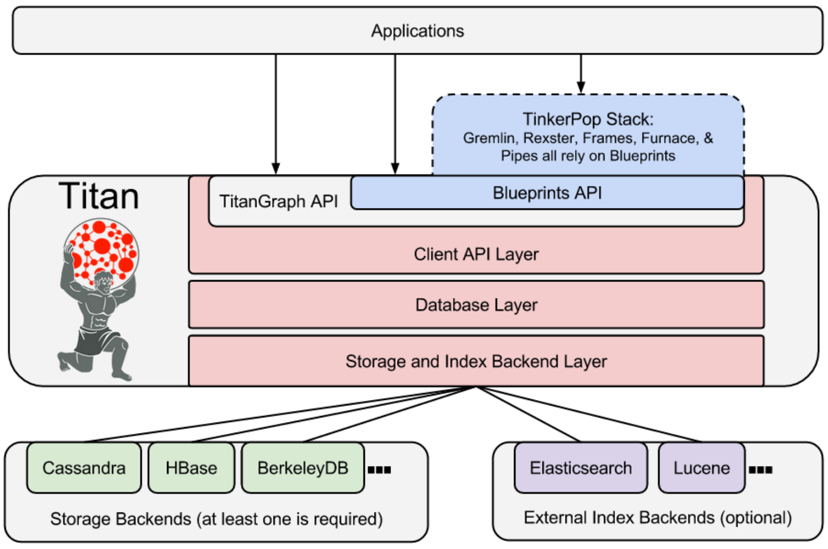

Applications
TinkerPop Stack:
Gremlin, Rexster, Frames, Furnace, &
Pipes all rely on Blueprints
Titan
TitanGraph API
Blueprints API
Client API Layer
Database Layer
Storage and Index Backend Layer
Cassandra
HBase
BerkeleyDB
Storage Backends (at least one is required)
Elasticsearch
Lucene
External Index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

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

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

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.
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 (see below), for example Cassandra and HBase (which are storage backends that are natively supported by Titan). 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.

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.

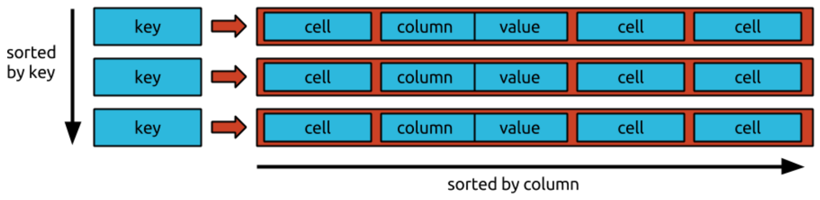


Figure 13: The BigTable data model.

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.

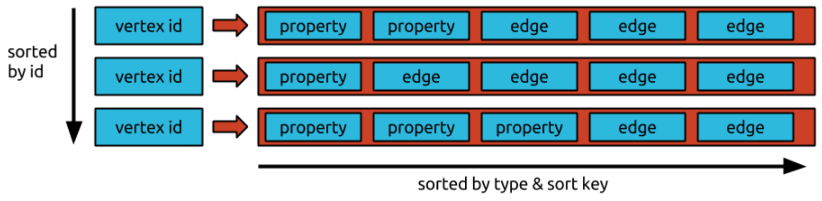


Figure 14: The Titan storage layout, based on the BigTable data model as shown in Figure 13.

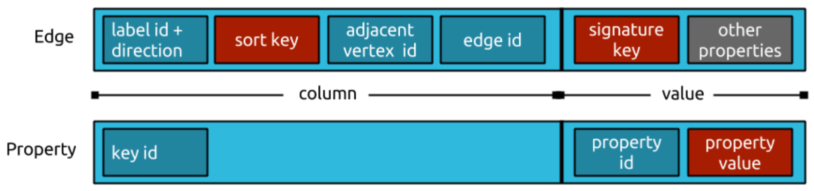


Figure 15: The relation layout inside a BigTable *cell.* shown here for both *properties* and *edges.*

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*.

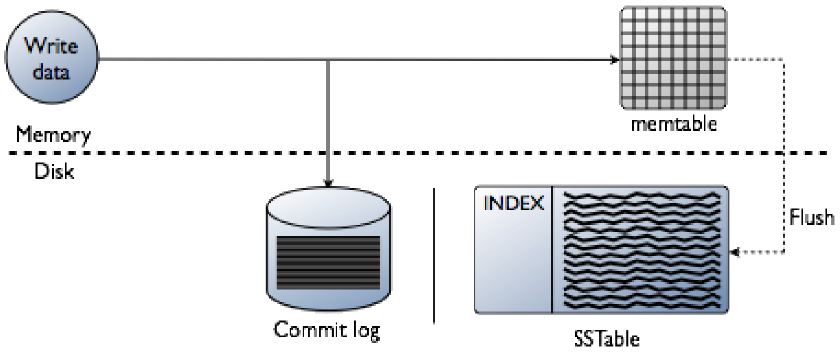


Figure 16: A schematic overview of writing data into Cassandra.

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.A bloom filter is a boolean structure that tells if a piece of data is certainly not inside a certain position (value of zero), or might be in a certain position (value of 1). In this case, the position of data is an SSTable. If the bloom filter tells that a piece of data might be in a certain SSTable, 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).

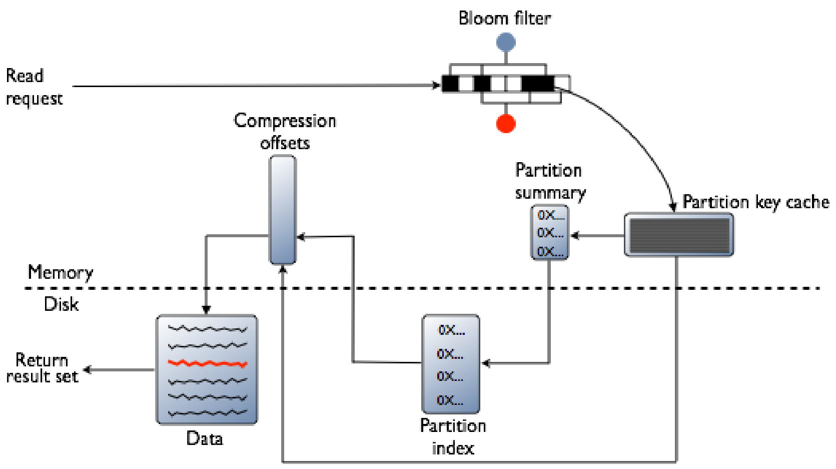


Figure 17: A schematic overview of reading data from Cassandra.

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 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.

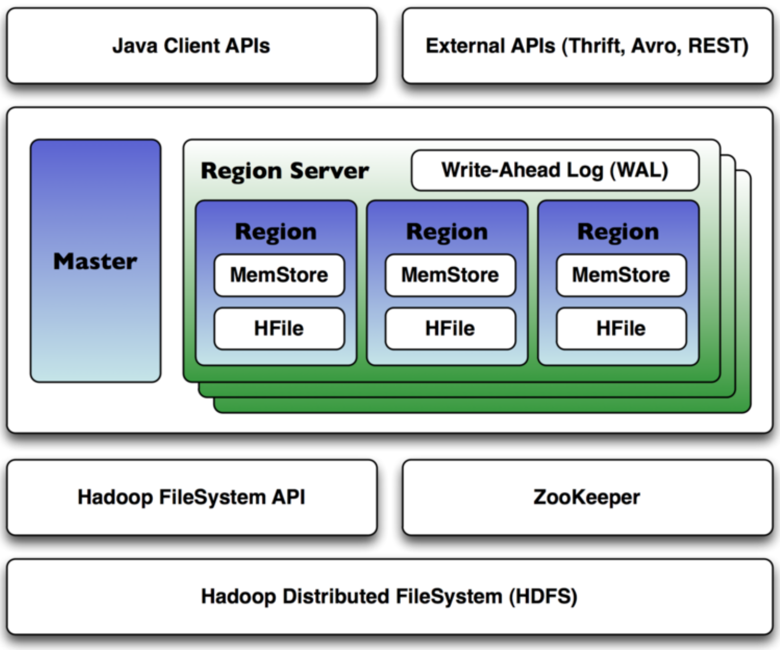


Figure 18: A schematic overview of the HBase architecture.

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 also 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 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 not a good backend solution for this research project and therefore a more detailed description of its internal structure is omitted.

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.

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

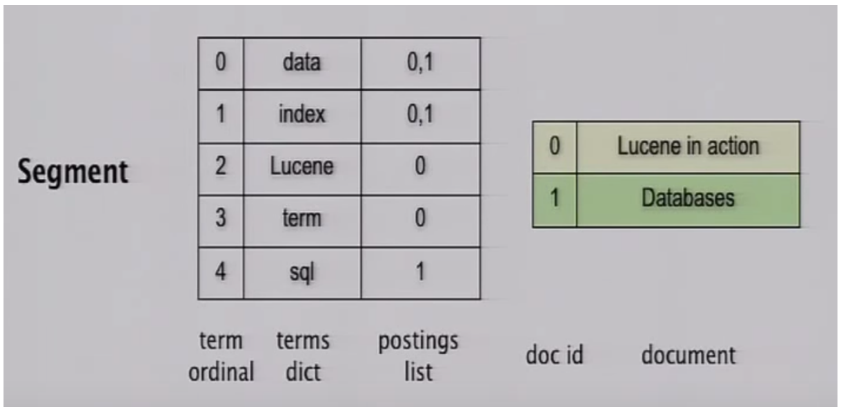


Figure 19: The architecture of the insides of a Lucene Index.

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.

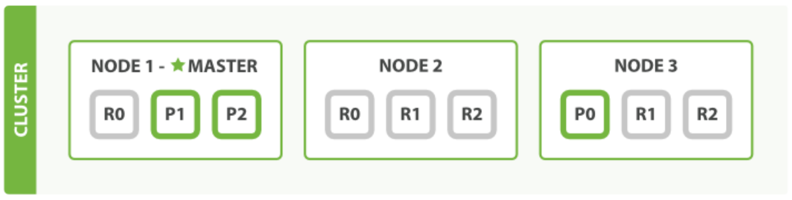


Figure 20: The layout of an ElasticSearch cluster.

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 [81]. 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*.

4.3 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **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.)?
  - This is a big project on its own, and is therefore outside the scope of this report. It would fit as a research question for any future work.
- **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?
- **Architecture**: how is data stored in a Titan graph database?
- **Architecture**: how (to what extend) is ElasticSearch integrated into Titan?
- **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 is 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)?

5 MonetDB

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 MonetDB 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.

Smith	1
Waterman	2
Pinkman	3

Table 2: Example column-based data store

5.2 MonetDB 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 3.5.1).

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 collection 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 used in combination with the SQL query language. The next two 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 [71] and SQL'03 [72] definitions. However, the actually supported definitions depend largely on the requirements of MonetDB's user base. Therefore, some SQL features that are defined in the '99- and '03 definitions are not supported in MonetDB (although this has not introduced problems for the research in this project), and some of the language constructs supported by MonetDB are not contained in SQL'99- and SQL'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[13].

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[14].

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 [69]. 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 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 BAT table *name*, using these IDs.

BATs are stored in two memory arrays: one for the surrogate (head) and one for the value (tail). In the case that the values that are stored in the tail array are of variable length, 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 [70].

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

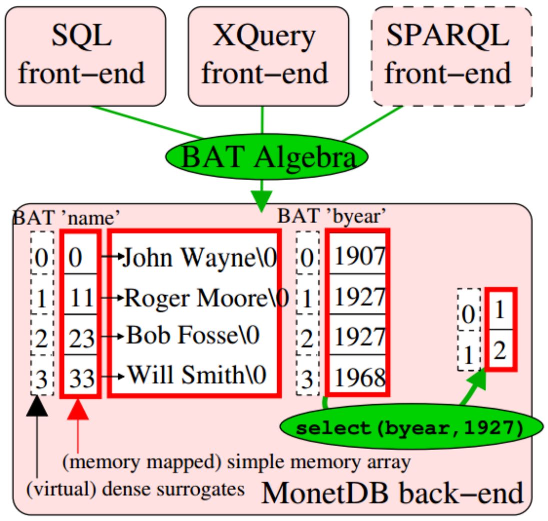


Figure 21: The architecture of MonetDB's BAT tables.

5.2.3 Hash tables

In (relational) databases, queries which contain a join between multiple tables are generally using hashing techniques for faster lookups between the joined tables. MonetDB also uses a hash-join algorithm for joining tables. The basic algorithm of creating a hash-join consists of the following steps, given two tables L and R:

1. Tables L and R are partitioned using a hash function on the join attribute. These partitions are then stored on disk, with the assumption that every created partition can be stored into main memory completely.
2. Each partition pair (containing all data with the same hash function result, e.g. the same attribute used for the join) is loaded into main memory, and a hash table is created for the smallest partition.
3. The other (larger) partition finds matches on the attribute used in the join using the hash table of the smaller partition.

This hash-join algorithm results in faster lookups in case of joined tables. MonetDB automatically creates new hash tables for joins when they first occur in a query, and are reused in future queries that use the same join.

5.2.4 Optimizer pipelines: the mitosis problem

As discussed in section 5.2.1, MonetDB uses an optimization layer, consisting of a collection of optimization modules which are used to let MAL queries execute as efficiently as possible. One of these optimizations is the *mitosis* pipeline, which splits up the columns of a table and divides them over multiple CPUs, in order to achieve a parallel query execution. While this technique is beneficial for some queries (with no dependencies to columns of other tables), it results in significantly reduced execution time for others. The following example shows a situation in which the mitosis optimization decreases the performance.

Consider two database tables: a *person* table containing information about persons (with columns *id*, *firstname* and *lastname*) and a *knows* table (with columns *personid1* and *personid2*), which contains an entry between two persons if they know each other. These two tables are visualized and filled with some dummy data in Table 3.

id	**firstname**	**lastname**
1	John	Smith
2	Sara	DeGeneres
3	Pablo	Garcia

personid1	**personid2**
1	2
1	3
2	3

Table 3: An example of *person* table (on the left) containing information about persons, and a *knows* table (on the right), containing a relation entry between two persons if they know each other.

Consider a query that takes a person ID as input (the start entity), and outputs all persons with a certain first name (also given as input) who are exactly three "knows-steps" away from the start person entity (without visiting a person twice on such a path). This query requires a join between the *person* table and the *knows* table, on the *person* ID attribute and the *knows* personid1/personid2 attributes. MonetDB's hash-join achieves this join by first partitioning all the joins, and then creating hash tables for fast lookups. But then, the mitosis optimizer also partitions the two tables on their columns, with the idea of dividing the total amount of work over multiple CPU cores, which theoretically results in faster performance. However, the mitosis optimizer removes the earlier created hash tables and creates new ones for the partitioned tables, which is a costly process. It gets even worse, because the partitioned tables and their new hash tables are not stored. So for every time a query like this is executed, the mitosis algorithm again partitions the tables and again creates new hash tables. With relatively large tables, the time needed by the mitosis optimizer to partition tables and create new hash tables could become larger than the actual query execution, resulting in unwanted overhead. To conclude: this example shows that the mitosis optimizer and the resulting partitioning of table columns is not always beneficial towards the query execution.

5.2.5 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.

5.3 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **Architecture**: what is the best way to store Twitter data in a MonetDB data store? More general: how is data stored in MonetDB?
- **Architecture**: how does MonetDB work internally? What is the underlying architecture of MonetDB (column-store)?
- **Architecture**: how (in what format) does MonetDB store its data?
- **Architecture**: how can data that is stored in a MonetDB database be queried? What is MonetDB'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 MonetDB database from a remote location? How much latency does this introduce, and is the latency a limiting factor?

6 Test data generation

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. In the light of OBI4wan, the best source of data would be their collected Twitter data, but the problem is that this data set is not complete. While the OBI4wan Twitter data set does contain information about *how many* followers and friends a user has (because this number is attached to each collected tweet), who these followers and friends *are exactly* (the actual followers- and friends relations) are not contained in the raw tweet data, which means that this data has to be retrieved separately. The friends and followers of a user can be retrieved by using the Twitter API, but this would be a process of months due to the limits Twitter poses on its API. Therefore, 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)[15], or (social) network data generators like Lancichinetti-Fortunato-Radicchi (LFR)[16] and LDBC DATAGEN[17]. 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 [58]. 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 [58] [60] . 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 [59].

[15]http://snap.stanford.edu/data/index.html#socnets
[16]https://sites.google.com/site/andrealancichinetti/files
[17]https://github.com/ldbc/ldbc_snb_datagen

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) [56]. 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 data source to use

The final choice for a data set to use should be one that is as close to the OBI4wan data set as possible. In other words, the data set that will be used in this project should be a data set that resembles the likes of a real social network. When this is taken into account, the SNAP- and LDBC data sets are the best options. Between these two, LDBC has the advantage of also providing a benchmark system in which the LDBC data set can be used. For this reason, the final choice for the data set to use in this project is the one from LDBC.

As mentioned before at the beginning of this section, at the start of this project the raw data set of tweets as collected by OBI4wan was not fully complete. Over time, the friends- and followers of users from the OBI4wan data set have been collected, which completed the initial OBI4wan data set. In other words: at some point during this project we *did* have enough friends and followers to use in combination with the initial OBI4wan data set, which would in theory cancel the need of another test data source. However, we have made the decision to keep using the LDBC data set anyway, and the reason for this is twofold.

First, the LDBC data set is nicely scalable and can generate data in various data sizes. This makes it easy to benchmark both Titan and MonetDB under different loads. While the OBI4wan data set is scalable on the number of tweets, it is much harder to scale on the number of friends and followers. Just taking less users (and their friends and followers) does not work, because the users and their friends and followers should be matched against the users who posted the tweets that are in a certain OBI4wan data subset. In comparison, scalability of the LDBC data set takes care of this automatically, because it only includes those users who have posted a tweet, and no others.

Second, by using the LDBC data set as a proxy for the real Twitter data set of OBI4wan, we can compare the two and evaluate the realism of the LDBC data set. This helps the LDBC in its development towards a data set that is as realistic and close to real social network data sets as possible.

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.
- 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.

6.2.1 Stripping down the LDBC data set

The full LDBC data set is shown in Figure 22. Some vertices and relations that are contained in the LDBC data set are not present in a real Twitter data set, such as the data set of OBI4wan. For example, Twitter data sets do not contain vertices like *City*, *Country*, *Continent* and *University* or *Company*. To make the LDBC data set more similar to the data set of a real Twitter network, we have excluded vertices and edges, resulting in a stripped-down version of the LDBC data set. The vertices and edges that are used in this project are enumerated in Table 4.

Vertices	**Edges**
Person	knows (Person *knows* Person)
Post	likes (Person *likes* Post/Comment)
Comment	replyof (Comment *replyof* Post/Comment)
Tag	hascreator (Post/Comment *hascreator* Person)
	hastag (Post/Comment *hastag* Tag)

Table 4: The vertices and edges of the complete LDBC data set that are used in this project.

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. This research will use the CSV format, as the extra data from the Ntriple format will not be used. 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 23 and 24.

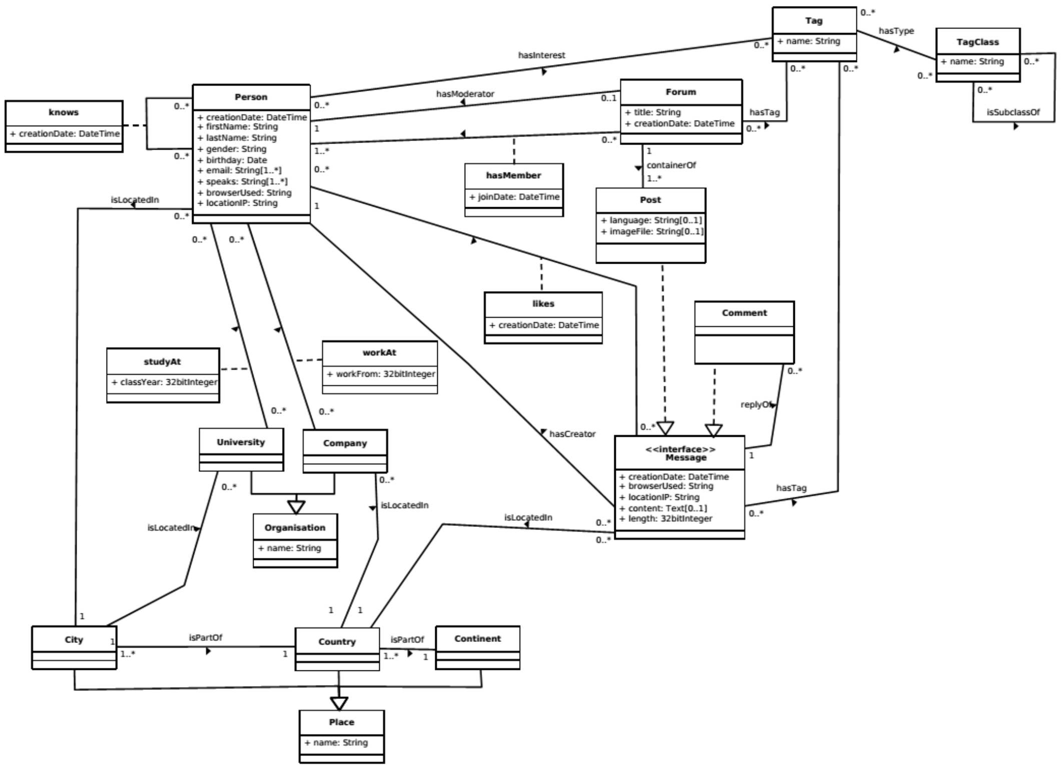


Figure 22: The full LDBC data set, with all vertices and edges.

