LDBC

Cooperative Project

FP7 – 317548

D2.2.3 Benchmarking Transactions

Coordinator: Andrey Gubichev

With contributions from: Alex Averbuch, Orri Erling

1st Quality Reviewer: Peter Boncz
2nd Quality Reviewer: Venelin Kotsev

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M24
Actual delivery date:	M24
Version:	1.0
Total number of pages:	33
Keywords:	transaction processing, distributed systems

Abstract

We describe the composition of the Interactive (transactional) Workload of SNB, namely the proportion between the read and write queries. In addition, we introduce a new type of read queries to balance the relatively long running SNB queries and SNB updates. In the second part of the deliverable, we describe the architecture and properties of our LDBC workload driver aimed at executing the transactional workload in a scalable and repeatable way.

EXECUTIVE SUMMARY

First part of the deliverable describes the query mix of the Interactive (transactional) Workload of LDBC Social Network Benchmark. By experimenting with different workload definitions we arrive at the following distribution of cumulative runtime of reads and writes: 10% update queries (from SNB data generator), 55% long read queries (SNB queries), 35% short read queries, where short reads are of the following type:

- profile view: one-join query that looks up the basic information about the given user and his friends
- message view: short lookup of the message (content, date) and its sender

In addition, we propose a random-walk based procedure to incorporate short reads into the workload: for every long read query (i.e. for one of the set of 14 SNB queries), its return set (messages or users) is used as starting point to the series of profile and message views. We present several measurements of the workload run (e.g., cache misses, CPU utilization etc.) on Virtuoso 7 Column Store

Once we have defined the workload, we need the driver that executes it in a scalable (i.e., parallel, distributed) manner. The key characteristic of the LDBC transactional workload is the dependencies between different (update) transactions: one can not create messages before the profile of the sender is created; same holds for friendship requests and various other user actions in the network. The driver has to, therefore, make sure that the (partial) order between dependent operations is enforced when *issuing* operations against the System Under Test. Moreover, the driver has to execute the workload in a repeatable way: the data generator outputs the update operations that follow a strict scenario (i.e., they have assigned timestamps in the simulation time), and driver has to execute the workload in parallel remaining true to the generated operation stream. This allows a fair and repeatable comparison of different systems. The second part of the deliverable describes the main principles and the architecture of the driver that satisfies these requirements. The driver is available on Github, see `https://github.com/ldbc/ldbc_driver/`

DOCUMENT INFORMATION

IST Project Number	FP7 – 317548	**Acronym**	LDBC
Full Title	LDBC		
Project URL	http://www.ldbc.eu/		
Document URL			
EU Project Officer	Carola Carstens		

Deliverable	**Number**	D2.2.3	**Title**	Benchmarking Transactions
Work Package	**Number**	WP2	**Title**	Query Processing

Date of Delivery	**Contractual**	M24	**Actual**	M24
Status	version 1.0		final ☒	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

Authors (Partner)	Andrey Gubichev (TUM)			
Responsible Author	**Name**	Andrey Gubichev	**E-mail**	gubichev@in.tum.de
	Partner	TUM	**Phone**	+49

Abstract (for dissemination)	We describe the composition of the Interactive (transactional) Workload of SNB, namely the proportion between the read and write queries. In addition, we introduce a new type of read queries to balance the relatively long running SNB queries and SNB updates. In the second part of the deliverable, we describe the architecture and properties of our LDBC workload driver aimed at executing the transactional workload in a scalable and repeatable way.
Keywords	transaction processing, distributed systems

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
22/09/2014	0.1	Andrey Gubichev	First version
30/09/2013	1.0	Andrey Gubichev	Final version

TABLE OF CONTENTS

1 INTRODUCTION

In this deliverable we consider two issues related to the Interactive workload of the LDBC Social Network Benchmark. In the first part, we complete the design of the workload by adding the update queries and defining the mix ratios between reads and writes in the workload: reads are the SNB queries, writes are the update operations generated by the data generator. The frequency of these operations should both be realistic (i.e., writes should not dominate the workload) and technically interesting (i.e., SNB queries should occupy a significant portion of the workload).

In the second part of the deliverable, we describe the workload driver. Its mission is to execute the given Interactive workload (i.e., mix of read and update queries) in a repeatable and scalable manner. The Workload driver achieves it by running streams of operations in parallel while respecting dependencies between operations (e.g., posts in the social network are created only after the corresponding user profiles are created). The fact that these dependencies may span several operation streams run in parallel makes the problem challenging and distinct from what state-of-the-art benchmark drivers (such as TPC-C) do. Another requirement that we put in the design of the driver is repeatable execution of the transactional workload: the data generator outputs the sequence of update operations that form a certain scenario in simulation time, and the driver has to run this scenario *as is* against any SUT. The SUT, in turn, has to keep up with the load generated by the driver, or fail the test run. The load that the system is put under (e.g. number of transactions per second) is configurable and is part of the benchmark score.