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

Figure 23: 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 24: 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

Each database has different ways of loading data into databases; the same is true for the two database systems used in this research: Titan and MonetDB. Therefore, the output from the LDBC DATAGEN needs to be translated into a format that is compatible with the respective database systems. The following paragraphs show this translation process for both Titan and MonetDB.

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. Each format is shortly described 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[18]. A Pig script is used to translate the LDBC DATAGEN output data into the user-defined format that is used by Titan's *Script IO.* In the full script, all relation types, property (attribute) types, entity types and data types that are present in the LDBC DATAGEN output are put into 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 specified in a special schema input file. For this project, only a selection of vertices, their respective properties and accompanying edges are specified in this schema input file (see section 6.2.1 for an enumeration of these vertices and edges). In the output files, each line contains exactly one vertex with all of its attributes and relations to other vertices. The total number of output files is at least equal to the total number of vertex types - all vertices of the same type are put into the same output file, although one vertex type can be spread over multiple files to reduce the size per file.

Figure 25 shows an example output line. This line is built up as shown in the list below.

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

- 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 25) 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 25: An example output line created by a Pig script that translates the output from LDBC DATAGEN into a user-defined output format.

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. Data can be loaded into a MonetDB database using the *COPY INTO* statement - this does not require any extra work other than just executing this statement - but loading data into 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 detailed the process of translating the LDBC DATAGEN output data set into a user-defined format that can be used to load the data into a Titan database. This process has created one or more output files for every entity type: *persons*, *posts*, *comments* and *tags*. Titan's *Script IO* format uses these files as input and reads them line by line. The user has to write his own script (written in the Groovy language) to define how each line should be parsed and written into a Titan database. The script that is used for this research takes the steps shown below 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. This special ID is just another vertex attribute which is

indexed, and is of the form $< vertextype > - < vertexid >$ (for example: $person - 12345$). The first step of the user-defined script is to create a new Titan vertex with this ID as its first attribute. In addition, Titan assigns its own internal ID to each vertex, which faces the same uniqueness problem as before. In order to resolve this issue, all IDs are suffixed with a single number that is unique for each vertex type. This ensures that all internal Titan vertex IDs are unique throughout the whole graph. Note that this suffixing solution is only possible when the total number of vertex types is not more than ten. In this case, there are four vertex types, as shown in section 6.2.1: Person, Post, Comment and Tag.

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 known, because it is the same vertex which line is currently processed. The incoming vertex is retrieved by its ID, and the function $addEdge(direction, type, invertexID)$ 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 $addProperty(key, value)$.

6.4.2 Using Hadoop for parallel data loading

Titan uses a Hadoop cluster to load large amounts of data in parallel into a Titan graph. The input data (each line of this data consisting of a vertex with all its properties and incoming- and outgoing edges) is split into multiple parts, which are then divided over the available machines in the Hadoop cluster. Each machine uses the earlier described user-defined input script to read the input data line by line and writes vertices and edges to a Titan graph. This writing process consists of two MapReduce jobs: the first having both a Mapper- and a Reducer-part, and the second only having one Mapper-part. Both jobs and their Mapper- and Reducer steps are listed below.

1. (**Job 1**): for each vertex (e.g. each line of the input data), the pair <temporary ID, permanent Titan-assigned ID> is written into the Hadoop file system (HDFS).
2. (**Job 1**): a Reducer reads this pair and pushes it to all edges, so that Titan knows to which two vertices (incoming and outgoing) an edge is connected. This information is then written into the HDFS.
3. (**Job 2**): a Mapper writes all vertices and edges from the HDFS into the Titan graph backend (Cassandra).

Manually defining Titan's schema An extra MapReduce job is added when the graph's schema has not been manually defined. In that case, the graph's schema (the possible labels and data types for vertices and edges) is inferred automatically. This is done just prior to the final Mapper-step of the second job that writes all vertices and edges to the Titan graph backend. To prevent the extra time that is needed to automatically infer the graph's schema, in this research the schema is manually defined before data is loaded into the graph. Creating a schema for a Titan graph is done by using the graph's management system, which is called by the following statement:

```
mgmt = t.getManagementSystem()
```

First, the possible edge labels and their multiplicity are set. An edge's multiplicity can be of one of the following five settings:

MULTI	Multiple edge labels of the same type between two vertices are possible.
SIMPLE	At most one edge label of a certain type between two vertices is possible.
MANY2ONE	At most one outgoing edge label of a certain type is possible, but there is no constraint on the number of incoming edge labels of this type.
ONE2MANY	At most one incoming edge label of a certain type is possible, but there is no constraint on the number of outgoing edge labels of this type.
ONE2ONE	At most one incoming- and one outgoing edge label of a certain type is possible on vertices.

Next, all properties that can be added to vertices are created in the graph's management system. Each property type has a cardinality setting, which can be *SINGLE* (at most one value for a property) or *LIST* or *SET* (multiple values for a property is possible). The final step in creating the schema is to define the possible vertex types, after which all changes are committed to the management system. The full schema declaration is shown below.

```
// create a new TitanFactory graph on a Cassandra/ElasticSearch backend
println "Opening new TitanFactory"
t = TitanFactory.open("conf/titan-cassandra-es.properties")

// get management system
println "Get ManagementSystem"
mgmt = t.getManagementSystem()

// create edge labels
println "Creating edge labels"
hascreator = mgmt.makeEdgeLabel('hascreator').multiplicity(Multiplicity.MANY2ONE).make()
hastag = mgmt.makeEdgeLabel('hastag').multiplicity(Multiplicity.MULTI).make()
replyof = mgmt.makeEdgeLabel('replyof').multiplicity(Multiplicity.MANY2ONE).make()
knows = mgmt.makeEdgeLabel('knows').multiplicity(Multiplicity.MULTI).make()
likes = mgmt.makeEdgeLabel('likes').multiplicity(Multiplicity.MULTI).make()

// create property keys
println "Creation property keys"

// persons
println "Creating person property keys"
firstname = mgmt.makePropertyKey('firstname').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

lastname = mgmt.makePropertyKey('lastname').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

gender = mgmt.makePropertyKey('gender').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()
```

```
birthday = mgmt.makePropertyKey('birthday').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

// posts and comments
println "Creating post and comment property keys"
imagefile = mgmt.makePropertyKey('imagefile').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

language = mgmt.makePropertyKey('language').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

content = mgmt.makePropertyKey('content').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

length = mgmt.makePropertyKey('length').dataType(Integer.class)
.cardinality(Cardinality.SINGLE).make()

// tags
println "Creating tag property keys"
name = mgmt.makePropertyKey('name').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

url = mgmt.makePropertyKey('url').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

// multiple entities
println "Creating other property keys"
creationdate = mgmt.makePropertyKey('creationdate').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

locationip = mgmt.makePropertyKey('locationip').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

browserused = mgmt.makePropertyKey('browserused').dataType(String.class)
.cardinality(Cardinality.SINGLE).make()

// create vertex labels
person = mgmt.makeVertexLabel('person').make()
post = mgmt.makeVertexLabel('post').make()
comment = mgmt.makeVertexLabel('comment').make()
tag = mgmt.makeVertexLabel('tag').make()

// create indices
type = mgmt.makePropertyKey('type').dataType(String.class).make()
iid = mgmt.makePropertyKey('iid').dataType(String.class).make()
mgmt.buildIndex('byType',Vertex.class).addKey(type).buildCompositeIndex()
mgmt.buildIndex('byIid',Vertex.class).addKey(iid).buildCompositeIndex()
```

```
// commit all changes
mgmt.commit()
```

After the schema has been inferred, the next step is to transfer the graph input data to Hadoop's file system (HDFS), from which it can be loaded into the graph.

Loading data with Hadoop After the graph's schema has been setup and input data has been written to HDFS, the actual data loading with Hadoop can be executed. It is possible to setup a custom Hadoop cluster for this task, as Titan's data loading with Hadoop uses normal MapReduce jobs as described earlier. From Titan's perspective, there are some necessary configuration options to set, such as pointers to the input data, the index- and storage backend, Hadoop's datanode and namenode locations and the default file system location. While setting up the Hadoop cluster to load data into Titan, some issues had to be resolved along the way. The errors which were thrown and the solution that solved the errors are discussed below.

Error: *Container launch failed for container_xxx : org.apache.hadoop.yarn.exceptions.InvalidAuxServiceException: The auxService:mapreduce_shuffle does not exist.*
Solution: It must be explicitly defined that the *mapreduce_shuffle* service is used in MapReduce jobs. This can be set in the *yarn-site.xml* configuration file:

```
yarn.nodemanager.aux-services=mapreduce_shuffle
yarn.nodemanager.aux-services.mapreduce.shuffle.class=org.apache.hadoop.mapred.ShuffleHandler
```

Error: *Error: java.io.IOException: Could not commit transaction due to exception during persistence. Caused by: java.lang.OutOfMemoryError: GC overhead limit exceeded.*
Solution: The main reason behind this problem can be found in the last part of the error message: *GC overhead limit exceeded*. *GC* stands for *garbage collection*. The error is thrown when the JVM that executes (part of) the MapReduce job spends more than 98% of its time doing garbage collection, after which less than 2% of the heap is recovered. To prevent this error, the available heap size for both map- and reduce jobs should be increased. Furthermore, the extra option *-XX:-UseGCOverheadLimit* can be added in the configuration to prevent checks on time spend in garbage collections. All these configuration settings are in the *mapred-site.xml* configuration file:

```
mapreduce.map.java.opts=-Xmx1024m
mapreduce.reduce.java.opts=-Xmx1024m
```

Error: *org.apache.hadoop.mapreduce.Job - Task Id : attempt_xxx, Status : FAILED. Container [pid=17420,containerID=container_xxx] is running beyond physical memory limits. Current usage: 1.0 GB of 1 GB physical memory used; 1.8 GB of 2.1 GB virtual memory used. Killing container.*
Solution: In a previous solution, the heap size for JVM processes was set to 1 GB. However, the default limit for map- and reduce containers is also set to 1 GB, which is the cause for the error message shown above. Best practices for Hadoop state that the JVM heap size should be around 75% of the total container size. In this case, a JVM heap size of 1 GB means that the container size should be around 1.3 GB. In general, first the container size is set, and then the JVM heap size. For example, if the container size is set to 4 GB (4096 MB), then the JVM heap size should be set at $0.75 \times 4096 = 3072$ MB. Container sizes and JVM heap sizes can be set in the *mapred-site.xml* configuration file:

```
mapreduce.map.memory.mb=4096
mapreduce.reduce.memory.mb=4096

mapreduce.map.java.opts=-Xmx3072m
mapreduce.reduce.java.opts=-Xmx3072m
```

Error: *java.lang.OutOfMemoryError: Java heap space*
Solution: Larger data sets generally need more amount of memory (or more data splits divided over more processes). In the case of loading data into Titan, there could be certain vertices which have a lot of incoming- and outgoing edges. During the addition of edges to a vertex, these edges must be kept in memory. When the total data size of all the edges is larger than the set memory limit, then a Java heap space error is thrown. This issue can be resolved by reserving more memory for map- and reduce containers.

Error: *org.apache.hadoop.mapreduce.Job - Task Id : attempt_xxx, Status : FAILED. AttemptID:attempt_xxx Timed out after 600 secs.*
Solution: Vertices with many incoming- and outgoing edges (see the previous error and solution) might cause a timeout, because it takes a relatively long time before the map (or reduce) job has finished. The default timeout setting is set to 600 seconds. In order to solve this error, this default setting can be changed in the *mapred-site.xml* configuration file. A value of zero means that the timeout is disabled:

```
mapreduce.task.timeout=0
```