This deliverable describes principles that are implemented in the open-source LDBC Driver [1].

[1] See `https://github.com/ldbc/ldbc_driver/` for software and documentation

2 UPDATE WORKLOAD IN SOCIAL NETWORK BENCHMARK

2.1 Background and Motivation

The Social Network Benchmark aims at mimicking user behavior on the social network website. Therefore, together with already described read queries (e.g. *see most recent updates from my friends*, see Task Force Report), the plausible workload has to contain update operations as well, such as *add friend*. Update operations in SNB are generated by the data generator. All the generated data is divided into two parts: one is bulk loaded, another (a smaller one) is "played out" as updates to the network. Overall, the interactive workload consists of updates played out from the generated data, and read queries issued with a certain frequency.

Our goal in this chapter is to define the specific mix of update and read queries of SNB Interactive Workload, that is, to define the frequency of update and read operations in the workload.

We will argue that frequency distribution of operations in the workload has to fit two patterns:

- it has to be *realistic*, i.e. has to correspond to the real scenario of our vertical domain
- it has to be *technically challenging*, i.e. the workload should not be dominated by point lookups and updates

We will show that it is impossible to satisfy both of these requirements with a single query mix composed of only SNB read queries and updates, and propose a compromise workload that stays as close to realistic as possible, while remaining somewhat challenging from the query optimization point of view.

2.2 Workload Queries

In this section we briefly re-visit the queries that form the SNB Interactive Workload. They are described in full details in Deliverable 3.3.3, so here we just list them for completeness:

Read Queries

- **Q1**: friends with a certain first name
- **Q2**: recent posts and comments by your friends
- **Q3**: friends and friends of friends that have been to countries X and Y
- **Q4**: new topics that have been discussed by your friends or friends of friends within last 24 hours but not before
- **Q5**: groups that your friends or friends of friends have joined in the last 24 hours
- **Q6**: most popular hashtags that are co-occur with a given tag in posts of your friends and friends of friends
- **Q7**: recent likes of your posts
- **Q8**: recent replies to your posts and comments
- **Q9**: recent posts and comments of your friends and friends of friends
- **Q10**: friend recommendation
- **Q11**: job referral. Find friends and friends of friends that have joined some company before a given date
- **Q12**: expert search. Find comments that friends did to posts with specific hashtag.
- **Q13**: single shortest path between two persons via Knows relationship
- **Q14**: weighted path between two persons; weight corresponds to amount of exchanged comments/replies

Update Queries

- **U1**: add a person to the social network
- **U2**: add friendship between two users
- **U3**: add aforum to the social network
- **U4**: add a forum membership for a given user
- **U5**: add a post to the social network
- **U6**: add a like to the given post of the network
- **U7**: add a comment replying to a given post/comment
- **U8**: add a like to a comment

2.3 Update Ratios

There are two ways to derive the frequency of updates in the SNB use case:

- From the *vertical domain* point of view: most of the user actions in the network are reads. Consider, for example, sending messages. There are several page impressions involved, e.g. *send message*, *come back to wall*, *clicking on the reply*, *clicking on the profile of replier* etc. An ad-hoc estimate is that only 1 in 5 impressions involves writing, while 4 in 5 are essentially read requests. On the other hand, an average user is likely to perform just very few write operations per day, spending most of the time online browsing and reading, so the total write operation share is likely to be just few percents.
- From the *DBMS perspective*, writes are extremely cheap, while our read queries (from the SNB Interactive workload) are very expensive. In other words, it is possible to execute tens of 1000s write operations per second, while only tens of read operations per second (the exact time varies per query).

We see that the two distributions (derived from the vertical domain and the technical considerations) can not be satisfied together. We propose a compromise workload, where the read queries from the SNB workload are balanced with both write operations and *short read queries*. To distinguish between SNB workload queries and *short reads* that we propose here, we will call the former *long read queries*. The two types of short read queries are the following:

- Profile view: for a given user ID returns basic information from her profile (name, city, age), and the list of at most 20 friends. This in turn boils down to three simple lookups:
  - **S1**: get the first and last names, birthday, IP address, browser, ID of the place for the given user ID
  - **S2**: get the last 10 posts created by the given user.
  - **S3**: retrieve all friends of the person
- Message view: for a given message ID return basic stats (when was it sent?) and some information about the sender (name etc.). Namely, there is a set of four queries:
  - **S4**: get the content for a given message ID
  - **S5**: get the first and last names of the creator of the message
  - **S6**: get the forum where this message appears (with the name of the forum moderator)
  - **S7**: get the replies to this message