6.4.3 Loading data into a MonetDB database

Loading the LDBC DATAGEN output into MonetDB is relatively simple. MonetDB 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 23 into a MonetDB 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>';
```

6.5 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **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 MonetDB)? What is the best format into which the LDBC data can be translated (if multiple formats are supported by the database solution)?

7 OBI4wan data usage

This section provides information about how the Twitter data set from OBI4wan is used in the benchmarks that are 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 the OBI4wan data set into an input format that is accepted by the databases under test. Section 7.3 shows how a missing part of the OBI4wan data set - the friends- and followers set of the top users (most active in terms of created messages) 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 the OBI4wan data set and the extra friends- and followers data, which provide information about the structure of the OBI4wan data set. Finally, section 7.2 describes the process of loading the translated OBI4wan data set 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 26. The attributes of interest contained in the JSON are explained in the list 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 26: Example tweet from the OBI4wan archive, in JSON format.

7.2 Translating the OBI4wan data set into database input

Before the OBI4wan Twitter data set can be put into Titan and MonetDB, 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 remaining task would then be to translate the OBI4wan Twitter data set 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 how graph databases store their data.

A Python script is used to translate the OBI4wan Twitter data set 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: *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 two steps.

   (a) 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 timestamp, language and content directly from the tweet's JSON.

   (b) 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 data set. 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 data set

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 with the username or ID of a user as input parameter in order to retrieve a list of friends and followers for that user. The Twitter API provides separate endpoints for retrieving the friends- and for retrieving the followers of a single user: the *GET friends/ids* and *GET followers/ids* endpoints, respectively. Both endpoints allow for the retrieval of at most 5000 entities (friends or followers) per user per API call. When the retrieved friend- and follower sets are stored for each user that occurs in the OBI4wan Twitter data set (e.g. all users who have posted one or more tweets that have been collected by OBI4wan), the friends/followers data set can be constructed.

The two API endpoints which can be called to receive a user's friends/followers are 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 or 5000 followers for one user per minute[19]. With each extra authentication token available, the number of available calls per minute increases with one. For example, with 20 authentication tokens, one can call the two Twitter API endpoints that retrieve a user's friends or -followers 20 times per minute.

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 at least 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 retrieval 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 does not suffice, but for statistical purposes this data set is still interesting to use. For this research the friends/followers data set is needed because some benchmark queries depend on this information. However, it is not required that this information is up-to-date for benchmarking purposes, so also an out-of-date friends/followers data set is good enough. For this research, only the top million users (sorted descending by how many tweets each user has posted) have been taken out from the complete set of 3.5 million users, and for these users their friend- and follower data sets have been retrieved.

In short, the process of keeping the friend- and follower data sets up-to-date is outside the scope of the benchmarks that are executed in this project. On the other hand, it is still interesting to think of algorithms which can be used to keep the data set up-to-date (outside the scope of

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

benchmarks). After all, up-to-date data is essential to perform real-time analysis on this data, which OBI4wan does. Section 7.3.3 proposes an unimplemented algorithm that does this.

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[20] 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 27.

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

Figure 27: 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 27 would translate into the *person_person* relations file as shown in Figure 28.

7.3.3 Keeping the friend- and follower data sets up-to-date

The initial retrieval of the friend- and follower data sets takes a relatively large amount of time. However, when this initial data set has been retrieved, it can theoretically be kept up-to-date using the algorithm described in the steps below. Again, note that this is only a proposed algorithm which is not further implemented in this project.

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

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

Figure 28: Example lines from the *person_person* relations file.

1. After a tweet (T) has been retrieved by OBI4wan, the tweet's JSON (as shown in Figure 26) is stored. This JSON contains - among other attributes - the user (U) who created the tweet and the number of friends and followers of the user (T_{fr} and T_{fo}, respectively). These friend- and follower numbers are also stored for each user (U^*) in the initially retrieved friends/followers data set (F), together with a timestamp (t) of the last time these numbers have been retrieved using the Twitter API.
2. The friend/follower numbers in tweet T created by user U can differ from the friend/follower numbers stored for user U^*. The general idea of this algorithm is to make a new call to the Twitter API to retrieve the up-to-date friend- and follower sets for U^* if they increased or decreased with a certain factor (f). This factor can be defined on user level, based on an average rate (δ) with which the friend/follower numbers change for user U^* given a predefined timeframe. This rate can be determined over a period of time and kept up-to-date after the initialization with the friend/follower numbers in each new tweet created by this user. User's whose δ is relatively high should be updated with new friend/follower numbers more often than user's with a lower value for δ.

The number of Twitter API calls that can be made to retrieve an user's friends and followers are rate limited, both on time and on size (see the start of section 7.3, which discusses the Twitter API endpoints used for these retrievals). This rate limit is set to 15 calls per 15 minutes (e.g. one call per minute) per API token per endpoint, with a maximum size of 5.000 friends or followers per call. For example, when only one API token is available, the maximum amount of friends or followers that can be retrieved for one user is 5.000. This rate limit means that friend/follower numbers cannot be retrieved continuously. Instead, API calls should be made in such a way that the rate limit set by Twitter is never broken. The following example shows how this can be done.

Consider the situation that is described in the list below.

- Total number of users: 1.000
- Available API tokens: 20
- Average for δ is 1.0

Given this data, the friend/follower numbers can be updated for 20 users per minute, meaning that each user U^* can be updated every $1000/20 = 50$ minutes. Users U^* with a high value for δ (higher than the average) should receive an update more often than users U^* with a low value for δ (lower than the average). For example, consider a user A ($\delta = 0.7$) and a user B ($\delta = 1.3$), where $\delta = 1.0$ is the average. The values for δ for each user determine how often this user should receive an update, relative to the average user update rate. In this case, user A would receive an update every $50/0.7 \approx 71$ minutes, and user B would receive an update every $50/1.3 \approx 38$ minutes. Such an update can be triggered when a tweet is retrieved. When the update interval for the user U that posted this tweet has expired, then the friend/followers number is updated. In theory, it is possible that there are no more API calls left because the number of users that need an update at a certain moment in time is too high. In this case, a first-in-first-out queue should be set up that contains the users to update as soon as a new API call is available.

Note that this approach assumes that only one API call is needed to retrieve all friends/followers of a user. However, there exist users who have more than 5.000 followers or friends (or both), meaning that one API call does not retrieve the full set. If you want to retrieve all followers and friends of a user when their update interval has expired, then a queue is also necessary because one user could require multiple API calls and thereby potentially more calls than the available calls per timeframe.

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 [62]. 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 [63], Infomap [64], Oslom [65], Link Clustering Algorithm (LCA) [66] and BigClam [67].

The SCD has been used in [68], 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[21] and LDBC DATAGEN[22]. 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.

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 29

TPR Given a set of vertices S and a vertex x, the triangle participation ration (TPR) is defined by the number of triangles in S that x is contained in. A triangle is defined by three distinct vertices x, y and z, where x is connected to y, y is connected to z and z is connected to x (see Figure 29).

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

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 [68]. 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.

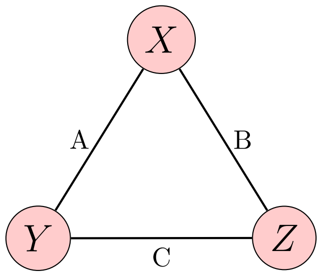


Figure 29: Example of a graph triangle

The implementation of SCD can be found on their GitHub repository[23], 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 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.

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

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).

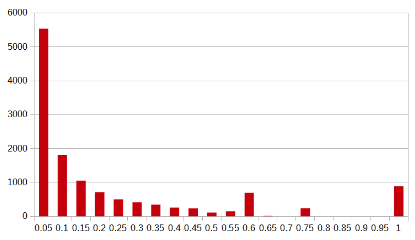
7.4.1 Friends/followers graph statistics

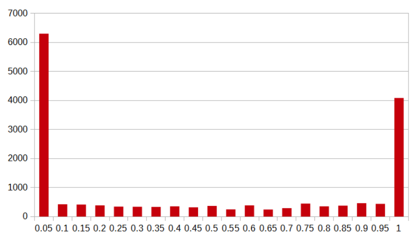
First, we will analyze the results from executing SCD's algorithm on OBI4wan's friends/followers graph, as shown in the histograms of Figure 30.

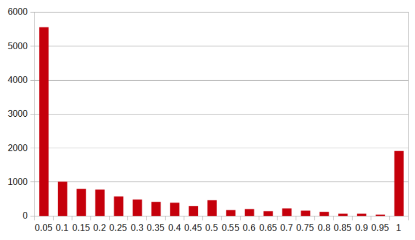
Clustering coefficient The clustering coefficient in Figure 30a shows that the majority of the graph's cluster has a relatively low clustering coefficient, meaning that they contain few closed triangles. There is a small peak at the extreme right of the histogram, showing those communities with a clustering 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.

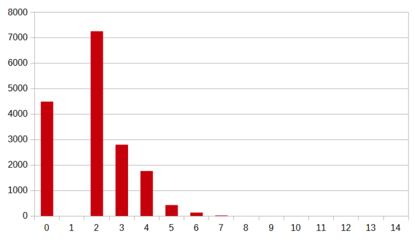
TPR The triangle participation ratio (TPR) in Figure 30b shows two peaks at the extremes, and the rest of the communities is divided over the intermediate ranges. This shows that many communities either have zero (or very few) vertices that close triangles, or (almost) all vertices close triangles.

Bridge ratio Figure 30c 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.


(a) Clustering coefficient


(b) TPR


(c) Bridge


(d) Diameter

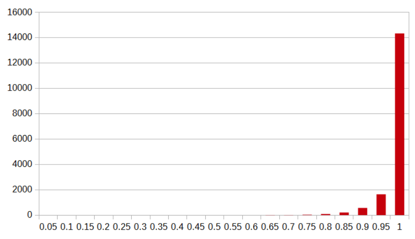

(e) Conductance

Figure 30: Distribution of the statistical indicators for the OBI4wan friends/followers graph (1 million users).

Diameter The diameter value in Figure 30d 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 30e 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

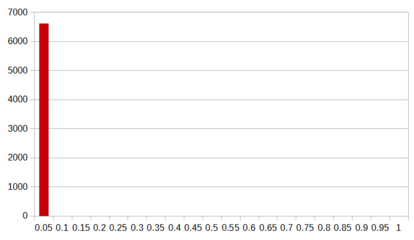
Next, we will analyze the results from executing SCD's algorithm on OBI4wan's "mentioned" graph, as shown in the histograms of Figure 31.

Clustering coefficient The clustering coefficient in Figure 31a 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.

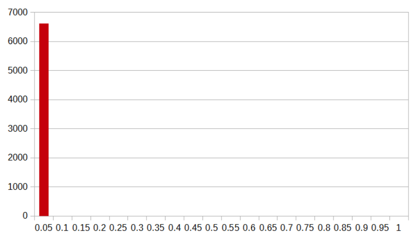
TPR Again, because there are no triangles in each of the communities (see the details about the clustering coefficient above), the TPR for all communities is equal to zero. This is shown in Figure 31b.

Bridge ratio Figure 31c 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).

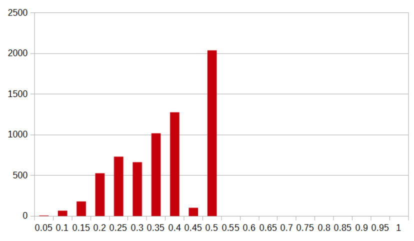
Diameter Figure 31d shows the diameter of all communities is equal to 2. In other words, it never takes more than two steps to travel from one person in a community to another.



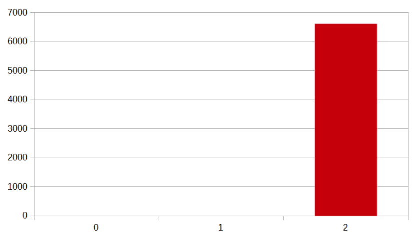
(a) Clustering coefficient



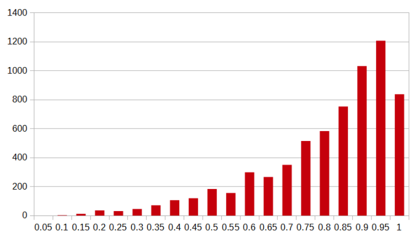
(b) TPR



(c) Bridge

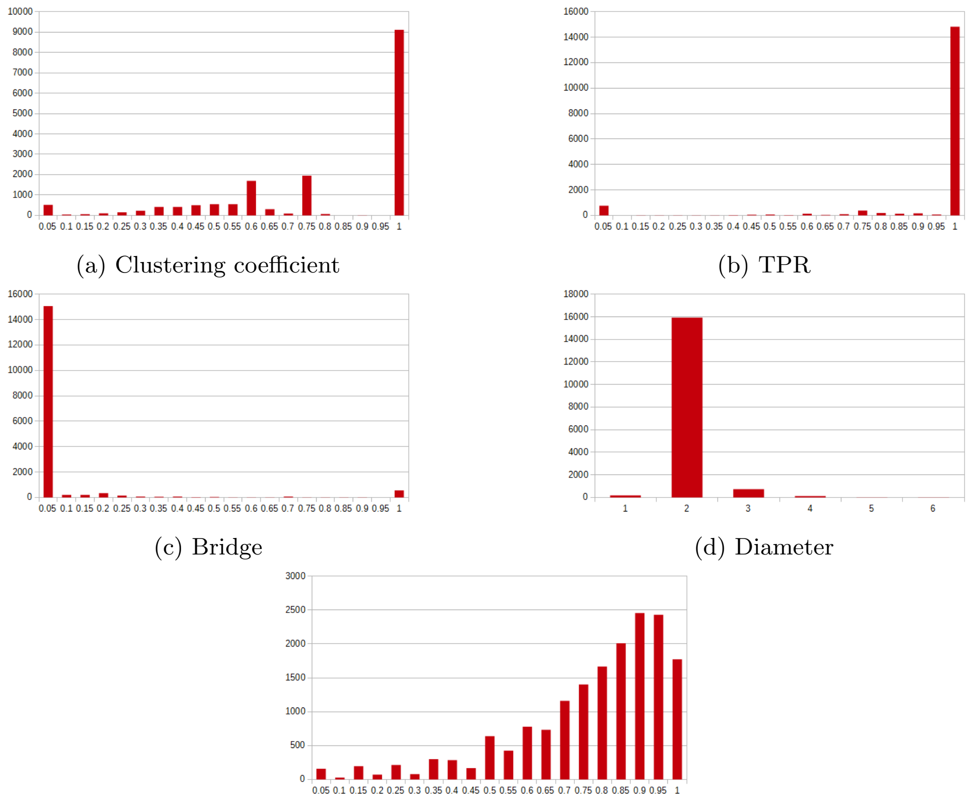


(d) Diameter



(e) Conductance

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



(a) Clustering coefficient

(b) TPR

(c) Bridge

(d) Diameter

(e) Conductance

Figure 32: Distribution of the statistical indicators for the OBI4wan retweeted graph (all tweets).

Conductance Figure 31e 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 32.

Clustering coefficient The clustering coefficient in Figure 32a 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.

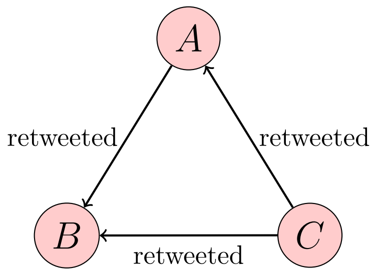


Figure 33: An example community of three persons, where person A has retweeted a tweet from person B, and person C has retweeted person B's retweet, thereby also indirectly retweeting person A's tweet.

TPR Figure 32b 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.

Bridge ratio Figure 32c 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 32d 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 32e 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 OBI4wan data versus LDBC data

The previous sections have shown statistics about the OBI4wan data set, which is a subset of the real Twitter network. The LDBC data set also claims to be a good representation of a real social network, although the type of network that is created by the LDBC DATAGEN is more like a Facebook network than a Twitter network. Nevertheless, it is worth to compare the statistical indicators of the OBI4wan data set to those of the LDBC data set. Section 7.5.1 shows how the LDBC data set from which the structural indicator values can be obtained has been created, which is based on the configuration used in [68]. Section 7.5.2 compares the structural indicator values of the OBI4wan friends/followers graph with the LDBC friends graph, where friends are defined as two people knowing each other.

7.5.1 Creating structural indicators for LDBC

Information about the friends and followers (e.g. two persons knowing each other) is directly contained in the LDBC data set, in the form of the *person_knows_person* relation file. This file contains - among other data - two person IDs on each line, representing the *knows* relation (edge) between two persons. Table 5 shows the average of the five structural indicators discussed in previous sections.

	friends/followers	mentions	retweets	LDBC friends
CC	0.2018	0	0.7807	0.2781
TPR	0.4374	0	0.9320	0.6446
Bridge	0.2867	0.3629	0.0516	0.0209
Diameter	1.9641	2	2.0406	2.0311
Conductance	0.9750	0.7830	0.7513	0.8873

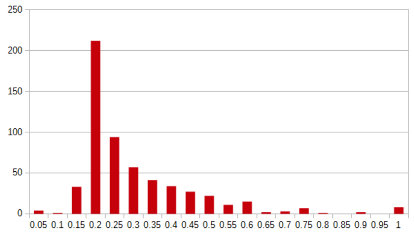
Table 5: Summary of the structural indicators for the OBI4wan friends/followers graph, the OBI4wan mentions graph, the OBI4wan retweets graph and the LDBC friends graph.

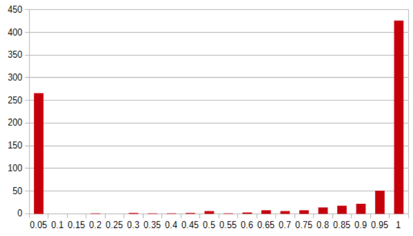
The first three columns are average values for the friends/followers-, mentions- and retweets graph extracted from the OBI4wan data set, respectively. The final column contains average values for the *knows* relation between to persons extracted from the LDBC data set. The LDBC data set used to obtain these statistical indicator values has been created by setting only the parameter for the number of persons to a custom value of 150K. The rest of the parameters use their default values. This recreates the experiments from [68].

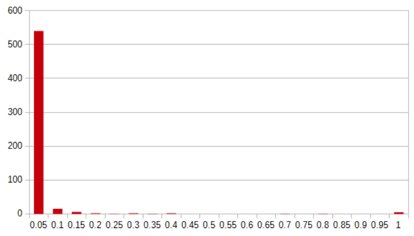
Side note To calculate the average numbers for the five structural indicators as shown in Table 5, we have used the SCD algorithm in combination with its CommunityAnalyzer[24]. The SCD algorithm creates communities based on a graph input file, where this input files contains one edge per line. In this case, each line contains a *knows* relation between two persons, defined by the IDs of the two persons involved in the relation. The CommunityAnalyzer then takes these created communities and the original graph input file, and outputs structural indicator values per community, of which five are shown in Table 5. A problem of the SCD algorithm is that is cannot handle numbers larger than *uint32_t*, but some person IDs from the LDBC data set are larger than that. Therefore, all person IDs have been translated into the *uint32_t* format, using the script shown below.

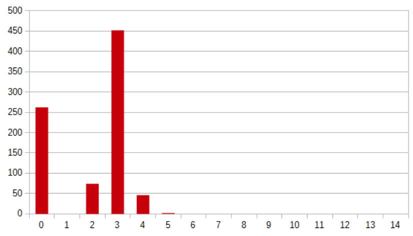
```
filename = sys.argv[1]
filename = sys.argv[1]
with open(filename) as f:
g = open("output.dat", "wb")
```

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


(a) Clustering coefficient


(b) TPR


(c) Bridge


(d) Diameter

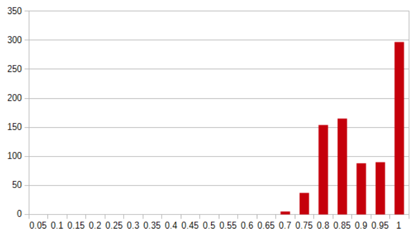

(e) Conductance

Figure 34: Distribution of the statistical indicators for the LDBC friends graph (based on the *person knows person* relationship).

```
for line in f:
    parts = line.split("|")
       number1 = ctypes.c_uint32(int(parts[0])).value
       number2 = ctypes.c_uint32(int(parts[1])).value
       edge = "{0} {1}\n".format(number1, number2)
       g.write(edge)
```

Using the resulting output file, the structural indicator values can be calculated per community. The histograms for these values are shown in Figure 34, their average values are shown in the last column of Table 5.

7.5.2 Comparing OBI4wan data and LDBC data

The following five paragraphs compare the histograms of the OBI4wan friends/followers graph (see Figure 30) to the histograms of the LDBC friends graph (see Figure 34). Also, the differences between the average values in Table 5 are discussed. Based on the findings, we can say something about how realistic (part of) the LDBC data set is compared to a real Twitter friends/followers graph like the one obtained from the OBI4wan data set.

Clustering coefficient See Figure 30a for the OBI4wan clustering coefficients and Figure 34a for the LDBC clustering coefficients. For both histograms, the majority of the communities have a clustering coefficient close to zero. The difference is that the peak for OBI4wan lies between 0 and 0.05, while the peak for LDBC lies between 0.15 and 0.20. This is also shown by the average value, which is 0.2018 for OBI4wan and 0.2781 for LDBC. So, the histogram shapes are similar, but the LDBC histogram is shifted to the right.

TPR See Figure 30b for the OBI4wan TPR values and Figure 34b for the LDBC TPR values. For both histograms, the majority of the communities have a TPR between 0 and 0.05, or between 0.95 and 1. The rest of the communities have TPR values divided equally over the rest of the values between 0 and 1. The highest peak for OBI4wan lies between 0 and 0.05, while the highest peak for LDBC lies between 0.95 and 1. This difference in peaks is represented by the average value for TPR, which is higher for LDBC (0.4374 in the case of OBI4wan and 0.6446 in the case of LDBC). So, the histogram shapes are similar, but the peaks on both ends of the histogram are switched between OBI4wan and LDBC. This means that relatively many LDBC communities contain vertices which are contained in one (or more) triangles, compared to OBI4wan communities. In other words: communities in LDBC are more tightly knit together in LDBC. This translated back to the difference between communities in a Facebook network (to which the LDBC data set can be compared) and communities in a Twitter network: Facebook communities tend to be more closely connected than Twitter networks because a relationship between two persons on Facebook is more like a real friendship than on Twitter (e.g. people can follow each other on Twitter without knowing each other in real life, because they like the content of each other's tweets).

Bridge See Figure 30c for the OBI4wan bridge values and Figure 34c for the LDBC bridge values. Both histograms have a peak between 0 and 0.05. The OBI4wan histogram also has a peak between 0.95 and 1, while the LDBC histogram does not have this peak. The missing of this peak on the right for LDBC might be caused by the translation of person IDs from one data format to another (see the side note in section 7.5.1, because the histograms for LDBC in [68] actually does contain this peak. The missing peak does also have a consequence for the average bridge value, which is higher for OBI4wan than for LDBC (0.2867 and 0.0209, respectively). The average bridge value in [68] will probably be closer to the OBI4wan value because of the right peak in the histogram.

Diameter See Figure 30d for the OBI4wan diameter values and Figure 34d for the LDBC bridge values. For both histograms, the majority of the communities have a relatively low diameter. The highest peak for OBI4wan is a diameter of 2, while the highest peak for LDBC is a diameter of 3. The second largest peak for both histograms is a diameter of 0, and both OBI4wan and LDBC do not have communities with a diameter of 1. The difference between OBI4wan and LDBC is that OBI4wan contains a few communities with a relatively high diameter, but the total number of communities with a high diameter is negligible (five with a diameter of 8, two with a diameter of 9 and 2 with a diameter of 14).

Conductance See Figure 30e for the OBI4wan conductance values and Figure 34e for the LDBC conductance values. For both histograms, the majority of the communities have conductance close to 1. Almost all OBI4wan communities have a conductance between 0.9 and 1, while LDBC also contains two small peaks between 0.75 and 0.85. This is shown in the average conductance value, which is higher for OBI4wan (0.9750) than for LDBC (0.8873). A higher conductance means that communities are not very well isolated and based on the two histograms and average values this is more the case for OBI4wan than for LDBC. Again, this can be related back to the type of social network of both data sets. The OBI4wan data set is a Twitter data set, which generally contains a less amount of isolated communities than a Facebook data set. Facebook communities are more representative of real world communities, in which friends tend to knit together into a relatively isolated community. Users on Twitter are not necessarily only connected to their real friends, but also to other persons they find interesting. This makes communities less isolated from each other.

7.5.3 Conclusion: OBI4wan friends/followers versus LDBC friends

In conclusion, the five histograms for the OBI4wan friends/followers graph and the LDBC friends graph are rather similar, except for the triangle participation ratio (TPR) and conductance. Both of these differences can be related back to the type of social network, where the OBI4wan data set is real Twitter data and the LDBC data set is more a Facebook-like network. Future work could focus on closing the gap between a Twitter data set and the LDBC data set, primarily by working on the differences between the two for the TPR- and conductance values. For both structural indicators, it is important that during the creation of the LDBC data set more inter-community communication is added, by introducing more relationships between persons of different communities.

7.6 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **Data set**: given the structure of the OBI4wan data set or the LDBC data set, what is the fastest, most efficient, best way to transform the data set to a graph-based format that can be used by graph databases (Titan)?
- **Architecture**: what is the best way to store Twitter data in a MonetDB data store?

8 Benchmarks setup and execution

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[25]. Section 8.3 shows further benchmark setups, for example the values of various properties that can be specified when running an LDBC benchmark. Section 8.4 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 Interactive Workload of the LDBC Social Network Benchmark (LDBC SNB IW) 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, 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 SNB IW 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 SNB IW substitution parameters during a benchmark run), building up queries for the respective database system (with the Gremlin language for Titan, and SQL for MonetDB), 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[26], which can be found in LDBC's SNB implementation GitHub repository[27]. In addition to the LDBC SNB IW, a few queries from the LDBC SNB Business Intelligence workload (LDBC SNB BI) have been taken from the GitHub repository[28]. Where the IW queries all start from a predefined query, the BI queries consider the whole graph in their execution. This leads to a graph-centric analysis (for BI), in comparison to the vertex-centric analysis (for IW). The next subsections will discuss the LDBC SNB IW query handlers and the LDBC SNB BI query handlers for both Titan and MonetDB. The textual definitions of all queries can be found in Appendix A.

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

[26]https://github.com/openlink/virtuoso-opensource

[27]https://github.com/ldbc/ldbc_snb_implementations/tree/master/interactive

[28]https://github.com/ldbc/ldbc_snb_implementations/tree/master/bi/virtuoso/queries/sql

8.1.1 LDBC SNB IW 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_NAME). The implementation of the HAS_NAME function in Java could look as below.

```
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 Java 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 SNB IW driver for MonetDB

Often used constructs

Substitution parameters The LDBC benchmark provides substitution parameters for each query. For MonetDB 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 MonetDB 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 *mepe*1 and *mepe*2:

```
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 MonetDB) has been based on the paper "Using the MonetDB database system as a platform for graph processing" [61]. This paper has proposed a solution for shortest path graph traversal in MonetDB 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 MonetDB, 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 MonetDB 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.

8.1.3 LDBC SNB BI driver

The LDBC SNB benchmark does not only contain vertex-centric queries - as in the Interactive Workload described before - but also graph-centric queries. These queries are combined in the Business Intelligence (BI) workload. At the time of writing, there exists only a Virtuoso SQL implementation of the BI queries - there is no detailed explanation of what each query is designed to do. Based on this existing SQL implementation, the list below shows the textual description for each of the BI queries that have been used in this research, from the perspective of MonetDB (e.g. as SQL queries). Because the data set from the LDBC DATAGEN has been stripped down, only the BI queries that contain the entities that remain are taken out from the complete set of 24 BI queries.

Query 3 This query measures the frequency of tag use in two consecutive months, and calculates the difference between these tags. The 100 tags that differ the most in frequency are taken as the result for this query. This query is executed by fetching frequency data for both months in two sub-queries, where in each sub-query the frequency of each unique tag is calculated. The counts per tag in the two sub-queries are then distracted from each other to retrieve the difference.

Query 6 This query retrieves all messages (either posts or comments) created by persons which contain a given tag. It then calculates the total number of likes that each person received on those messages, and calculates a score based on the number of posts, the number of comments and the number of likes. The result is given by the top 100 persons, ordered by this score. This query is executed by first retrieving all messages from persons which contain the given tag and then summing up these intermediate results, grouped by persons.

Query 7 This query retrieves the 100 persons that received the most likes on messages that contain a given tag. This query is executed by first retrieving all messages that contain the given tag (grouped by person), and then retrieves and counts all likes that were received on these messages. Finally, the messages are joined with the persons who created them in order to deliver the final result of the most liked persons for a given tag.

Query 8 This query retrieves the top 100 most used tags that are used in comments that reply to a post containing a given tag. This query is executed by retrieving all comments that were created in reply to a post with the given tag, where the comment does not contain the given tag. Then, all other tags that were used in the comments are retrieved, grouped and counted for the final result.

Query 12 This query retrieves the 100 most liked persons, defined by those persons who received the most likes on their created messages, given that these messages were created after a given date. Only the messages that received more than 100 likes are considered. This query is executed by retrieving all messages that received more than 100 likes and were created after the given date, and then retrieving the persons who posted these messages.

Query 18 This query retrieves the number of messages created by each person after a given date, and shows per number of messages how many persons have created that many messages. This results in a histogram containing the number of persons that have created a certain number of messages. This query is executed by retrieving the number of messages created by each person after the given date, and then grouping the results on the count of persons that have created a certain number of messages.

Query 20 This query retrieves the frequency of each tag in messages. This query is executed by retrieving all tags from created messages, and grouping them per unique tag.

8.2 Validating the LDBC SNB IW 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[29]. 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

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

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 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 MonetDB 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 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 MonetDB. Some of the configuration parameters have the same values for Titan and MonetDB, others are slightly different.

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

8.3.1 Setting up the general benchmark configuration

Both Titan and MonetDB 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[30].

database **value**: com.ldbc.driver.workloads.titan/MonetDB.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 MonetDB 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.

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

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.

8.3.2 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.

8.3.3 MonetDB-specific configuration parameters

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

8.4 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*[31]. 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 6.

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 6: 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.

8.4.1 Running the Titan benchmark

The Titan graph database is located on one or more *bricks* machines from the CWI SciLens cluster. Benchmarks have been executed for Titan with one Cassandra backend, four Cassandra backends and eight Cassandra backends. For this research, version 0.5.4 of Titan (with Hadoop version 2) has been used.

Titan problems Up until a scale factor of 10, benchmarks have been executed on Titan without any problems other than the increasing amount of time the benchmarks took for each higher scale factor. However, loading data for the benchmark on scale factor 10 - and especially benchmarks on scale factor 30 or higher seems to be problematic for Titan. The problems arise at the moment Hadoop writes edges to the graph, which is the final part of the Hadoop process

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

as detailed in section 6.4.2. While most of the reduce jobs succeed in writing their edges to the Titan graph, there are some reduce jobs which seem to have difficulty doing this. Based on the type of data that is loaded into the graph (a social network), the reasoning about why some of the reduce jobs are slow could refer back to the fact that social networks are hard to evenly distribute over multiple locations. Each social network contains some vertices which have a relatively high in- and/or out-degree compared to other vertices. Reducers that have the task of loading exactly the edges for those vertices into the graph have to do relatively more work than other reducers, which could be the reason for being relatively slow.

One solution could be to distribute the edges more evenly over the available reducers, so that each reducer receives more or less the same amount of work. In theory, this would then eliminate the slow reducers. Titan dedicated a special chapter of its documentation on partitioning the input data in such a way that the data is more evenly distributed, using cluster partitioning [45]. When the *cluster.partition* configuration parameter is set to true, elements of the graph are randomly distributed across the cluster. The *cluster.max-partitions* configuration parameters can then be set to represent the number of virtual partition blocks that will be created. According to the documentation, this number should be roughly twice the number of backend instances used to store the graph. Finally, the *ids.flush* configuration parameter should be set to false for graph partitioning to work correctly. Setting this parameter to false delays the assigning of IDs to vertices and edges to the moment when the transaction commits, instead of assigning immediately upon creation. This prevents the assignment of all edges which are connected to the same vertex to the same reducer job based on the vertex ID, introducing the high load on one reducer as discussed earlier.

While this solution could work in theory, practice shows that it does not. The same problem arises, and there are still reducers who seem to receive more work than others. Both with and without the graph partitioning detailed above, some of the slow reducers finish with an I/O exception, not able to committing the transaction due to an exception during persistence of the graph data:

```
Error: java.io.IOException:
Could not commit transaction due to exception during persistence
```

In the end, this error can be related back to an OutOfMemoryError for the java heap space in Cassandra:

```
Caused by: java.lang.OutOfMemoryError: Java heap space
```

In an attempt to overcome this problem, the heap size of the Cassandra backend instances has been increased manually. First to the maximum heap size as posed by Cassandra, which is 8GB. When the heap space is set higher, this could give problems with the Java garbage collection and the Operating System's page cache [46]. However, when the 8GB heap size did not solve the problem, the java heap size has been set to a value of 64GB, which is 1/4th of the total amount of memory available on the used machines. Unfortunately, this too did not solve the problem of slow reducers.

8.4.2 Running the MonetDB 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.

MonetDB problems Up until a scale factor of 10, benchmarks have been executed on MonetDB without any problems other than the increasing amount of time the benchmarks took for each higher scale factor. However, executing benchmarks on MonetDB of scale factor 30 or higher introduced problems regarding the total execution time. According to the specifications of the LDBC benchmark [73], benchmarks have to cover at least two hours of benchmark data (e.g. based on the timestamps of the messages in the benchmark, there should be at least two hours between the first and last message). This matches with an operation count (e.g. the total number of operations that will be executed during the benchmark), for example an operation count of $\tilde{2}75.000$ for scale factor 30. However, when executing the SF30 benchmark on MonetDB, the query execution time for some queries rose up to a level where it would take hours to complete, or not complete at all due to a timeout. The shortest path queries (query 13 and 14) are two examples of such queries. Even when lowering the operation count to a level where all queries are executed at least once (too low operation counts skip some queries due to their interval settings), the query execution times were still way above a reasonable execution time.

Efforts have been taken in order to try and reduce the execution times for MonetDB. Initially, the tables created in MonetDB did not have an explicitly defined primary key, which caused MonetDB to sometimes create query plans which were not optimal. For example, the best query plan for queries starting at a single vertex is to narrow down the total amount of data to the scope of this single vertex, instead of to some other column. Setting primary keys helps MonetDB to create this optimal query plan. Furthermore, initially some of the queries did some unnecessary work regarding the joining of tables. For example, when tables have already been joined in a sub-query, there is no need for another join between the same tables in the final part of the query. These unnecessary joins have been eliminated in the course of this project. However, these efforts have not led to (significantly) faster execution times.

8.5 Introducing Virtuoso

Sections 8.4.1 and 8.4.2 shown that both Titan and MonetDB have some problems regarding loading data (Titan) and running benchmarks on this data (MonetDB), especially as the amount of data grows larger. Therefore, while both systems could in theory be used on relatively small amounts of data, they are not (yet) suitable for use with larger amounts of data. To find a solution for this, we have taken a quick look at another database system which has also executed the LDBC data set and -benchmarks. This system is Virtuoso, which has been described earlier in section 3.5.2.

Virtuoso's implementation of the LDBC data set and -benchmark can be found on GitHub [47]. This page shows how Virtuoso can be installed on a Linux environment, and contains scripts that can be used to load LDBC data sets into a Virtuoso database. In addition to Titan and MonetDB, the LDBC benchmarks have also been executed on the same scale factors as used for Titan and MonetDB (scale factors 1, 3 and 10). Without going into further details of Virtuoso and how this database system works internally (which is outside the scope of this report), the Virtuoso benchmarks add another platform to the comparison of (graph) database systems.

8.6 Analyzing benchmark results

The next subsections contain the results of the LDBC benchmark runs for both Titan and MonetDB. First, a general analysis is given per query type, regardless of the scale factor. Then a specific analysis is provided per scale factor for remarks that do not fit in the general analysis. For each 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*). The JSON output that contains all these values is also added for reference. The graphs and JSON output can be found in appendix B.

8.6.1 LDBC SNB IW benchmark, general analysis

For each query type (complex, short or update), there are some general remarks about the obtained results for MonetDB and Titan, regardless of the scale factor. For example, there might be queries which are always performing slower/faster on one database system compared to another database system. This general analysis is given in the paragraphs below, where each paragraph discusses the queries of one query type.

Complex queries Four of the complex queries have significant different execution times between MonetDB and Titan: queries 6, 9, 13 and 14. On the first two queries, Titan performs significantly slower than MonetDB. On the second two queries, MonetDB performs slower than Titan, although the difference is less significant then for the first two queries.

Both query 6 and query 9 require to retrieve the friends and friends-of-friends of some start person. Titan's implementation of these queries requires two separate graphs traversals: the first retrieves all friends and friends-of-friends, and the second uses this subset as the starting point for the rest of the query. The set with friends-of-friends can become a relatively large subset of data, which in combination with the two separate graph traversals might be the cause of Titan's slow execution.

Query 13 and query 14 are both shortest path queries, trying to find the length of the shortest path (query 13) or all shortest paths and their weight, defined by some algorithm (query 14). For both queries, MonetDB performs less well than Titan, and the difference in execution time is getting bigger when the scale factor increases. This shows that "real" graph queries like a shortest path algorithm do not perform very well on a relational/column database system like MonetDB, while a dedicated graph database like Titan - which should be optimized for this type of queries - performs relatively well.

Virtuoso is performing better on all complex queries than both Titan and MonetDB. Titan is only faster than Virtuoso in executing query 7 on scale factor 1, and is on par with Virtuoso on the same query on scale factor 3.

Short queries On all scale factors, the short queries perform significantly better on Titan than on MonetDB, up to multiple orders of magnitude (four orders of magnitude for short query 1 on scale factor 1, and even five orders of magnitude for short query 1 on scale factor 3). What is also important to note, is the fact that there is a large gap between the minimum and maximum execution times on MonetDB. The minimum execution time is comparable to the mean execution time on Titan, while the maximum execution time is up to three orders of magnitude higher than the minimum execution time. The fact that multiple threads (24 in total) are executing queries at the same time might an explanation for this large gap: the maximum execution times might be measured at periods of high load, causing queries to have a higher-than-average execution time.

Virtuoso is faster than both Titan and MonetDB on short queries 2 and 3, and on par with Titan on the other short queries. When there are differences between Titan and Virtuoso on those other short queries, then the differences are not significant.

Update queries Like with the short queries, Titan performs faster on all update queries than MonetDB, although the differences are not always as big as with the short queries. The most expensive update queries for MonetDB are queries 1 and 7, which have the most significant differences compared to Titan. Both of these queries require the insertion of one or more new edges, besides the insertion of a new vertex.

Query 1 inserts a new person vertex into the database, and consequently also all edges from this new person to other persons (e.g. *knows* relations between persons). There could be more than one person that the new person knows, possibly requiring multiple edge insertions.

Query 7 inserts a new comment vertex into the database, and consequently also the edge from this new comment to the person who created the comment (e.g. the *hascreator* relation), and the edge to the post to which the comment is a reply (e.g. the *replyof* relation). In other words, both query 1 and query 7 require inserts into multiple tables (at least two for query 1, and two for query 7), which causes relatively much trouble for MonetDB. Also query 6, which inserts a new post, requires the insertion into multiple tables: the table holding all posts and the table holding the *hascreator* relationship. However - while query 6 is slower on MonetDB than on Titan - the difference in execution time between MonetDB and Titan is not as significant as for queries 1 and 7.

Virtuoso is on par with Titan on update queries 2, 3, 6 and 7, and slower than Titan on update queries 1 and 8. Virtuoso is faster in all update queries than MonetDB.

Titan Cassandra backends Titan claims to be a scalable graph database system, meaning that it supports multiple backend nodes where graph data can be stored. In the case of this research, these backends are Cassandra instances. All executed benchmarks have been executed on Titan using one Cassandra backend, four Cassandra backends and eight Cassandra backends. The results of these three benchmarks per scale factor are bundled into graphs for the complex-, short- and update queries (see Appendix B.1. In theory, spreading the data over multiple backends means that every backend has to do less work regarding the retrieval of a result for a query. However, the downside of multiple backends is that they need to communicate with each other and combine data from multiple locations in order to obtain the final result for a query. Multiple backends are then only faster than a single backend when the benefit of doing less work weights bigger than the downside of communication and data combining. The graphs in Appendix B.1 show that this is not the case: the more Cassandra backends that are used, the longer it takes for queries to complete. In other words, queries lose more time in communication and data combining than they win in sharing the work over multiple locations. The main reason for this is that the data in the graph is social network data, which is very hard to distribute over multiple locations without introducing edge cuts[32], thereby introducing many communication between the backends.

8.6.2 LDBC SNB IW benchmark, SF1 specific analysis

The results of the LDBC SNB Interactive Workload (IW) benchmark on scale factor 1 (SF1) can be found in the following three figures in appendix B:

- **Complex queries**: see Figure 35
- **Short queries**: see Figure 36
- **Update queries**: see Figure 37

8.6.3 LDBC SNB IW benchmark, SF3 specific analysis

The results of the LDBC SNB Interactive Workload (IW) benchmark on scale factor 3 (SF3) can be found in the following three figures in appendix B:

[32]A cut edge is an edge whose endpoints are location on different machines.

- **Complex queries**: see Figure 41
- **Short queries**: see Figure 42
- **Update queries**: see Figure 43

The results of scale factor 1 show that complex query 14 is around 1.5 times slower on MonetDB than on Titan. For scale factor 3, the same query performs more than 27 times slower on MonetDB than on Titan, which is a significant difference. The fact that MonetDB is not optimized for real graph queries like finding all shortest paths and their weights is really beginning to show when the scale factor increases: an increase in scale factor of 3 causes an increase in execution time of 18.

8.6.4 LDBC SNB IW benchmark, SF10 specific analysis

The results of the LDBC SNB Interactive Workload (IW) benchmark on scale factor 10 (SF10) can be found in the following three figures in appendix B:

- **Complex queries**: see Figure 47
- **Short queries**: see Figure 48
- **Update queries**: see Figure 49

8.6.5 Conclusions about LDBC SNB IW

While some of the complex queries are performing better on MonetDB than on Titan, the majority of the queries (including short queries and update queries) are performing significantly better on Titan. The shortest path traversals (complex queries 13 and 14) are becoming significantly slower on MonetDB compared to Titan with each increase of scale factors. The same is true for the short- and updates queries. In a real-world scenario most of the queries executed on a database system that holds a social network are short- and update queries, while the complex queries are executed much less. Therefore, when choosing between MonetDB and Titan as the database system in an interactive workload scenario, Titan is the best choice.

8.6.6 LDBC SNB BI benchmark on MonetDB on SF1, SF3 and SF10

The results of the LDBC SNB BI benchmark on scale factor 1, 3 and 10 (SF1, SF3 and SF10) for MonetDB can be found in Figure 53 in appendix B.

8.7 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **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)?

  - Because Titan and MonetDB cannot use the same query language, one can never exclude all advantages/disadvantages of one database system compared to another. For example, the graph query language Gremlin (used by Titan) is more suitable for graph-like queries, while SQL (used by MonetDB) is more suitable for transactional queries.

- **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?
- **Driver**: how can the drivers for both database solutions be validated?
- **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?
- **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?

9 Discussion & Conclusion

he Dutch company OBI4wan delivers a complete solution for social media monitoring, webcare and social analytics. In order to do this, they are collecting data from various social networks, such as Twitter and Facebook. Using various types of ElasticSearch- and SQL-queries, OBI4wan can provide insights in this large amount of data. For example, organizations can keep track of how people talk about them on these social networks and whether this talk has a positive or negative sentiment. However, more complex queries are hard to answer with the querying capabilities of ElasticSearch and SQL. For example, one might 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). Using traditional relational databases, queries like these have proven to be difficult to execute because of the large amount of required joins. On the other side, graph networks and graph databases are designed to answer these type of questions

This report compares the centralized solution of the more traditionally based database MonetDB to the distributed, graph-based database solution of Titan. Both database systems have been described in detail in this report. The decentralized graph database Titan provides a pluggable storage- and indexing backend, with native support for Cassandra, HBase, BerkeleyDB (storage backend), ElasticSearch, Solr and Lucene (indexing backend). Queries can be executed on Titan using the graph query language Gremlin. The centralized column store MonetDB used so-called Binary Association Tables (BAT) to store its data, which can be queried with either SQL or the more lower level MonetDB Assembly Language (MAL). In this report, SQL has been used as the query language.

In order to test both database systems, the dynamically generated dataset from the Linked Data Benchmark Council (LDBC) has been used. This data set represents a real social media network with the likes of Facebook. To make the data set look more like a Twitter data set - and thereby more representative of the real OBI4wan data set - some elements from the LDBC data set have been stripped out. Using transformation scripts, the data set has been transformed into a format that can be read by either MonetDB (SQL COPY INTO statements) or Titan (adjacency lists loaded into a Titan instance using Hadoop).

Another section of the paper has focused on the real data set from OBI4wan, providing some analysis on this data set and comparing it to the fictional LDBC data set. While there are some differences between both data sets, these differences are not too significant and can be overcome to make the LDBC data set look more like a real Twitter social network.

Finally, the benchmark that has been executed on both MonetDB and Titan is the Social Network Benchmark (SNB), also from LDBC. This benchmark executes a variety of complex queries on both databases, ranging from retrieving friends-of-friends in a graph traversal to retrieving the latest likes on messages created by some person his friends. In addition, the SNB executes relatively simple short queries like retrieving a person's profile information or this person's last messages. These short queries have been added to the benchmark to make it more realistic: short queries like these are likely to occur more often than the complex queries. Finally, the SNB executes update queries like inserting a new person or a new message into the database. Unfortunately, the benchmark results for both MonetDB and Titan are not promising for large amounts of data. The bigger the data set becomes, the more problems this introduces for both database systems. Titan has difficulties loading big data sets into a graph, which either takes a very long time (up to multiple days) or does not succeed at all. MonetDB has difficulties with executing queries, especially the more graph-like queries such as finding the shortest path in a network. Therefore, the same conclusion can be drawn for both database systems: loading data into the database and executing benchmark queries does work for relatively small data sets, but more and more problems are introduced as the data set grows in size. So, while both systems may look promising in theory, they do not seem capable of handling a large social network, at least not at this point. The fact that a social network is complex and cannot be distributed over multiple machines easily is one of the reasons for the problems that arise with larger data sets.

Because both Titan and MonetDB have some problems, another database system has been introduced into the comparison: Virtuoso. Virtuoso has been working on executing the LDBC benchmarks in the past, and the accompanying implementation is freely available on GitHub [47]. Virtuoso has a plus over both Titan and MonetDB in that it does not present any problems regarding loading data and executing the benchmarks. Furthermore, it is faster than both Titan and MonetDB in its execution of the complex queries of the LDBC benchmark. For the short- and update queries, Virtuoso is always faster than MonetDB, and mostly on par with Titan. These results are introducing Virtuoso as another competitor in the comparison of (graph) databases, and at this point even the best possible choice regarding a (graph) database that is suitable for executing graph-like queries in real-time.

10 Future work

Both Titan and MonetDB have difficulties with the handling of relatively large social networks. Titan's problems can be related back to the nature of a social network: largely scattered and hard to distribute. MonetDB has difficulties with executing graph-like queries, which is explainable because MonetDB is not designed for this type of queries. In order to make MonetDB more suitable as a database that can store a graph network and answer graph queries, a future project (coordinated by the Centrum Wiskunde en Informatica [CWI] and OBI4wan) will extend MonetDB with these graph querying capabilities. This project will results in a dashboard in which graph queries can be executed, with the extended MonetDB as the backend.

Another untouched subject is the distribution of a social network. The connections in a social network are very complex, because there are not only connections inside communities within the social network, but also between persons in different communities. On forehand, one cannot easily tell what these communities will look like, and consequently how the social network can best be distributed over multiple locations. In theory, every social network could have its own best distribution scenario, but this scenario is not suitable for all social networks. A future project can try to find the best possible algorithm to distribute any social network.

10.1 Research questions

The following research questions have been answered in this section and/or will be answered in this report.

- **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?

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] http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSIntro

[45] http://s3.thinkaurelius.com/docs/titan/0.5.4/graph-partitioning.html

[46] https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_tune_jvm_c.html