All these short queries are highly interactive, we expect to be able to execute 1000s of them per second on a reasonably efficient system. For this reason the special Parameter Curation procedure introduced in Deliverable 2.2.4 is not needed here: the fact that we execute many of such queries means that the differences between runtimes *on average* would even out. For the same reason we will not distinguish between these two types when calculating the frequency of short queries: the actual workload will contain a randomly selected mix of the two types, such that their cumulative frequency follows a specified distribution

Our target workload, derived to suite both domain and technical requirements, is as follows: 10% of total runtime take update queries (taken from the data generator), more than 50% of time take long read queries (taken from the SNB set of queries), and less than 40% is given to the short read queries described above. This distribution is a compromise between two requirements in a sense that it contains an unusually large share of long read queries, while still keeping the update rate quite low.

2.3.1 Workload Composition

In order to compose the workload from the operations of the three described types (long reads, short reads, updates), the following principles are applied:

- Updates are taken from the data generator stream of events. We interpret the last n% of the data as updates, and play them out against the (bulk-loaded) rest of the social network (where n is a parameter given to the data generator)
- Read queries are introduced at a specified rate per update query. Each read query returns a set of users, posts or comments
- Starting from the set of users/posts returned by read queries, we perform a random walk consisting of short read queries (view a profile, view a message). At each step, a random message or user is selected, then we "click" (perform a corresponding read query) this item, and select one of the friends of the user (sender of the message, respectively). This way we fill out the quota of the short read queries. The random walk has two parameters:
  - $InitP$: the probability with which the first item in the walk is selected (e.g., 50% chance to pick elements from the return set of the read query)
  - $StepP$: with every step we decrease the selection chance by this value (e.g., if we start with 50%, the walk continues with selecting 40%, then 30% etc.)

  The random walk is therefore always finite, since at some point (typically after just a few iteration), the probability of selecting an item becomes 0, and we stop.

This workload mimics the user behavior online, where the news feed requests are interleaved with profile and message views.

2.3.2 Defining the Query Mix

Finally, we need to experimentally derive the mix of queries that would approximate our desired target workload composition. Experiments described here were performed on LDBC dataset Scale Factor 30 (roughly 30 GB of CSV files), using Virtuoso 7 Column Store as a DBMS.

As we have seen in 2.3.1, the long read queries are introduced at a specific rate per update query, which has to be defined. Additionally, the amount of short read queries (and their share in the workload) depends on two parameters, $InitP$ and $StepP$. Generally, decreasing these two values leads to smaller amount of short read queries (and therefore more updates and long reads).

First, the relative frequencies of the long reads is defined so that each query gets an equal share of CPU time. This will avoid having one query that dominates the entire workload. For example, Query 2 on average takes 113 ms, and Query 3 – 895 ms, i.e. 8.75 times longer. Therefore, in the query mix Query 2 should appear

8.75 times more frequently than Query 3, so that in total the CPU time for both queries in the entire workload is the same. The query frequency is computed in a way that "heavier" queries are executed less frequently than "lighter" ones, and no query type dominates the workload and the total runtime. The following counts of queries satisfy the condition of (roughly) equal CPU time per query type:

Query	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Count	205	406	42	6271	179	18	9235	30445	19	45	3069	155	204	109

Table 2.1: Long Read Query distribution (number of update queries per one long read query)

By setting $InitP$ and $StepP$ values for the short query selection, we define the counts of the short queries in the workload. For example, with $InitP$ = 70% and $StepP$ = 10%, we get the following counts of short read queries S1 - S7 in the workload

Query	S1	S2	S3	S4	S5	S6	S7
Count	453082	451767	452618	515092	515760	513779	514509

Table 2.2: Short Read query counts

In total, setting $InitP = 70\%$ and $StepP = 10\%$ result in the following runtime distribution for the three different classes of queries (long reads, short reads, updates):

Query Type	Long Reads	Short Reads	Updates
Total Time (ms)	205572	7615851	28667
Share of Time	2%	97%	0.3%

Table 2.3: Total runtime and share of runtime for Long Reads, Short Reads and Updates with $InitP = 70\%$

As we see, this distribution is far from desired target workload. In particular, the update ratio is negligible, so the workload stops being transactional. This happened because the original probabilities for short read queries were too high, so we ended up having too many of them. Table 2.4 displays the workload query distribution when the queries are generated with different (decreasing) values of $InitP$ and $StepP$.