[47] https://github.com/ldbc/ldbc_snb_implementations/tree/master/interactive/virtuoso

[48] 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

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

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

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

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

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

[54] 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.

[55] 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.

[56] 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.

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

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

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

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

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

[62] 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.

[63] 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.

[64] 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.

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

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

[67] 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.

[68] 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.

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

[70] 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.

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

[72] 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.

[73] Arnau Prat, Peter Boncz, Josep Lluís Larriba et. al. "LDBC Social Network Benchmark (SNB) - v0.2.2 First Public Draft Release v0.2.2". *LDBC Council* (2015)

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

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

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

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

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

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

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

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

A LDBC queries textual definitions

This appendix provides the official query descriptions for the LDBC Social Network Benchmark Interactive Workload, as provided in the first public draft of the Social Network Benchmark [73].

A.1 Complex query 1

Friends with a certain name Given a start Person, find up to 20 Persons with a given first name that the start Person is connected to (excluding start Person) by at most 3 steps via Knows relationships. Return Persons, including summaries of the Persons workplaces and places of study. Sort results ascending by their distance from the start Person, for Persons within the same distance sort ascending by their last name, and for Persons with same last name ascending by their identifier

A.2 Complex query 2

Recent posts and comments by your friends Given a start Person, find (most recent) Posts and Comments from all of that Person's friends, that were created before (and including) a given date. Return the top 20 Posts/Comments, and the Person that created each of them. Sort results descending by creation date, and then ascending by Post identifier.

A.3 Complex query 4

New topics Given a start Person, find Tags that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to friends' Posts created within a given time interval, and that were never attached to friends' Posts created before this interval. Return top 10 Tags, and the count of Posts, which were created within the given time interval, that this Tag was attached to. Sort results descending by Post count, and then ascending by Tag name.

A.4 Complex query 6

Tag co-occurrence Given a start Person and some Tag, find the other Tags that occur together with this Tag on Posts that were created by start Person's friends and friends of friends (excluding start Person). Return top 10 Tags, and the count of Posts that were created by these Persons, which contain both this Tag and the given Tag. Sort results descending by count, and then ascending by Tag name.

A.5 Complex query 7

Recent likes Given a start Person, find (most recent) Likes on any of start Person's Posts/Comments. Return top 20 Persons that Liked any of start Person's Posts/Comments, the Post/Comment they liked most recently, creation date of that Like, and the latency (in minutes) between creation of Post/Comment and Like. Additionally, return a flag indicating whether the liker is a friend of start Person. In the case that a Person Liked multiple Posts/Comments at the same time, return the Post/Comment with lowest identifier. Sort results descending by creation time of Like, then ascending by Person identifier of liker.

A.6 Complex query 8

Recent replies Given a start Person, find (most recent) Likes on any of start Person's Posts/Comments. Return top 20 Persons that Liked any of start Person's Posts/Comments, the Post/Comment they liked most recently, creation date of that Like, and the latency (in minutes) between creation of Post/Comment and Like. Additionally, return a flag indicating whether the liker is a friend of start Person. In the case that a Person Liked multiple Posts/Comments at the same time, return the Post/Comment with lowest identifier. Sort results descending by creation time of Like, then ascending by Person identifier of liker.

A.7 Complex query 9

Recent posts and comments by friends or friends of friends Given a start Person, find the (most recent) Posts/Comments created by that Person's friends or friends of friends (excluding start Person). Only consider the Posts/Comments created before a given date (excluding that date). Return the top 20 Posts/Comments, and the Person that created each of those Posts/-Comments. Sort results descending by creation date of Post/Comment, and then ascending by Post/Comment identifier.

A.8 Complex query 13

Single shortest path Given two Persons, find the shortest path between these two Persons in the subgraph induced by the Knows relationships. Return the length of this path.

A.9 Complex query 14

Weighted/unweighted paths Given two Persons, find all (unweighted) shortest paths between these two Persons, in the subgraph induced by the Knows relationship. Then, for each path calculate a weight. The nodes in the path are Persons, and the weight of a path is the sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated such that every reply (by one of the Persons) to a Post (by the other Person) contributes 1.0, and every reply (by ones of the Persons) to a Comment (by the other Person) contributes 0.5. Return all the paths with shortest length, and their weights. Sort results descending by path weight. The order of paths with the same weight is unspecified.

A.10 Short query 1

Person profile Given a start Person, retrieve their first name, last name, birthday, IP address, browser, and city of residence.

A.11 Short query 2

Person recent messages Given a start Person, retrieve the last 10 Messages (Posts or Comments) created by that user. For each message, return that message, the original post in its conversation, and the author of that post. If any of the Messages is a Post, then the original Post will be the same Message, i.e., that Message will appear twice in that result. Order results descending by message creation date, then descending by message identifier.

A.12 Short query 3

Person friends Given a start Person, retrieve all of their friends, and the date at which they became friends. Order results descending by friendship creation date, then ascending by friend identifier.

A.13 Short query 4

Message content Given a Message (Post or Comment), retrieve its content and creation date.

A.14 Short query 5

Message creator Given a Message (Post or Comment), retrieve its author.

A.15 Short query 7

Message replies Given a Message (Post or Comment), retrieve the (1-hop) Comments that reply to it. In addition, return a boolean flag indicating if the author of the reply knows the author of the original message. If author is same as original author, return false for "knows" flag. Order results descending by creation date, then ascending by author identifier.

A.16 Update query 1

Add Person Add a Person to the social network.

A.17 Update query 2

Add Post Like Add a Like to a Post of the social network.

A.18 Update query 3

Add Comment Like Add a Like to a Comment of the social network.

A.19 Update query 6

Add Post Add a Post to the social network.

A.20 Update query 7

Add Comment Add a Comment replying to a Post/Comment to the social network.

A.21 Update query 8

Add Friendship Add a friendship relation to the social network.

B LDBC benchmark results

This appendix contains the results of the LDBC Social Network Benchmarks on various data scale factors, visualized in JSON and 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 in graphs. Furthermore, the JSON result data from the LDBC data generator are shown.

B.1 LDBC Interactive Workload

B.1.1 Scale factor 1

See these Figures for the graphs created from the output JSON:

- Figure 35 (complex queries)
- Figure 36 (short queries)
- Figure 37 (update queries)
- Figure 38 (complex queries Titan)
- Figure 39 (short queries Titan)
- Figure 40 (update queries Titan)

The output JSON itself (for all query types) is shown these Tables:

- Table 7 (MonetDB)
- Table 8 (Virtuoso)
- Table 9 (Titan, 1 Cassandra backend)
- Table 10 (Titan, 4 Cassandra backends)
- Table 11 (Titan, 8 Cassandra backends)

B.1.2 Scale factor 3

See these Figures for the graphs created from the output JSON:

- Figure 41 (complex queries)
- Figure 42 (short queries)
- Figure 43 (update queries)
- Figure 44 (complex queries Titan)
- Figure 45 (short queries Titan)
- Figure 46 (update queries Titan)

The output JSON itself (for all query types) is shown these Tables:

- Table 12 (MonetDB)

- Table 13 (Virtuoso)
- Table 14 (Titan, 1 Cassandra backend)
- Table 15 (Titan, 4 Cassandra backends)
- Table 16 (Titan, 8 Cassandra backends)

B.1.3 Scale factor 10

See these Figures for the graphs created from the output JSON:

- Figure 47 (complex queries)
- Figure 48 (short queries)
- Figure 49 (update queries)
- Figure 50 (complex queries Titan)
- Figure 51 (short queries Titan)
- Figure 52 (update queries Titan)

The output JSON itself (for all query types) is shown these Tables:

- Table 17 (MonetDB)
- Table 18 (Virtuoso)
- Table 19 (Titan, 1 Cassandra backend)
- Table 20 (Titan, 4 Cassandra backends)
- Table 21 (Titan, 8 Cassandra backends)

B.2 LDBC Business Intelligence Workload

See Figure 53.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	112	5940.29464286	49	58454	1287	23316	45178	56258
complex13	153	23346.5098039	1936	74700	9550	61306	65100	69068
complex14	59	110655.525424	3295	1320512	67860	142800	225464	1300800