We stop decreasing $InitP$ as soon as the runtime proportion of Long Reads exceeds the Short Reads (and thus the former occupy more than 50% of the workload runtime); at the same time, the total update runtime increases to desired 10%. Finally, based on runtime distributions of long queries, we compute their frequencies per update query, presented in Table 2.1. The goal is again to have an equal share of CPU time per query type,

To summarize, our workload query distribution is constructed as follows:

- updates are taken from the data generator, and are therefore the unit for measuring frequency of other queries
- we set $InitP = 30\%$ and $StepP = 10\%$ when generating Short Read queries, the total runtime of these queries will therefore be around 34%
- we set the frequency of Long Reads per one Update as specified in Table 2.1

2.3.3 Using Materialized Views

Real-world systems dealing with social network data are very likely to use materialized views (i.e., to precompute some answers) for interactive queries. It is also the case that materialized views are going to speed up certain Long Read queries from the SNB workload, for example, Query 2 (*recent posts and comments by user's friends*): having precomputed information on recent posts/comments of users will make query execution much faster.

	Long Reads Share	Short Reads Share	Updates Share
$InitP = 70\%$, $StepP = 10\%$	2%	97%	0.3%
$InitP = 50\%$, $StepP = 10\%$	22%	73.7%	4.3%
$InitP = 40\%$, $StepP = 10\%$	39%	54%	6.1%
$InitP = 30\%$, $StepP = 10\%$	57%	33.8%	9.1%

Table 2.4: Different probabilities $InitP$ and $StepP$ lead to different workload composition

On the other hand, a system that runs the complete SNB workload (long reads, short reads, updates) and uses materialized views has to incorporate updates into them, thus paying the price for improved processing of Long Reads. For some queries, materialized views are therefore becoming too expensive: a single update can touch tens of thousands users within two-step friendship environment (i.e. friends of friends), since the network is dense and small-world. Queries touching two steps in the friendship graph would therefore have to update thousands of counts per single update. In case of Query 5(*groups that friends and friends of friends of a given user have joined lately*), for example, that gets 179 updates per one query invocation (see Table 2.1) this excludes using materialized views altogether.

If we are to encourage selective usage of materialized views for some of SNB queries, we have to make them more frequent in comparison to updates. If Query 2, for instance, gets 5 times less updates than what we defined in Table 2.1, this will explicitly encourage systems to keep materialized views for that query. This will not change the entire workload distribution, that is long reads will still get 57% of the workload runtime, but within that part Query 2 will benefit a lot from (updatable) precomputation.

2.3.4 Running the Workload

This section reports our experiments with the LDBC Interactive workload (containing updates, long reads and short reads) described earlier in this chapter. We run our experiments on 24 core machine with 190 Gb RAM, using Virtuoso 7 Column Store. We use the dataset generated with the scale factor 30 (30 Gb of CSV data).

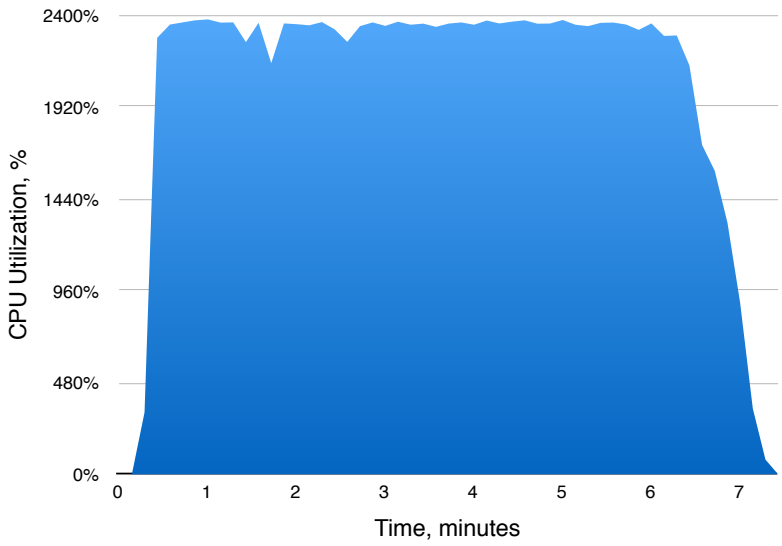

Figure 2.1: CPU Utilization

We have run the query workload generated with $InitP = 70\%$, $StepP = 10\%$ and Long Read frequencies from Table 2.1 and measured three important characteristics during the run:

- CPU utilization, given in Figure 2.1. Since the machine has 24 cores, the workload almost reaches full CPU utilization (2400%)
- Amount of cache misses (in proportion of every row touch), depicted in Figure 2.2. We have traced cache misses for three indexes: *likes* column of *likes* table (denoted *likes.likes*), as well as *post.ps_forumid* and *forum_person.forum_person* . In the beginning of the experiment the cache is cold, so each touched tuple causes several cache misses. As the workload runs, the cache warms up and there are fewer and fewer cache misses. Note that the whole dataset is never in memory, so there are still misses at the end of the run. On the other hand, some indexes (primary keys, not depicted) are always in memory so the amount of cache misses is always 0.

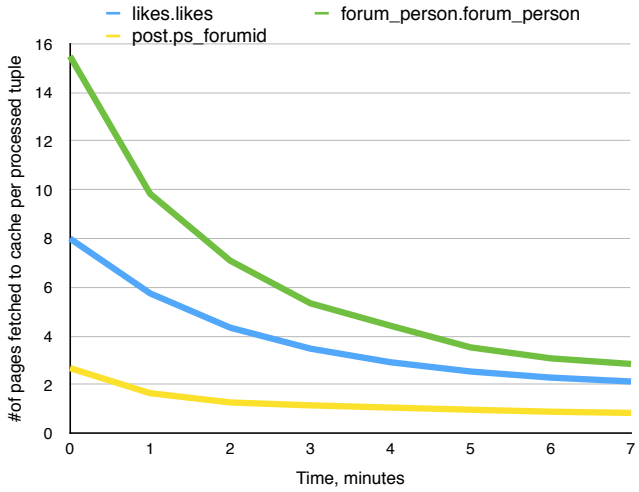

Figure 2.2: Percentage of cache misses over touches

- Disk reads (cumulative) as the run proceeds, presented in Figure 2.3. Similary to the previous case, disk reads decrease as the cache gets warmer.

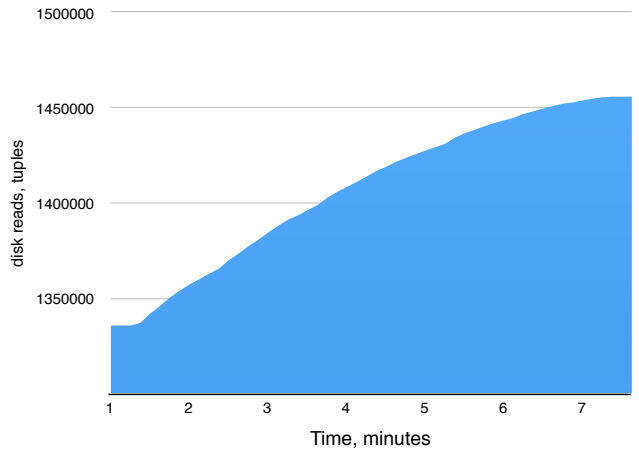

Figure 2.3: Disk reads (cumulative), in number of records fetched from disk

3 WORKLOAD DRIVER

In the previous chapter we described the Interactive workload of the LDBC SNB. In this chapter we consider a problem of running this workload in parallel, i.e. the design of the workload driver that will issue the queries against the System Under Test. This document describes design principles that were implemented for the LDBC Social Network Benchmark workload generator (*a driver*). The software for the driver (together with the detailed documentation on how to run it) is available on Github: `https://github.com/ldbc/ldbc_driver/`. Two reference implementations by two vendors are also on Github: `https://github.com/ldbc/ldbc_snb_implementations`. Description of the schema and generated data properties is given in the SNB task force report [1] Here we concentrate on key decisions and techniques that were developed for the scalable, repeatable, distributed workload driver.

3.1 Background and Motivation

The driver generates a stream of operations (e.g. create user, create post, create comment, retrieve person's posts etc.) and then executes them using the provided database connector. To be capable of generating heavier loads, it executes the operations from this stream in parallel. If there were no dependencies between operations (e.g., reads that depend on the completion of writes) this would be trivial. This is the case, for example, for the classical TPC-C benchmark, where splitting transaction stream into parallel clients (terminals) is trivial. However, for LDBC SNB Interactive Workload this is not the case: some operations within the stream do depend on others, others are depended on, some both depend on others and are depended on, and some neither depend on others nor are they depended on.

Consider, for example, a Social Network Benchmark scenario, where the data generator outputs a sequence of events such as *User A posted a picture*, *User B left a comment to the picture of User A*, etc. The second event depends on the first one in a sense that there is a causal ordering between them: User B can only leave a comment on the picture once it has been posted. The generated events are already ordered by their time stamp, so in case of the single-threaded execution this ordering is observed by default: the driver issues a request to the SUT with the first event (i.e., User A posts a picture), after its completion it issues the second event (create a comment). However, if events are executed in parallel, these two events may end up in different parallel sequences of events. Therefore, a driver needs a mechanism to ensure the dependency is observed even when the dependant events are in different parallel update streams.