complex2	79	3806.73417722	355	48436	2021	3301	7173	46686
complex4	81	3408.67901235	134	55226	1744	3647	10559	45078
complex6	9	8158.66666667	1083	54762	2275	4483	54762	54762
complex7	60	3173.93333333	83	47838	1608	3126	3379	47636
complex8	324	4822.38888889	8	58998	1613	3953	43994	56280
complex9	8	2328.75	781	4104	2430	2817	4104	4104
short1	1116	4538.86290323	0	58888	1521	4197	41668	51118
short2	1116	4679.61917563	217	57832	1786	4344	41590	53136
short3	1116	5262.99731183	2	65276	1655	6381	45362	57848
short4	1104	4166.87228261	0	58984	1530	3668	40266	51388
short5	1104	4291.06521739	4	57514	1706	4045	31735	52684
short7	1104	4059.03442029	3	60094	1571	3798	31741	51640
update1	3	21857.6666667	3343	56978	5251	56978	56978	56978
update2	298	1028.67114094	1	50872	68	1919	3406	45920
update3	383	758.248041775	1	47964	69	328	3343	11136
update6	204	1184.46078431	2	59028	155	312	4914	48116
update7	203	4818.12807882	4	69052	255	9808	49706	65348
update8	34	2488.44117647	2	57822	125	300	6031	57822

Table 7: MonetDB statistics on scale factor 1.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	112	40.0803571429	7	912	8	105	182	312
complex13	154	26.3701298701	1	180	11	43	45	178
complex14	59	67.593220339	4	847	42	97	142	707
complex2	79	131.962025316	4	873	64	347	427	867
complex4	81	187.407407407	45	876	135	287	386	862
complex6	9	449.444444444	176	1052	383	655	1052	1052
complex7	61	121.524590164	4	909	48	267	365	760
complex8	326	151.791411043	14	879	74	347	499	837
complex9	7	456.428571429	232	935	399	661	935	935
short1	1090	2.06330275229	0	181	1	3	5	12
short2	1090	40.7064220183	1	217	41	66	108	193
short3	1090	7.57889908257	0	229	2	29	41	43
short4	1084	13.057195572	0	212	1	41	42	48
short5	1084	2.64760147601	0	49	1	3	6	41
short7	1084	14.1291512915	1	210	3	42	43	51
update1	2	399.0	7	791	7	791	791	791
update2	298	3.62416107383	1	108	2	4	7	70
update3	385	2.42857142857	0	120	2	3	4	20
update6	198	2.61111111111	1	53	1	4	7	13
update7	209	9.03827751196	1	834	3	10	18	47
update8	32	128.9375	1	643	17	433	528	643

Table 8: Virtuoso statistics on scale factor 1.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	112	878.375	119	6293	723	1155	1981	4795
complex13	153	106.568627451	1	2303	70	181	238	418
complex14	59	69852.6101695	16942	421872	34456	202640	348432	412048
complex2	79	13588.4683544	1	22934	13573	18733	20140	22688
complex4	81	4351.27160494	617	10583	4182	8637	9095	10315
complex6	9	496133.333333	396368	587072	504768	565664	587072	587072
complex7	60	91.2	1	519	29	227	254	421
complex8	324	4077.2962963	154	22635	3465	8781	12049	18298
complex9	8	472414.75	75516	688256	482960	578528	688256	688256
short1	1122	0.310160427807	0	80	0	1	1	2
short2	1122	542.751336898	0	9375	239	1302	2276	4820
short3	1122	26.0053475936	0	2705	7	43	130	285
short4	1110	1.13153153153	0	204	1	1	2	3
short5	1110	1.10720720721	0	164	1	2	2	5
short7	1110	5.07387387387	0	265	2	10	13	20
update1	3	3.66666666667	2	5	4	5	5	5
update2	298	3.04697986577	1	10	3	4	5	7
update3	383	3.08355091384	1	42	3	4	4	7
update6	204	2.99509803922	1	59	2	4	5	8
update7	203	5.27093596059	2	14	5	8	9	13
update8	34	3.02941176471	2	8	3	4	4	8

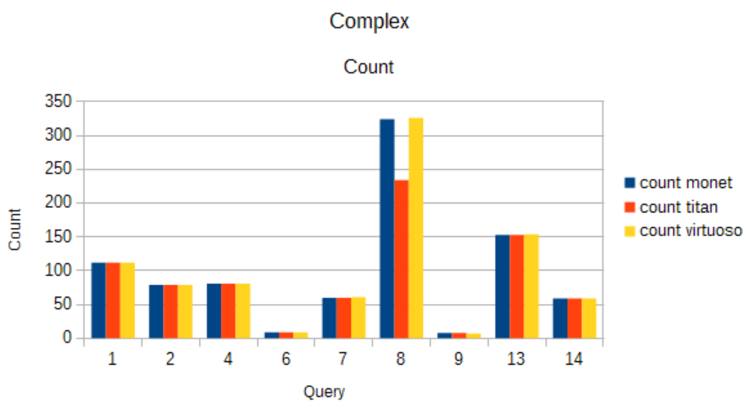
Table 9: Titan statistics on scale factor 1, 1 Cassandra backend.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	112	803.857142857	154	7309	550	1593	2347	3655
complex13	153	99.4509803922	1	1400	53	191	331	965
complex14	59	62776.5254237	12818	496192	23659	195440	326592	412624
complex2	79	14067.0253165	1	27996	13664	20505	23134	26059
complex4	81	4567.0	517	13719	3864	8583	10917	12789
complex6	9	644561.777778	483024	868832	613600	807232	868832	868832
complex7	60	109.666666667	0	1382	33	232	264	405
complex8	324	4299.71604938	134	20195	3720	8717	11915	16236
complex9	8	699256.25	89416	1600832	601920	849600	1600832	1600832
short1	1079	0.68860055607	0	222	0	1	1	2
short2	1079	587.976830399	1	14267	294	1389	2356	4131
short3	1079	26.3540315107	0	2442	8	43	110	292
short4	1082	0.963955637708	0	45	1	2	2	3
short5	1082	1.35489833641	0	199	1	2	2	3
short7	1082	4.72458410351	0	211	3	11	13	20
update1	3	37.6666666667	2	108	3	108	108	108
update2	298	3.96644295302	2	251	3	4	5	7
update3	383	3.37336814621	1	132	3	4	5	6
update6	204	3.59803921569	2	28	3	5	6	9
update7	203	6.3842364532	2	68	5	9	11	17
update8	34	4.55882352941	1	59	3	4	4	59

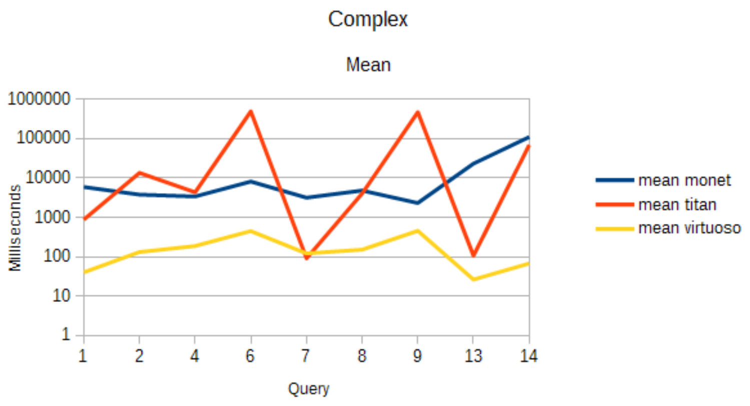
Table 10: Titan statistics on scale factor 1, 4 Cassandra backends.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	112	831.794642857	121	6145	583	1561	2618	4762
complex13	153	108.267973856	1	1118	64	205	450	958
complex14	59	76296.5423729	15665	574464	30102	236216	444272	502640
complex2	79	14840.7088608	1	30323	14786	20547	23421	28770
complex4	81	5367.37037037	680	20269	4749	9685	10517	16826
complex6	9	916878.222222	681920	1723136	827648	994592	1723136	1723136
complex7	60	97.7833333333	1	313	33	259	274	298
complex8	324	4849.62345679	156	25819	4334	10649	14089	22783
complex9	8	842535.25	96792	1635840	813248	986560	1635840	1635840
short1	1076	0.899628252788	0	213	1	1	2	3
short2	1076	655.096654275	2	13637	320	1619	2404	4443
short3	1076	28.282527881	0	1336	9	44	142	331
short4	1085	2.38709677419	0	1384	1	2	2	3
short5	1085	1.65622119816	0	66	1	3	3	4
short7	1085	4.97880184332	0	55	3	12	16	23
update1	3	3.66666666667	2	7	2	7	7	7
update2	298	4.05033557047	2	92	3	5	5	6
update3	383	3.56396866841	1	50	3	4	5	6
update6	204	3.65196078431	2	29	3	5	6	10
update7	203	6.66502463054	3	108	5	9	10	13
update8	34	3.23529411765	2	6	3	4	4	6

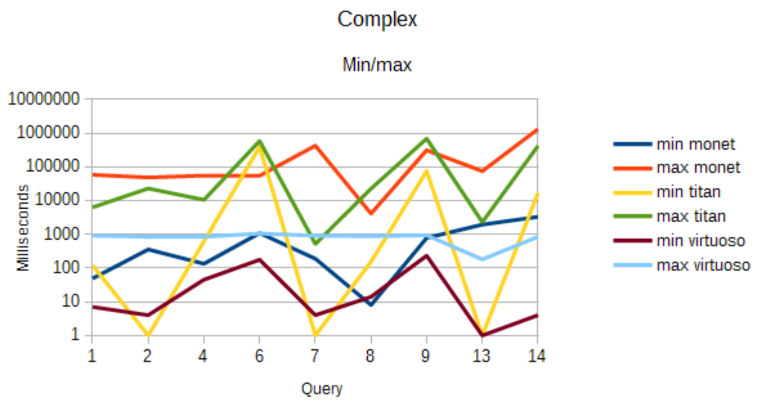
Table 11: Titan statistics on scale factor 1, 8 Cassandra backends.



(a) Count

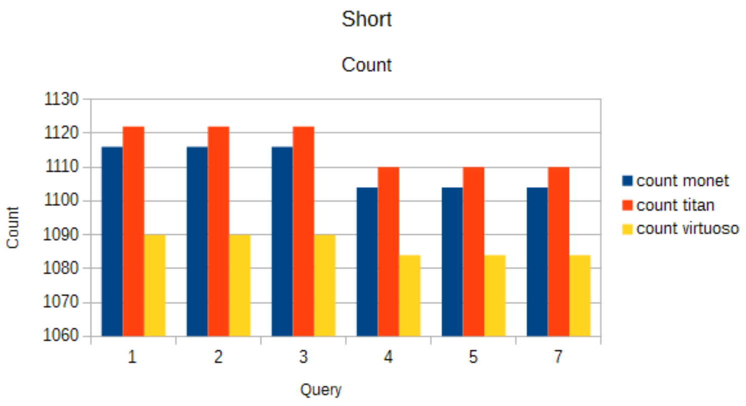


(b) Mean

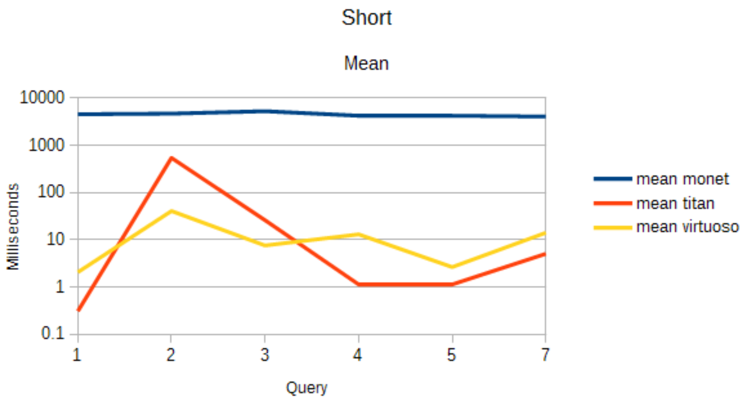


(c) Min/max

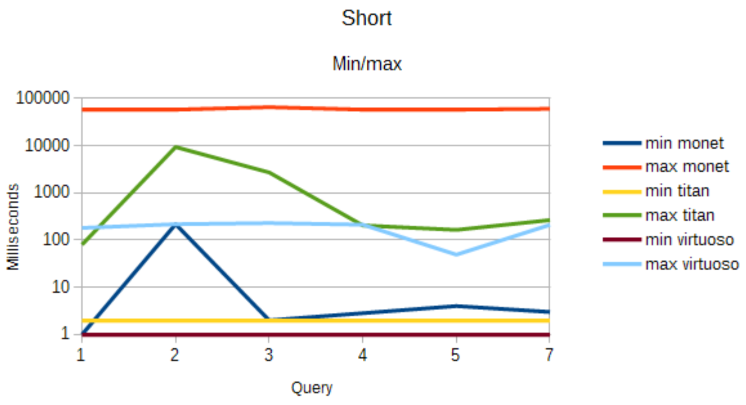
Figure 35: LDBC benchmark on SF1 data, complex queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

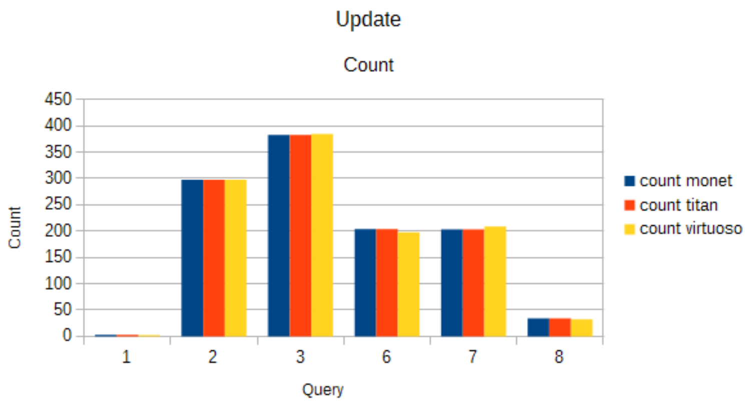


(b) Mean

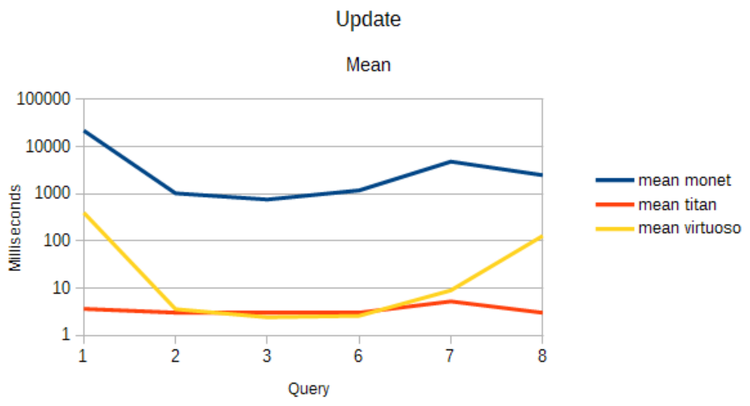


(c) Min/max

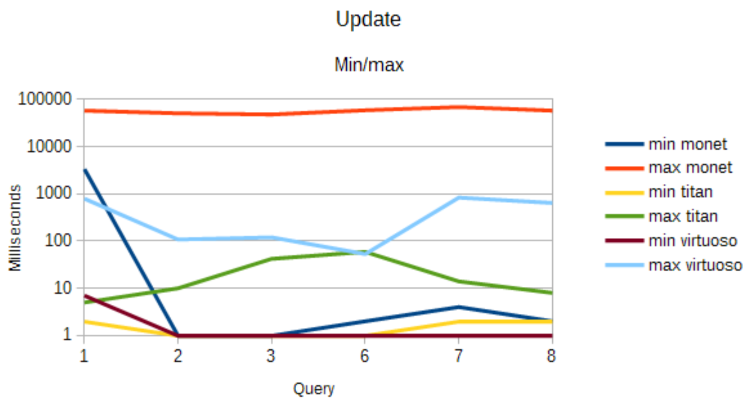
Figure 36: LDBC benchmark on SF1 data, short queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

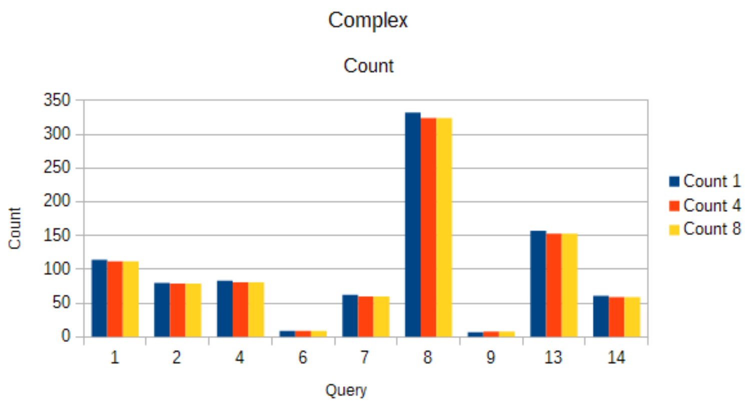


(b) Mean

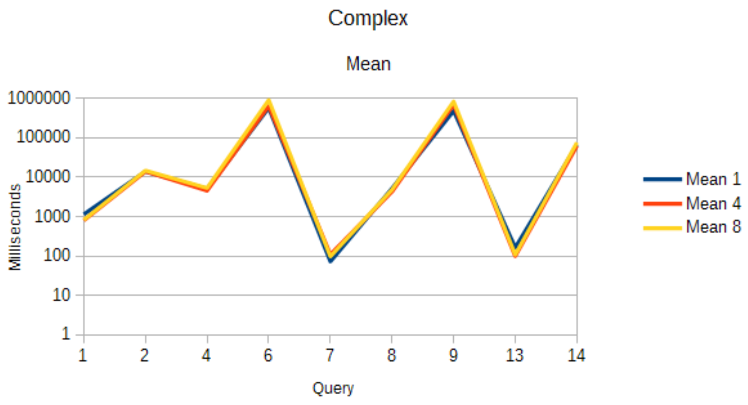


(c) Min/max

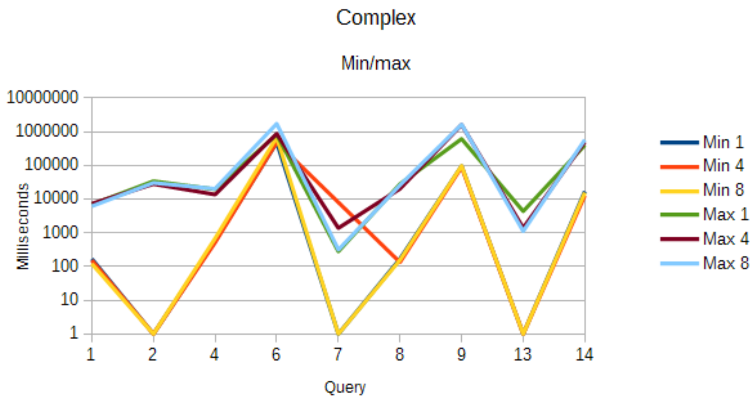
Figure 37: LDBC benchmark on SF1 data, update queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

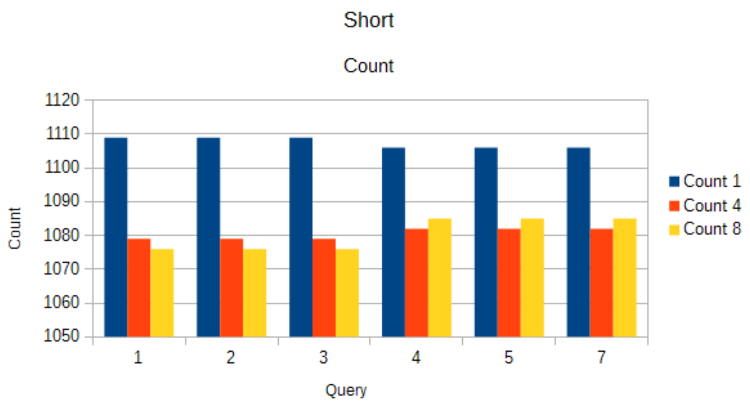


(b) Mean

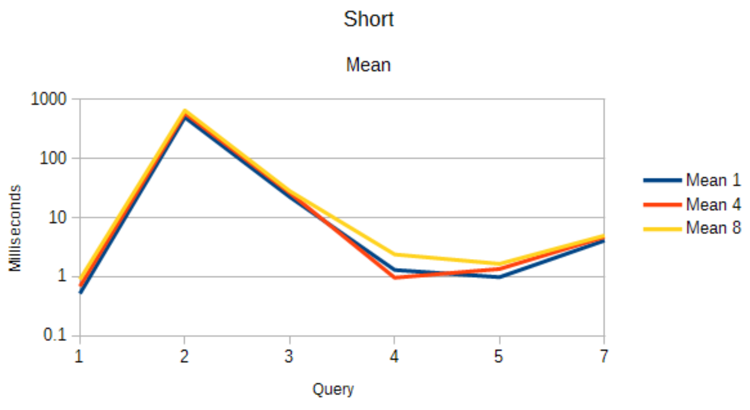


(c) Min/max

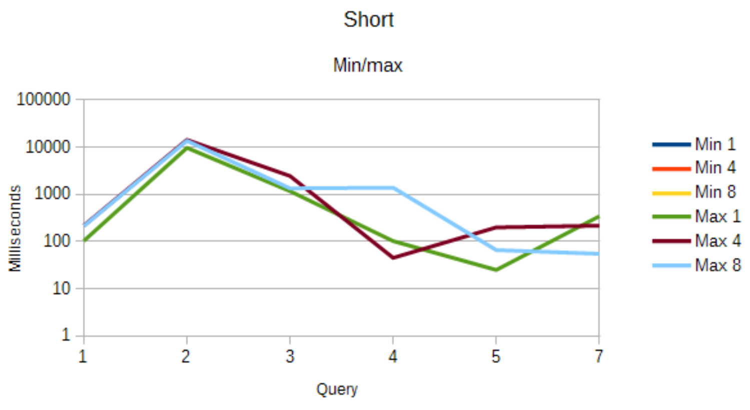
Figure 38: LDBC Titan benchmark on SF1 data, complex queries.



(a) Count

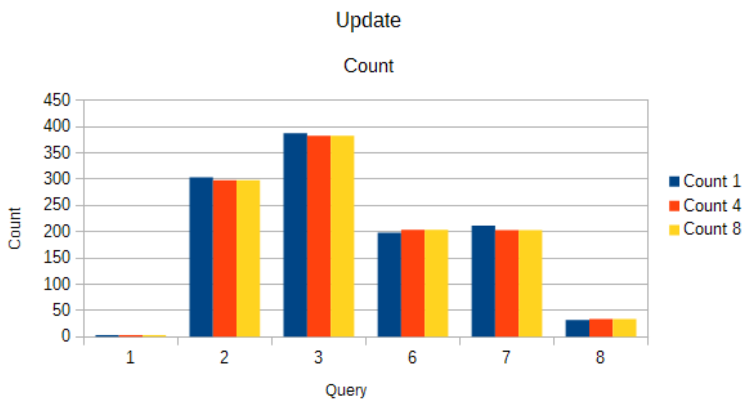


(b) Mean

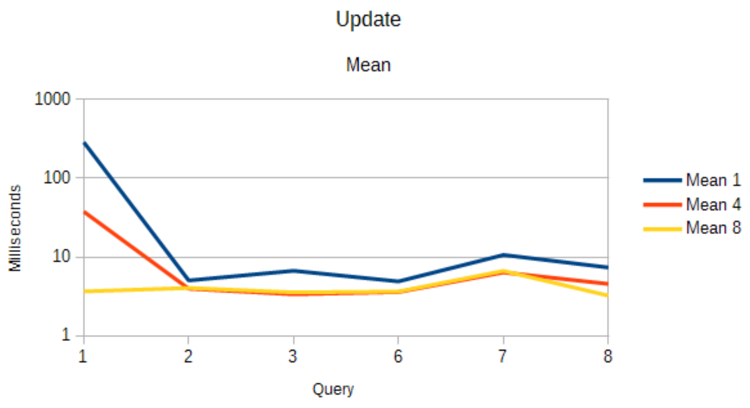


(c) Min/max

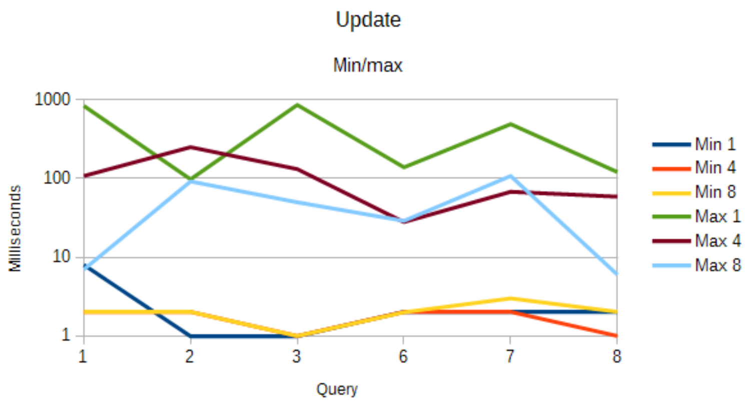
Figure 39: LDBC Titan benchmark on SF1 data, short queries.



(a) Count



(b) Mean



(c) Min/max

Figure 40: LDBC Titan benchmark on SF1 data, update queries.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	329	17461.7386018	115	422768	4104	45742	59716	305072
complex13	450	73766.3955556	51	587232	15086	267808	385072	458352
complex14	175	2591579.12	6614	5399808	2641408	5399808	5399808	5399808
complex2	231	22707.4112554	830	414992	5160	54592	126196	382288
complex4	238	22101.7478992	117	503280	4184	49956	89580	403616
complex6	27	21998.2592593	3798	200320	9226	12678	125224	200320
complex7	179	16483.7206704	240	432496	3858	11163	56684	360832
complex8	952	22381.9716387	7	507264	3515	49226	148944	387424
complex9	22	7201.27272727	1466	44900	5003	10939	10968	44900
short1	3245	19013.8089368	0	502512	3447	45710	128164	360848
short2	3245	21976.3830508	476	506848	4288	51254	152088	377008
short3	3245	25127.1812018	2	579488	4069	56042	181304	393024
short4	3260	20241.15	0	503280	3593	46710	145120	350640
short5	3260	20139.2315951	45	507232	3981	47036	128384	372736
short7	3260	19139.2828221	3	505472	3916	45172	126860	346000
update1	4	52399.75	8766	177016	10159	177016	177016	177016
update2	895	8749.94860335	1	459200	1	7436	12367	291472
update3	1524	8006.4488189	1	435056	1	8038	12876	227072
update6	389	9161.8714653	2	462304	3	12	14446	376528
update7	687	24114.9839884	3	570336	12	31531	201200	409024
update8	94	13628.0851064	2	309744	3	16244	54676	293472

Table 12: MonetDB statistics on scale factor 3.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	330	28.296969697	1	591	12	59	116	235
complex13	452	10.6128318584	1	425	2	41	42	59
complex14	175	14.1542857143	1	448	2	42	42	124
complex2	232	147.844827586	1	1411	2	550	884	1020
complex4	239	315.221757322	4	1848	209	759	932	1354
complex6	27	3072.25925926	168	6064	2882	4970	6045	6064
complex7	179	72.530726257	7	1058	10	204	265	730
complex8	956	105.432008368	0	917	22	349	473	774
complex9	22	1164.0	1	4802	2	4030	4362	4802
short1	3250	0.859076923077	0	16	1	1	2	4
short2	3250	73.6996923077	0	590	42	175	227	366
short3	3250	6.068	0	221	2	16	40	42
short4	3219	10.4836905871	0	216	1	41	41	57
short5	3219	1.09288598944	0	207	1	1	2	5
short7	3219	6.51071761417	0	227	1	40	41	44
update1	4	5.75	2	10	3	10	10	10
update2	897	1.38573021182	0	19	1	2	2	5
update3	1538	1.48764629389	0	73	1	2	2	8
update6	366	11.6612021858	0	1217	1	3	10	158
update7	695	26.0374100719	0	1984	2	18	97	635
update8	98	54.7040816327	1	1035	3	63	310	912

Table 13: Virtuoso statistics on scale factor 3.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	329	2929.73860182	930	16668	1975	6129	8690	13325
complex13	450	576.213333333	1	7058	336	1056	1680	4732
complex14	175	94304.2342857	1	310080	87184	152528	196032	282176
complex2	231	33725.4069264	169	50804	33634	44166	45654	49084
complex4	238	9061.82773109	700	28953	8888	14399	16882	20007
complex6	27	1557622.51852	1110272	2103424	1510720	1914944	1993088	2103424
complex7	179	78.1787709497	1	593	32	189	220	478
complex8	952	9017.12289916	998	32737	6874	17180	19632	25390
complex9	22	1606682.18182	1160832	2095616	1621504	1971840	1981760	2095616
short1	3235	0.474188562597	0	166	0	1	1	3
short2	3235	716.319938176	0	21912	253	1592	2803	8738
short3	3235	30.6608964451	0	2562	9	57	179	332
short4	3241	0.958346189448	0	286	1	1	2	4
short5	3240	1.25432098765	0	326	1	2	2	4
short7	3240	5.41635802469	0	1603	3	11	14	19
update1	4	4.75	2	10	2	10	10	10
update2	895	2.89273743017	0	270	2	4	5	7
update3	1524	2.52887139108	0	78	2	4	4	6
update6	389	2.27506426735	1	9	2	4	5	7
update7	687	4.54294032023	1	252	4	6	8	11
update8	94	2.48936170213	1	5	2	4	4	5

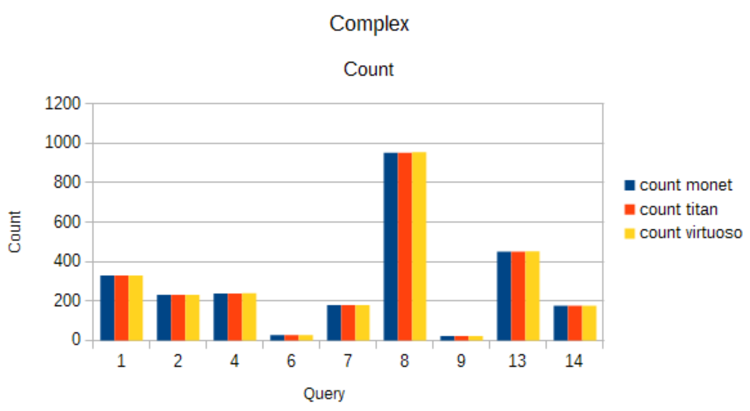
Table 14: Titan statistics on scale factor 3 (1 Cassandra backend).

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	329	3738.14589666	962	23307	2455	8881	10687	16372
complex13	450	665.371111111	1	13615	354	1423	2277	5298
complex14	175	98122.3771429	2	308688	94364	158088	207800	298592
complex2	231	36704.025974	566	59640	36562	47168	50406	57286
complex4	238	11791.1428571	66	30197	11680	20017	21845	27922
complex6	27	2082331.25926	1654272	3645696	1996800	2373632	2497024	3645696