All operations are time-stamped, giving a partial ordering (partial because some operations may have equal time stamps), and as long as the time stamps of dependent operations are never equal this partial ordering captures the dependencies between operations.

Even with a known ordering, executing dependent operations in a parallel (and distributed) context, in a way that is scalable (i.e., with more resources the driver can generate higher throughput) is not trivial. Benchmarks where queries do not introduce dependencies (like TPC-C), follow the simple partitioning approach, where the load is split across multiple drivers (emulating terminals). In case of the SNB, workload partitioning boils down to friendship graph partitioning – an inherently hard problem.

Furthermore, it is equally important that no unwanted bursty behavior is introduced by the driver. To maintain a deterministic execution order in distributed environments requires communication and synchronization between remote processes. This synchronization takes time and, especially if the duration between start times of dependent operations is short, can lead to the driver processes behaving in a bursty manner, i.e., blocking for periods, executing for periods, blocking again, etc.

For example, social networks tend to have a time-varying activity of users: certain concepts or topics cause flashmob effects, when a large amount of users posts (reposts, replies, shares) about a certain phenomena (e.g. an earthquake). In such a case the dataset contains a lot of dependant events, so the synchronization overhead would significantly alter the driver (and SUT) behavior. From the benchmarking point of view, it means that the driver would alter the workload in an unpredictable (and non-repeatable) way.

[1] `http://www.ldbc.eu:8090/download/attachments/4325436/LDBC_SNB_Report_Nov2013.pdf`

The following sections summarize the approaches used in the driver to deal with these challenges.

3.2 Definitions

Definitions for the various terms that will be introduced throughout the document are listed below:

- **Driver**: distributed, parallel workload generating software
  - **Driver Coordinator**: single, centralized, coordinating driver process. Responsible for coordinating the, potentially multiple, driver workers
  - **Driver Worker**: many distributed driver worker processes may exist. Responsible for executing the benchmark workload, i.e., sending queries to the System Under Test (SUT)
- **Simulation Time (ST)**: notion of time created by data generator. All time stamps in the generated data set are in simulation time
- **Real Time (RT)**: wall clock time
- **Time Compression Ratio**: function that maps simulation time to real time, e.g., an offset in combination with a compression ratio. It is a static value, set in driver configuration. Real Time Ratio is reported along with benchmark results, allowing others to recreate the same benchmark
- **Operation**: read and/or write
- **Dependencies**: operations in this set introduce dependencies in the workload. That is, for every operation in this set there exists at least one other operation (in **Dependents**) that can not be executed until this operation has been processed
- **Dependents**: operations in this set are dependent on at least one other operation (in **Dependencies**) in the workload
- **Due Time (DueT)**: point in simulation time at which the execution of an operation should be initiated.
- **Dependent Time (DepT)**: in addition to **Due Time**, every operation in **Dependents** also has a Dependent Time, which corresponds to the Due Time of the operation that it depends on. Dependent Time is always before Due Time. For operations with multiple dependencies Dependent Time is the maximum Due Time of all the operations it depends on.
- **Safe Time (SafeT)**: time duration.
  - when two operations have a necessary order in time (i.e., dependency) there is at least a SafeT interval between them
  - SafeT is the minimum duration between the Dependency Time and Due Time of any operations in Dependents
- **Operation Stream**: sequence of operations ordered by Due Time (dependent operations must separated by at least SafeT)
- **Initiated Operations**: operations that have started executing but not yet finished
- **Local Completion Time** (per driver): point in simulation time behind which there are no uncompleted operations

$$\text{Local Completion Time} = min(min(\text{Initiated Operations}), max(\text{Completed Operations}))$$

- **Global Completion Time (GCT)**: minimum completion time of all drivers. Once GCT has advanced to the Dependent Time of some operation that operation is safe to execute, i.e., the operations it depends on have all completed executing.

$$\text{Global Completion Time} = min(\text{Local Completion Time})$$

- **Execution Window (Window)**: a timespan within which all operations can be safely executed
  - All operations satisfying *window.startTime* <= *operation.DueT* < *window.endTime* may be executed
  - Within a window no restrictions on operation ordering or operation execution time are enforced, driver has a freedom of choosing an arbitrary scheduling strategy inside the window
  - To ensure that execution order respects dependencies between operations, window size is bounded by SafeT, such that: 0 < *window.duration* <= *SafeT*
  - Window duration is fixed, per operation stream; this is to simplify scheduling and make benchmark runs repeatable
  - Before any operations within a window can start executing it is required that:
    GCT >= *window.startTime - (SafeT - window.duration)*
  - All operations within a window must initiate and complete between window start and end times:
    window.startTime <= *operation.initiate* < *window.endTime* and
    window.startTime <= *operation.complete* < *window.endTime*
- **Dependency Mode**: defines dependencies, constraints on operation execution order
- **Execution Mode**: defines how the runtime should execute operations of a given type

3.3 Tracking Dependencies

Consider that every operation in a workload belongs to none, one, or both of the following sets: Dependencies and Dependents.

As mentioned, the driver uses operation time stamps (Due Times) to ensure that dependencies are maintained. It keeps track of the latest point in time behind which every operation has completed. That is, every operation (i.e., dependency) with a Due Time lower or equal to this time is guaranteed to have completed execution. It does this by maintaining a monotonically increasing (distributed) variable called Global Completion Time (GCT).

Logically, every time the driver (via a database connector) begins execution of an operation from Dependencies that operation is added to Initiated Operations: the set of operations that have started executing but not yet finished. Then, upon completion, the operation is removed from Initiated Operations and added to Completed Operations: the set of operations that have started and finished executing. Using these sets, each driver process maintains its own view of GCT in the following way. Local progress is monitored and managed using a variable called Local Completion Time (LCT): the point in time behind which there are no uncompleted operations, i.e., no operation in Initiated Operations has a lower or equal Due Time and no operation in Completed Operations has an equal or higher Due Time (see Definitions for specifics). LCT is periodically sent to all other driver processes, which all then (locally) set their view of GCT to the minimum LCT of all driver processes.

At this point the driver has two, of the necessary three (third covered shortly), pieces of information required for knowing when to execute an operation:

- **Due Time**: point in time at which an operation should be executed, assuming all preconditions (e.g., dependencies) have been fulfilled
- **GCT**: every operation (from Dependencies) with a Due Time before this point in time has completed execution

However, with only GCT to track dependencies the driver has no way of knowing when it is safe to execute any particular dependent operation.

What GCT communicates is that *all* dependencies up to some point in time have completed, but whether or not the dependencies for any particular operation are within these completed operations is unknown. The driver would have to wait until GCT has *passed* the Due Time (because Dependency Time is always lower) of an operation before that operation could be safely executed, which would result in the undesirable outcome of every operation missing its Due Time.

The required information is which particular operation in Dependencies does any operation in Dependents depend on. More specifically, the Due Time of this operation. This is referred to as **Dependent Time**: in addition to Due Time, every operation in Dependents also has (read: must have) a Dependent Time, which corresponds to the latest Due Time of all the operations it depends on. Once GCT has advanced beyond the Dependent Time of an operation that operation is safe to execute.

Using these three mechanisms (**Due Time**, **GCT**, and **Dependent Time**) the driver is able to execute operations, while ensuring their dependencies are satisfied beforehand.

3.4 Scalability in the Presence of Dependencies

The mechanisms introduced in Section 3.3 guarantee that dependency constraints are not violated, but in doing so they unavoidably introduce overhead of communication/synchronization between driver threads/processes. To minimize the negative effects that synchronization has on driver scalability an additional Execution Mode was introduced (for more on Execution Modes see Section 3.5): **Windowed Execution**.

Windowed Execution has two design goals: (a) make the generated load less 'bursty' (b) allow the driver to 'scale', so when the driver is given more resources (CPUs, servers, etc.) it is able to generate more load.

In the context of Windowed Execution, operations are executed in groups (**Windows**), where operations are grouped according to their Due Time. Every Window has a *Start Time*, a *Duration*, and an *End Time*, and Windows contain only those operations that have a Due Time between *Window.startTime* and *Window.endTime*. Logically, all operations within a Window are executed at the same time, *some* time within the Window. No guaranty is made regarding exactly when, or in what order, an operation will execute within its Window.

The reasons this approach is correct are as follows:

- Operations belonging to the Dependencies set are never executed in this manner - the Due Times of Dependencies operations are never modified as this would affect how dependencies are tracked
- The minimum duration between the Dependency Time and Due Time of any operation in Dependents is known (can be calculated by scanning through workload once), this duration is referred to as Safe Time (SafeT)
- A window does not start executing until the dependencies of all its operations have been fulfilled. This is ensured by enforcing that window execution does not start until
  GCT >= window.startTime - (SafeT - window.duration) = window.endTime - SafeT;
  that is, the duration between GCT and the end of the window is no longer than SafeT

The way this translates into improved scalability is it can drastically reduce the amount of communication/synchronization necessary for tracking and maintaining GCT. Instead of reading GCT before the execution of every operation, it only needs to be read before the execution of every window. Then, as GCT is read less frequently, it follows that GCT does not need to be communicated between driver processes as frequently. Communicating GCT at some fraction of *SafeT* (e.g., *SafeT/3*) is likely sufficient.