complex7	179	77.4413407821	1	450	36	193	223	270
complex8	952	10114.4926471	1053	40666	7680	18963	22119	27960
complex9	22	2189861.81818	1522432	3042816	2044608	2693632	2911232	3042816
short1	3225	0.530852713178	0	77	0	1	1	2
short2	3224	788.86662531	0	21164	289	1802	2900	10085
short3	3224	31.8830645161	0	1045	9	66	180	369
short4	3241	1.05584696081	0	269	1	1	2	3
short5	3241	1.28170317803	0	67	1	2	2	3
short7	3241	5.58284480099	0	451	3	13	15	22
update1	4	4.75	2	10	3	10	10	10
update2	895	2.52290502793	1	47	2	3	4	5
update3	1524	2.34842519685	1	41	2	3	4	4
update6	389	2.41388174807	1	38	2	4	5	7
update7	687	6.41775836972	1	1390	4	7	8	13
update8	94	2.94680851064	1	63	2	4	4	6

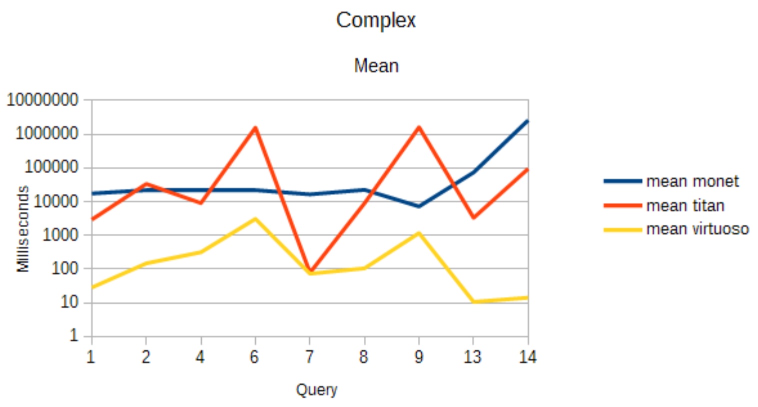
Table 15: Titan statistics on scale factor 3 (4 Cassandra backends).

query	count	mean	min	max	50th perc.	90th perc.	95th perc.	99th perc.
complex1	329	4189.04255319	857	19625	2738	9844	12181	19061
complex13	450	638.748888889	0	11311	327	1316	2464	5243
complex14	175	101650.342857	2	318160	98224	164264	205992	304384
complex2	231	41448.3246753	687	69708	41276	53552	55698	64246
complex4	238	13793.6470588	63	35840	14112	22586	26568	30792
complex6	27	2432103.11111	1648960	4141312	2395008	2841088	3102080	4141312
complex7	179	104.402234637	1	2626	44	226	260	452
complex8	952	11275.9789916	1527	42062	9068	20855	23611	31314
complex9	22	2670088.72727	1627712	3867008	2518784	3324160	3631232	3867008
short1	3253	0.84260682447	0	70	1	1	1	2
short2	3253	917.412849677	0	21553	362	1950	2992	11953
short3	3253	35.7568398401	0	2461	10	71	194	408
short4	3286	1.17894096166	0	49	1	2	2	3
short5	3285	1.67397260274	0	298	1	2	3	4
short7	3285	6.47579908676	0	338	3	15	18	28
update1	4	3.0	2	5	2	5	5	5
update2	895	2.77765363128	0	68	3	4	4	5
update3	1524	3.58595800525	1	1393	2	4	4	6
update6	389	2.92544987147	1	76	2	5	6	10
update7	687	4.87190684134	2	65	4	7	9	13
update8	94	2.3829787234	1	5	2	3	3	5

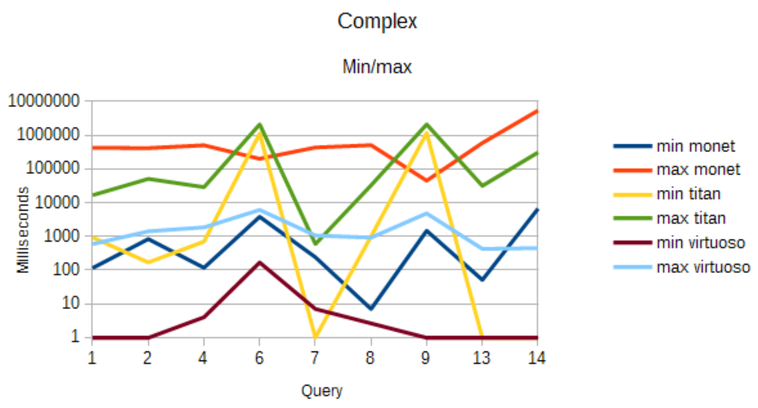
Table 16: Titan statistics on scale factor 3 (8 Cassandra backends).



(a) Count

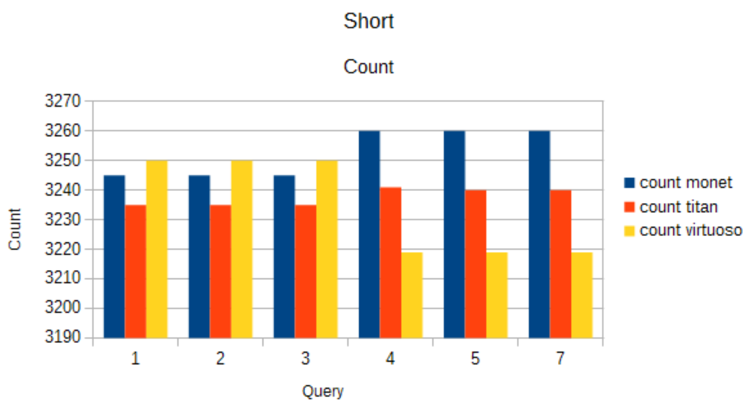


(b) Mean

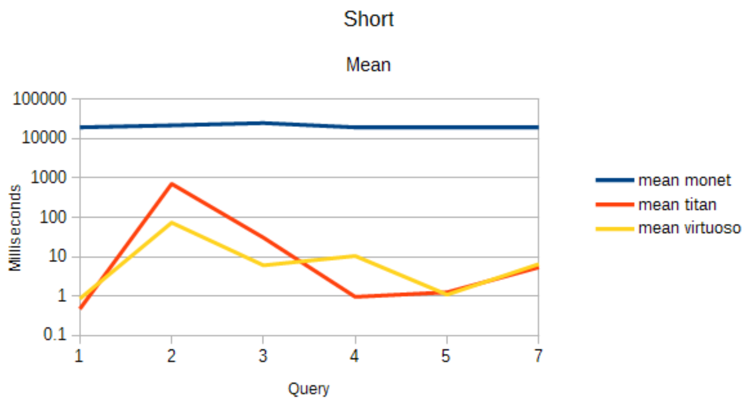


(c) Min/max

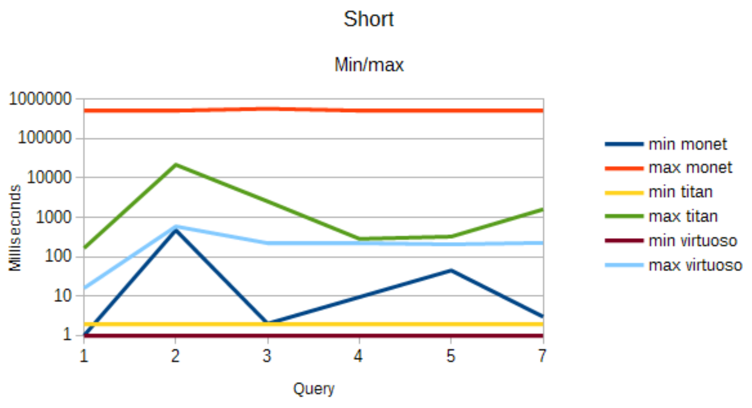
Figure 41: LDBC benchmark on SF3 data, complex queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

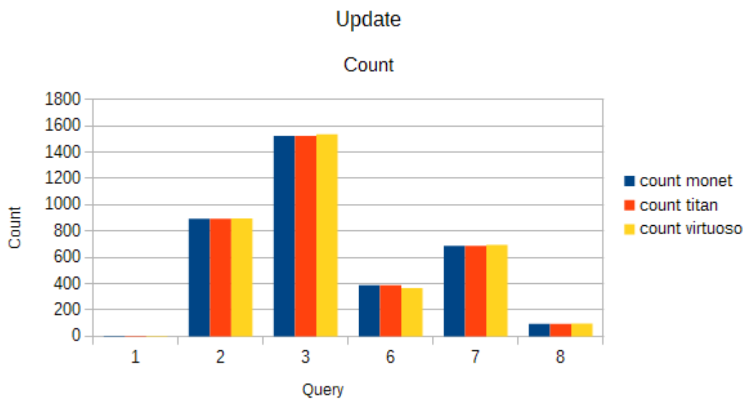


(b) Mean

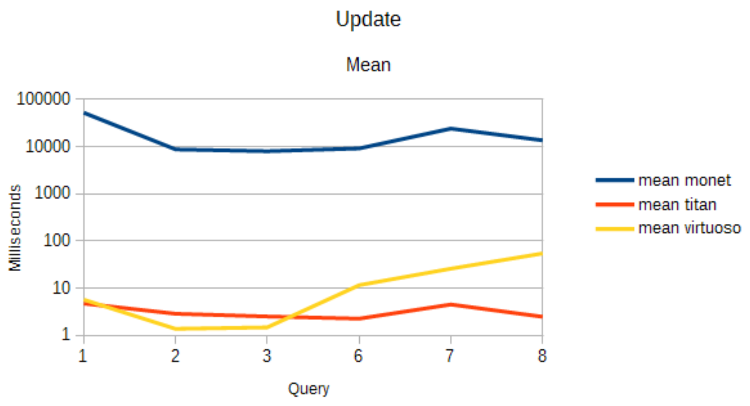


(c) Min/max

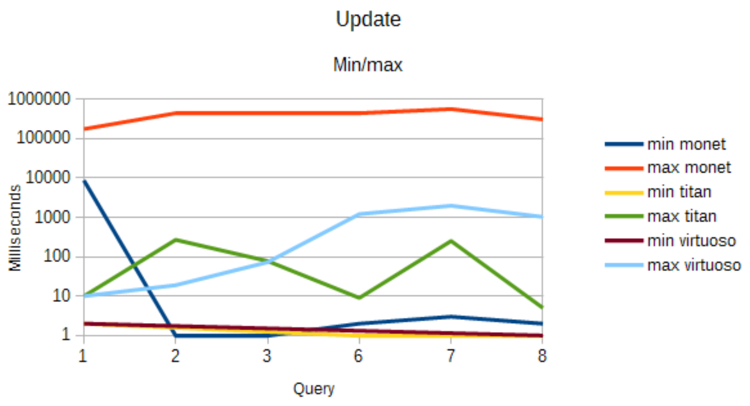
Figure 42: LDBC benchmark on SF3 data, short queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

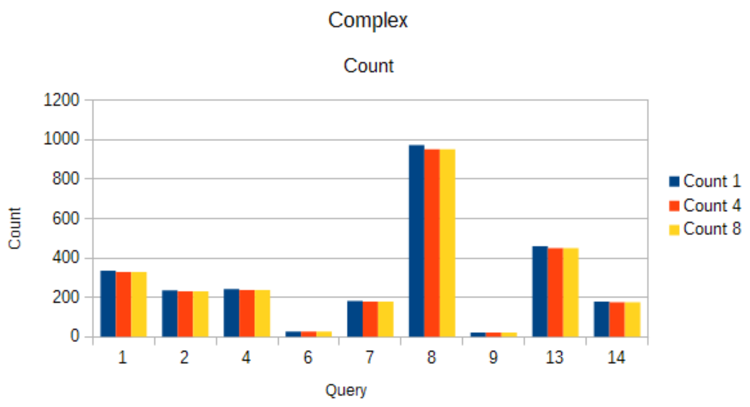


(b) Mean

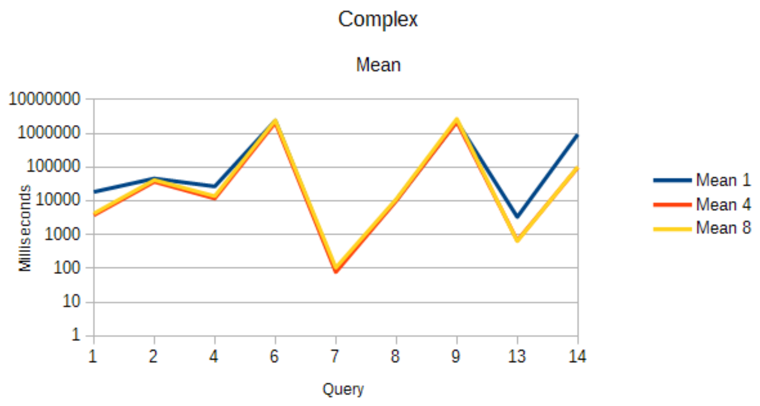


(c) Min/max

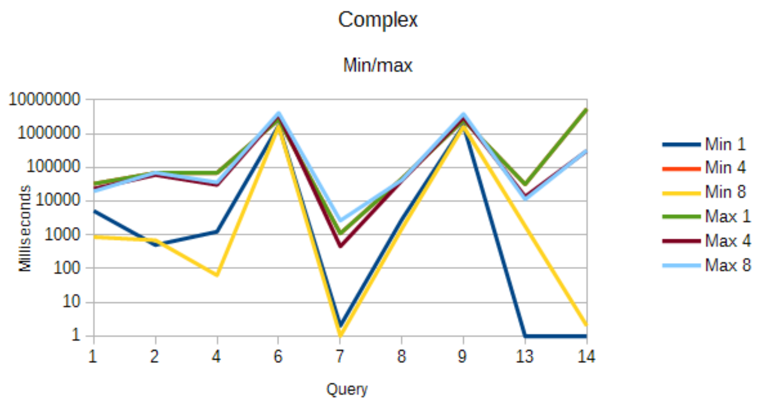
Figure 43: LDBC benchmark on SF3 data, update queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

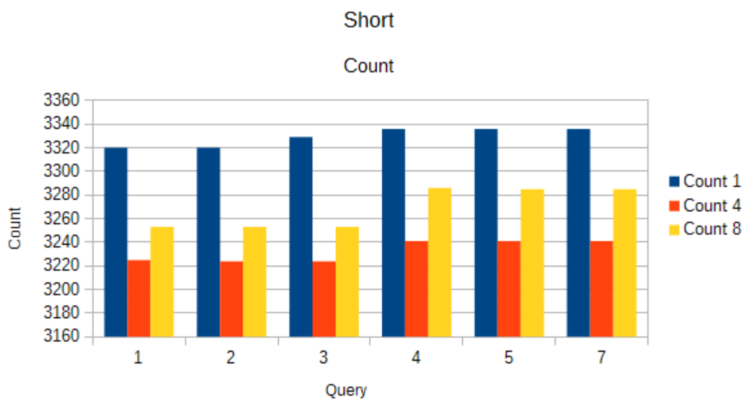


(b) Mean

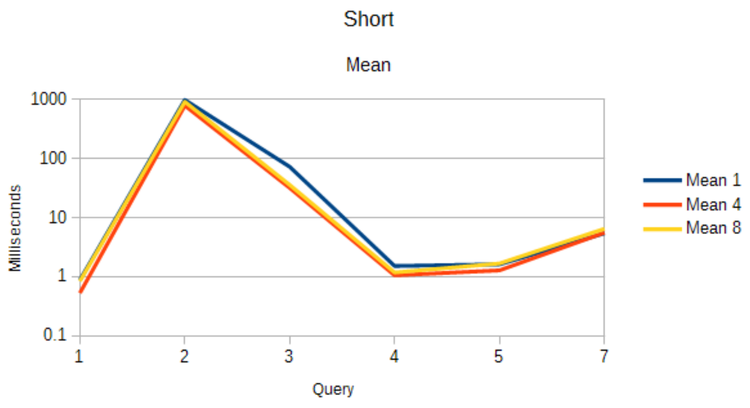


(c) Min/max

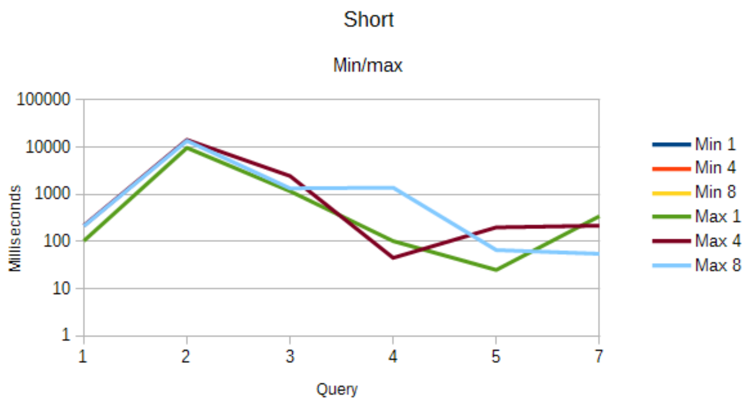
Figure 44: LDBC Titan benchmark on SF3 data, complex queries.



(a) Count

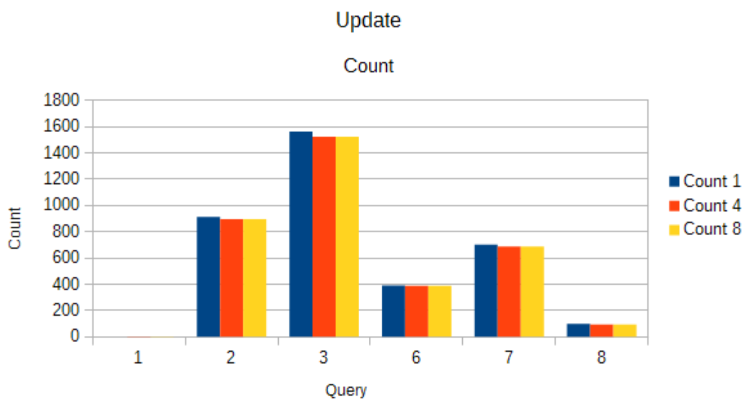


(b) Mean

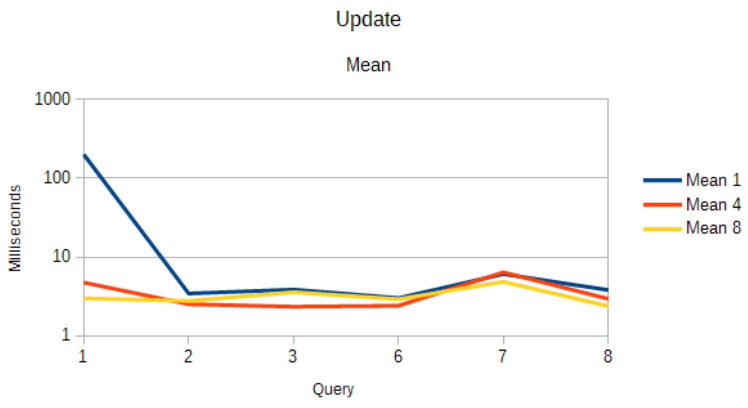


(c) Min/max

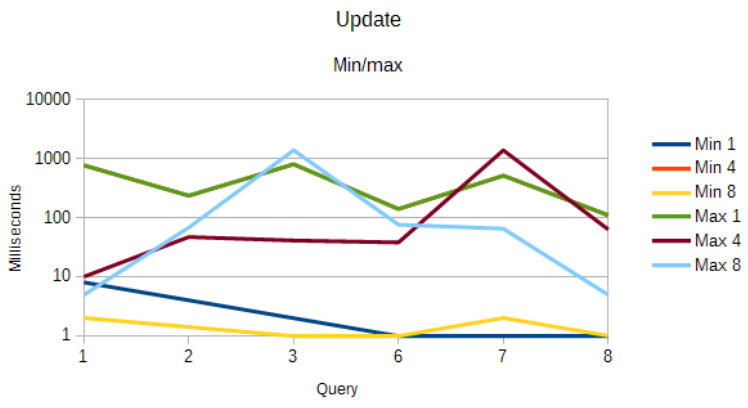
Figure 45: LDBC Titan benchmark on SF3 data, short queries.



(a) Count



(b) Mean



(c) Min/max

Figure 46: LDBC Titan benchmark on SF3 data, update queries.

query	count	mean	min	max	50th perc.	90th perc.	95th perc.	99th perc.
complex1	13	258419.923077	398	3089408	25509	42262	42262	3089408
complex13	18	1072008.77778	14995	3688960	169096	3496064	3503360	3688960
complex14	7	3911351.42857	171488	5399808	5399808	5399808	5399808	5399808
complex2	9	377759.0	2767	3186688	27637	65720	3186688	3186688
complex4	10	20544.7	1565	59452	5610	54378	59452	59452
complex6	1	25228.0	25228	25228	25228	25228	25228	25228
complex7	7	27031.1428571	9781	37772	28976	34818	37772	37772
complex8	39	192062.897436	3959	3179904	30965	61092	115464	3179904
short1	119	169788.647059	1	3179904	21545	59130	189488	3158400
short2	119	30927.9663866	1966	192832	25422	53980	67600	171168
short3	119	330615.571429	4	3369984	31647	207472	3123584	3355520
short4	112	255951.830357	3	3191552	28074	146768	3125760	3171200
short5	112	112851.3125	3	3108096	27086	64224	77472	3096832
short7	112	211076.223214	8	3184640	27031	74396	1513792	3164288
update2	43	4639.8372093	4	178864	36	1568	3766	178864
update3	79	46615.7974684	3	3377280	30	1544	11421	204968
update6	16	17931.125	8	139288	2616	19374	102956	139288
update7	35	150207.657143	7	3380480	11	10142	171728	3380480
update8	8	56.625	6	403	7	9	403	403

Table 17: MonetDB statistics on scale factor 10.

query	count	mean	min	max	50th perc.	90th perc.	95th perc.	99th perc.
complex1	1090	18.8091743119	7	599	12	45	47	89
complex13	1492	5.74195710456	1	593	2	5	41	43
complex14	578	4.73875432526	1	132	2	5	8	43
complex2	766	14.6422976501	1	879	2	15	33	375
complex4	787	127.763659466	4	2591	105	187	257	898
complex6	89	1189.59550562	6	3790	1194	1865	2280	3008
complex7	590	29.7542372881	3	932	8	49	77	445
complex8	3151	14.568390987	0	911	6	41	46	93
complex9	73	63.7123287671	1	1451	2	138	354	984
short1	10711	1.51769209224	0	276	1	3	4	6
short2	10711	36.7428811502	0	633	41	45	79	207
short3	10711	3.67444683036	0	435	2	6	8	41
short4	10773	7.8243757542	0	599	1	41	41	43
short5	10773	2.6619326093	0	432	1	3	5	41
short7	10772	11.7257705162	0	621	3	42	43	56
update1	10	63.7	2	604	3	8	604	604
update2	3198	1.62382739212	0	265	1	3	4	6
update3	5566	1.59935321595	0	168	1	3	4	6
update6	984	4.77134146341	0	871	2	5	8	56
update7	2230	5.07668161435	0	665	2	6	14	66
update8	295	24.0203389831	0	956	2	38	119	499

Table 18: Virtuoso statistics on scale factor 10.

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	1089	25814.2066116	1	311584	15000	49324	80316	241008
complex13	1492	7291.71715818	1	205336	2473	13401	36784	79764
complex14	579	239826.83247	1	2315648	181736	447072	590368	1242688
complex2	765	55452.4888889	857	326976	45856	88024	128436	249288
complex4	787	52424.0775095	1	506096	40550	83408	136872	275280
complex6	90	4833619.2	3287040	5399808	5024512	5399808	5399808	5399808
complex7	591	118.490693739	1	8764	29	217	267	555
complex8	3149	99.4045728803	1	32250	35	142	203	364
complex9	74	4683067.72973	218880	5399808	4790016	5399808	5399808	5399808
short1	10755	0.660344026034	0	359	0	1	1	3
short2	10754	1616.59010601	0	114212	494	2771	6279	26501
short3	10754	94.3448019342	0	33778	15	111	250	615
short4	10815	6.91558021267	0	30430	1	2	2	6
short5	10815	1.57993527508	0	93	1	2	3	5
short7	10815	17.1420249653	0	31823	5	16	20	30
update1	10	3.1	2	7	2	5	7	7
update2	3194	3.03600500939	0	1726	2	4	4	7
update3	5553	2.5586169638	0	345	2	4	4	7
update6	991	2.03229061554	0	46	2	4	5	7
update7	2227	3.65289627301	0	307	3	6	7	11
update8	293	2.35153583618	0	62	2	3	4	6

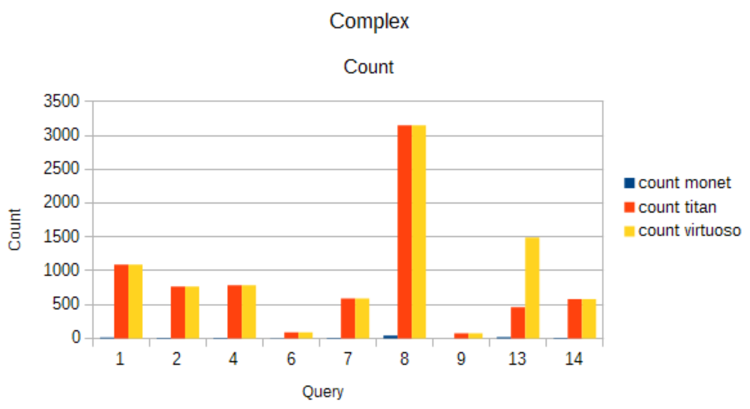
Table 19: Titan statistics on scale factor 10 (1 Cassandra backend).

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	1089	22890.3425161	1	203624	15227	46948	62938	140112
complex13	1492	5039.23659517	1	213344	1967	9480	20184	52924
complex14	579	178000.319516	1	981632	151544	307872	402256	581120
complex2	765	57281.6718954	4014	219120	50388	96452	130128	161480
complex4	787	51411.9707751	1	226744	44276	85456	115868	168776
complex6	90	4979133.86667	2885760	5399808	5247232	5399808	5399808	5399808
complex7	591	73.730964467	1	683	28	209	241	375
complex8	3149	67.4163226421	3	35254	32	117	162	284
complex9	74	4886299.2973	220512	5399808	5125632	5399808	5399808	5399808
short1	10796	1.12189699889	0	3492	1	1	1	2
short2	10795	1593.46465956	0	132992	532	2973	5593	19413
short3	10795	71.2566929134	0	33750	16	133	269	734
short4	10857	1.42967670627	0	410	1	2	2	4
short5	10857	5.10988302478	0	29254	1	2	3	4
short7	10857	14.1015013355	0	37574	4	16	19	30
update1	10	2.9	2	6	2	4	6	6
update2	3194	2.58484658735	0	208	2	3	4	6
update3	5553	2.52332072753	0	393	2	3	4	6
update6	991	2.4702320888	1	305	2	4	5	8
update7	2227	4.64032330489	1	1513	3	6	8	12
update8	293	2.18088737201	0	8	2	3	4	6

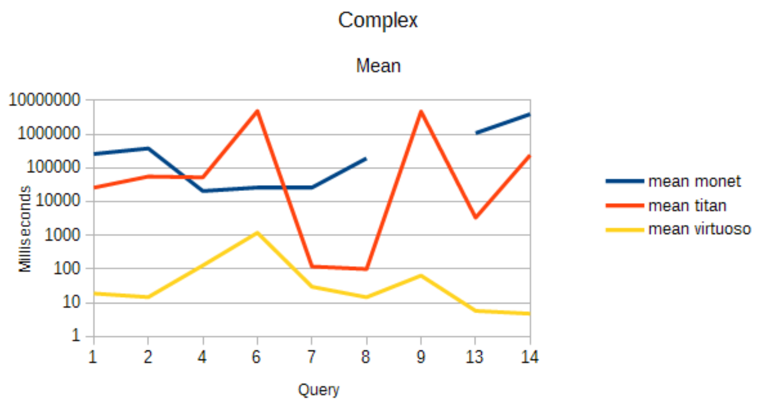
Table 20: Titan statistics on scale factor 10 (4 Cassandra backends).

query	**count**	**mean**	**min**	**max**	**50th perc.**	**90th perc.**	**95th perc.**	**99th perc.**
complex1	1089	26923.3425161	1	400592	15649	52304	74412	252704
complex13	1492	5516.28686327	1	118904	2236	10611	27608	52444
complex14	579	183720.174439	2	1757952	140664	329472	445776	774752
complex2	765	62597.0980392	3724	289200	54796	92156	144600	212856
complex4	787	61037.935197	1	579488	51738	85944	135240	233352
complex6	90	5213671.82222	2318592	5399808	5399808	5399808	5399808	5399808
complex7	591	74.7495769882	1	510	29	206	237	359
complex8	3149	63.2372181645	1	10395	35	128	174	292
complex9	74	5177670.16216	282176	5399808	5399808	5399808	5399808	5399808
short1	10749	0.993673830124	0	248	1	1	2	2
short2	10748	1811.24581317	0	109360	594	3323	7004	24478
short3	10748	77.3903982136	0	31801	18	128	272	720
short4	10804	10.5603480193	0	33638	1	2	2	3
short5	10804	4.75601629026	0	29632	2	3	3	4
short7	10804	8.88624583488	0	8346	4	17	21	32
update1	10	3.2	2	5	3	4	5	5
update2	3194	2.73356293049	1	63	3	4	4	5
update3	5553	2.92688636773	0	1265	2	4	4	5
update6	991	2.67507568113	1	93	2	4	5	10
update7	2227	4.77503367759	1	1119	4	6	8	11
update8	293	3.44368600683	1	239	2	3	4	6

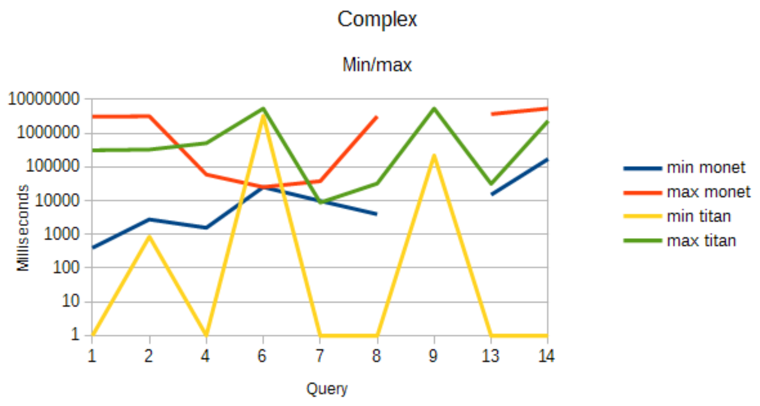
Table 21: Platform statistics on scale factor 10 (8 Cassandra backends).



(a) Count

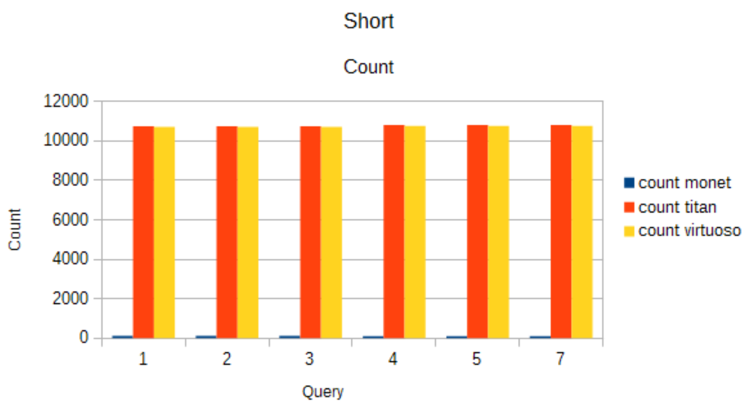


(b) Mean

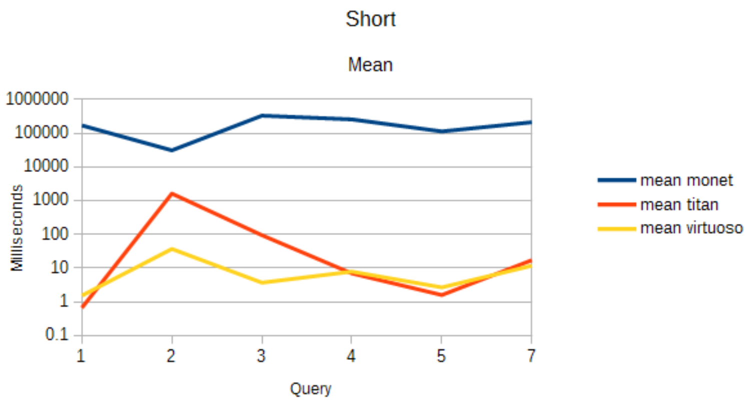


(c) Min/max

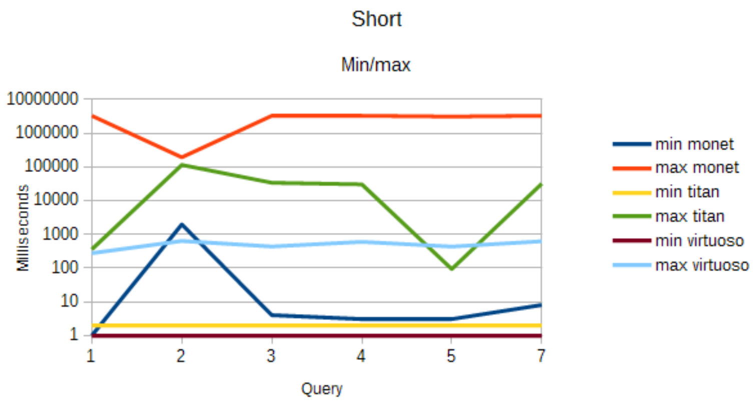
Figure 47: LDBC benchmark on SF10 data, complex queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

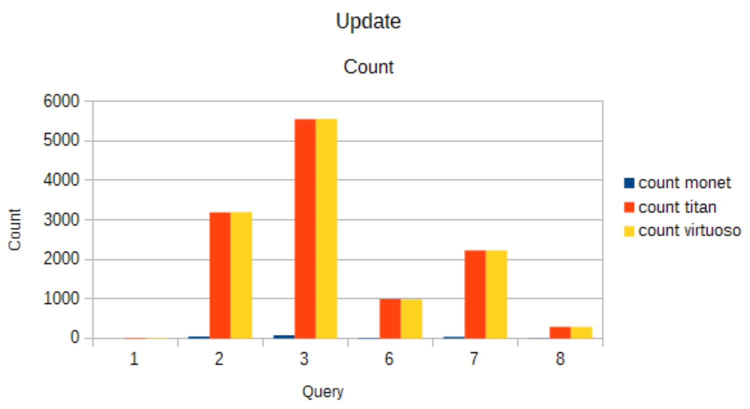


(b) Mean

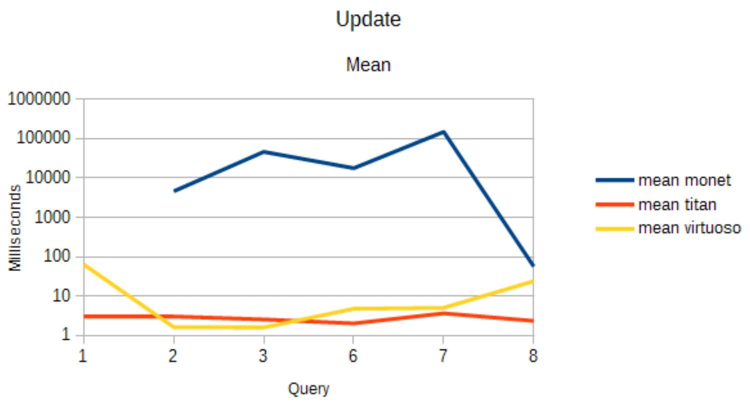


(c) Min/max

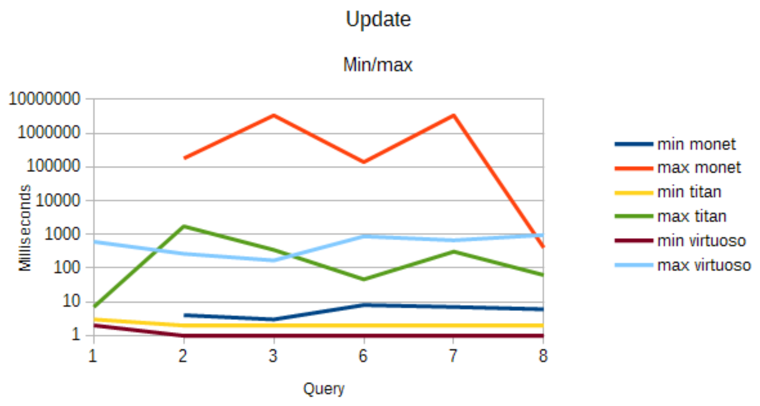
Figure 48: LDBC benchmark on SF10 data, short queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

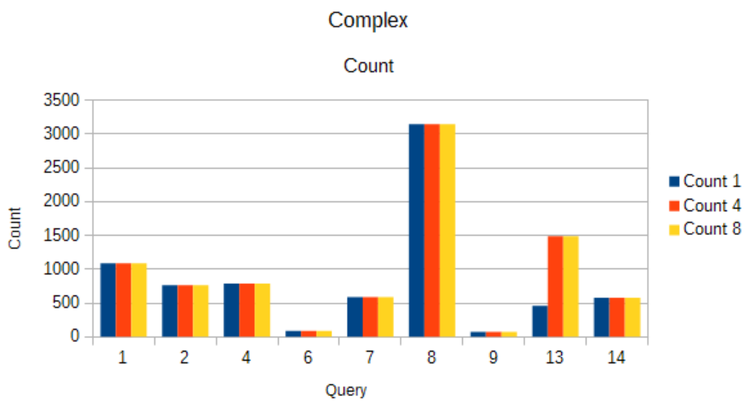


(b) Mean

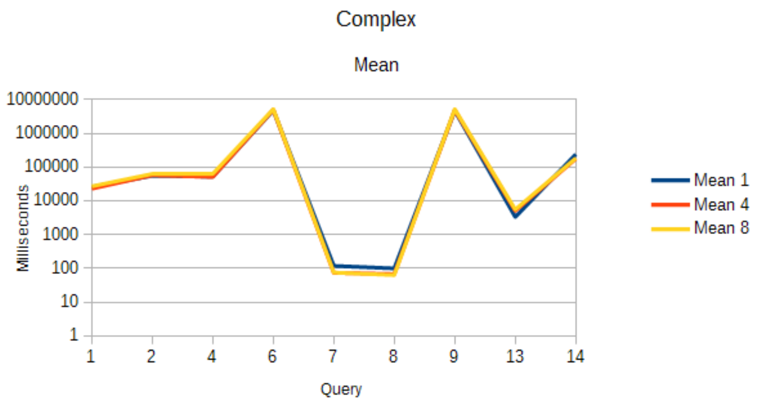


(c) Min/max

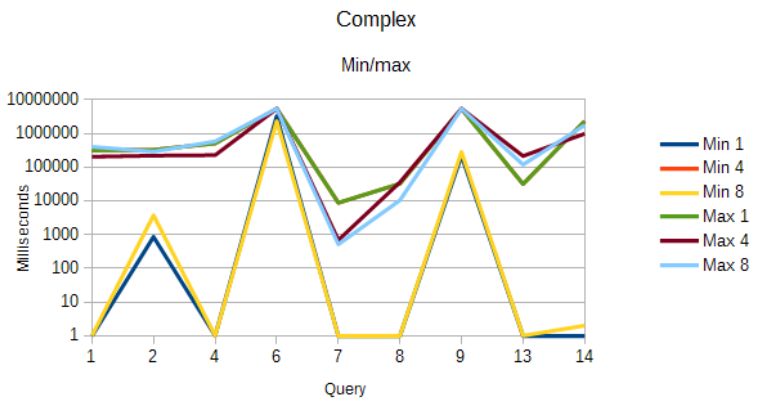
Figure 49: LDBC benchmark on SF10 data, update queries. The Titan data shown in the graphs uses one Cassandra backend.



(a) Count

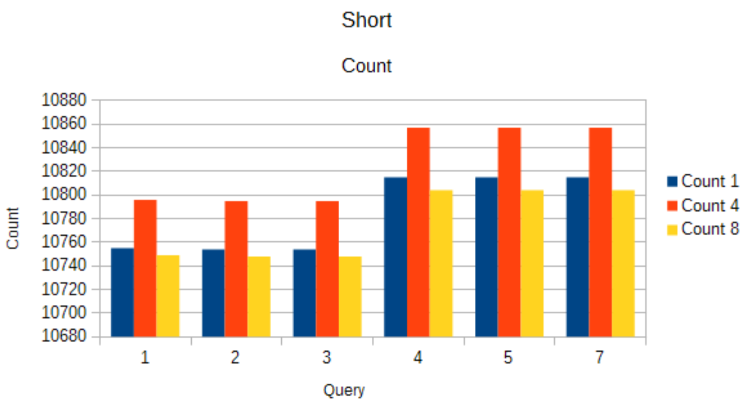


(b) Mean

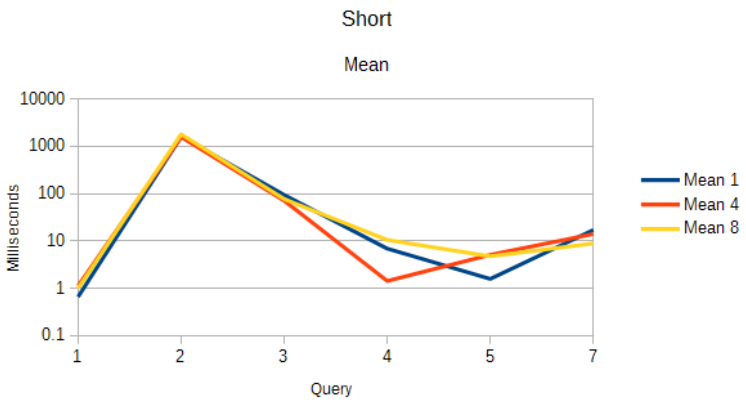


(c) Min/max

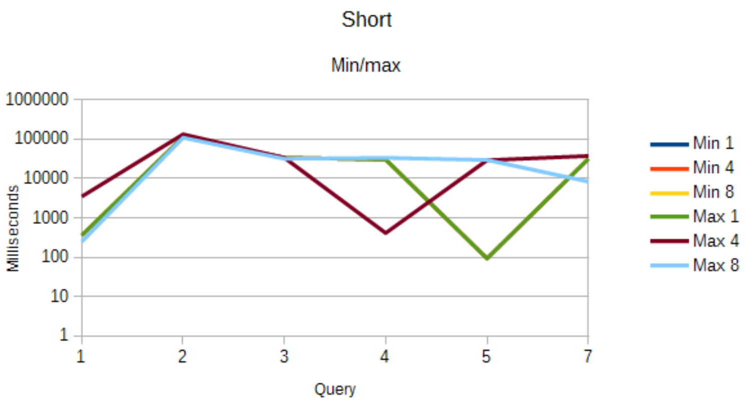
Figure 50: LDBC Titan benchmark on SF10 data, complex queries.



(a) Count

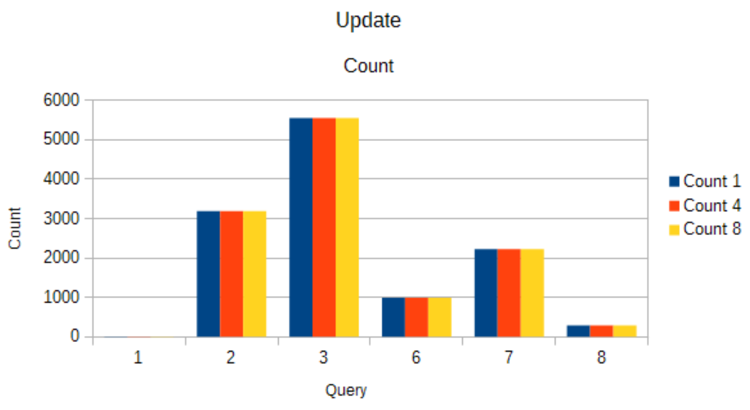


(b) Mean

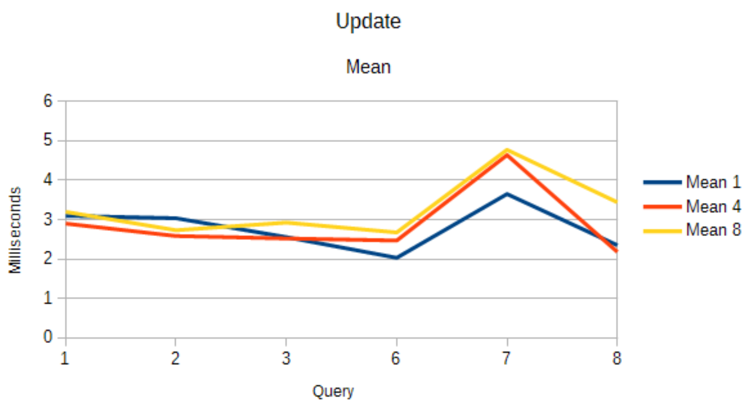


(c) Min/max

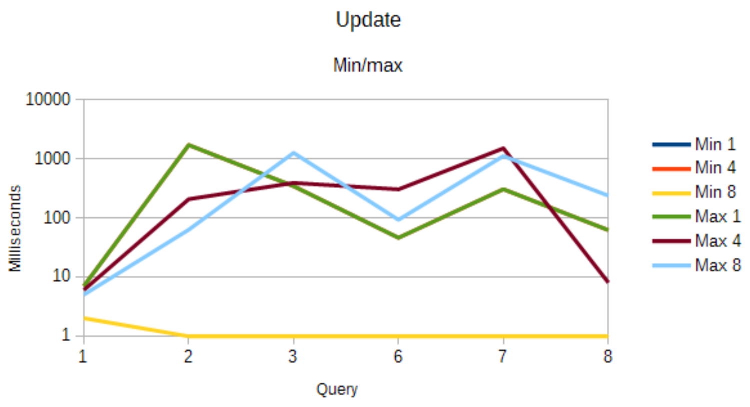
Figure 51: LDBC Titan benchmark on SF10 data, short queries.



(a) Count



(b) Mean



(c) Min/max

Figure 52: LDBC Titan benchmark on SF10 data, update queries.

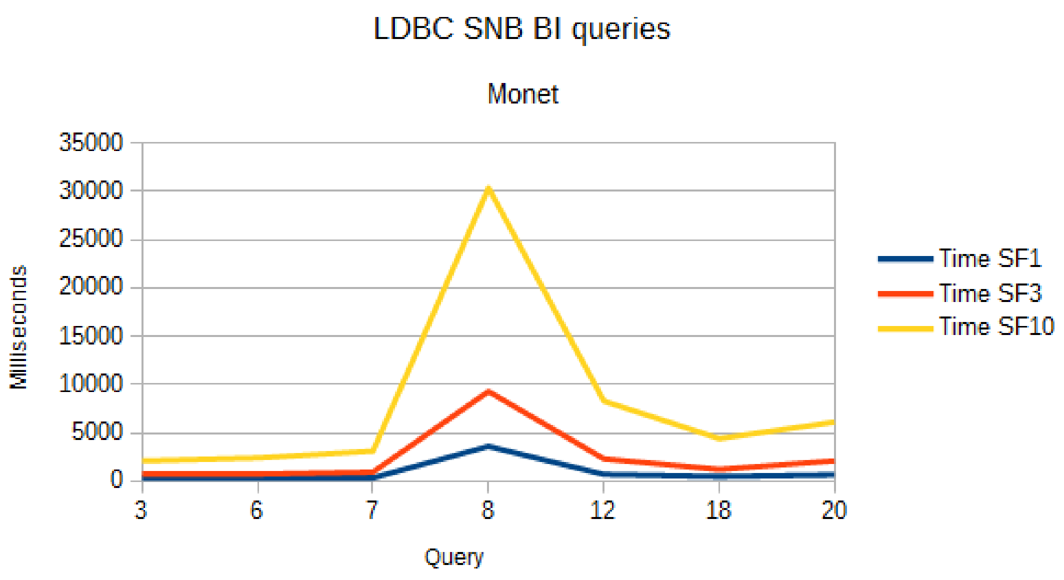


Figure 53: Execution time in milliseconds for LDBC SNB BI queries on MonetDB on scale factors 1, 3 and 10.