The advantages of such an execution mode are as follows:

- As no guarantees are made regarding time or order of operation execution within a Window, GCT no longer needs to be read before the execution of every operation, only before the execution of every window

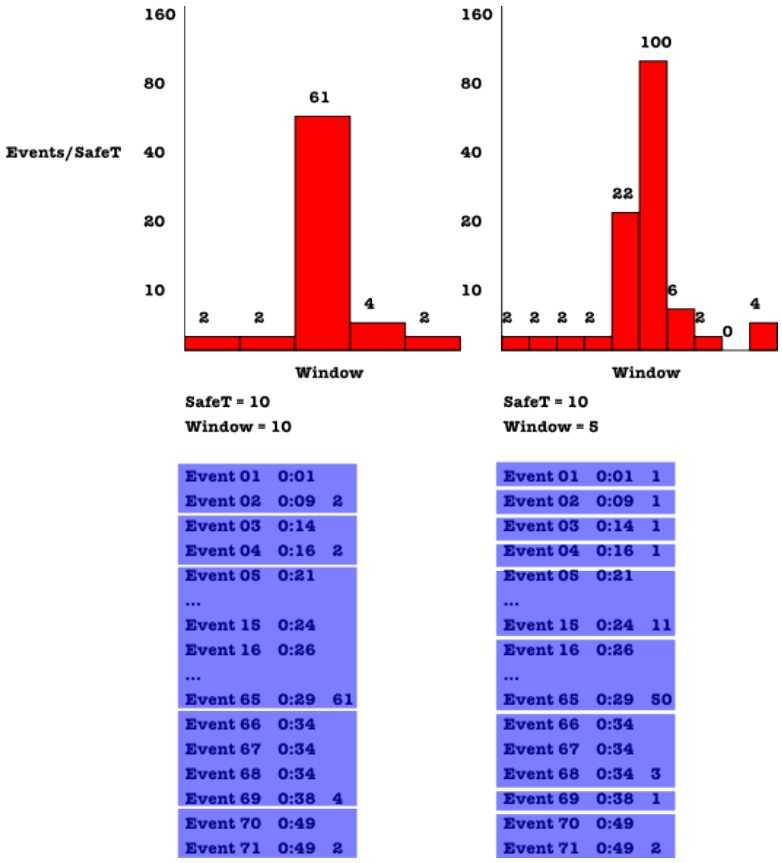

Figure 3.1: Workload Execution With Different Resolution (Window Size)

- Then, as GCT is read less frequently, it follows that it does not need to be communicated between driver processes as frequently. There is no need or benefit to communicating GCT protocol message more frequently than approximately Window.duration, the side effect of which is reduced network traffic
- Further, by making no guarantees regarding the order of execution the driver is free to reschedule operations (within Window bounds). The advantage being that operations can be rearranged in such a way as to reduce unwanted bursts of load during execution, which could otherwise occur while synchronizing GCT during demanding workloads. For example, a uniform scheduler may modify operation Due Times to be uniformly distributed across the Window timespan, to 'smoothen' the load within a Window.

As with any system, there are tradeoffs to this design, particularly regarding *Window.duration*. The main trade-off is that between 'workload resolution' and scalability. Increasing *Window.duration* reduces communication/synchronization but also reduces the resolution at which the workload definition is followed. That is, the generated workload becomes less like the workload definition. However, as this is both bounded and configurable, it is not a major concern. This issue is illustrated in Figure 3.1, where the same stream of events is split into two different workloads based on different size of the Window. The workload with Window size 5 (on the right) has better resolution, especially for the "bursty" part of the event stream.

This design also trades a small amount of repeatability for scalability: as there are no timing or ordering guarantees within a window, two executions of the same window are not guaranteed to be equivalent - 'what happens in the window stays in the window'. Despite sacrificing this repeatability, the results of operations do not change. No dependency-altering operations occur during the execution of a Window, therefore results for all queries should be equivalent between two executions of the same workload, there is no effect on the expected result for any given operation.

3.5 Workload Execution

Using the information and mechanisms introduced in previous sections, in addition to a little additional context where necessary, we can now explain precisely how operations are executed.

Based on the dependencies certain operations have, and on the granularity of parallelism we wish to achieve while executing them, we assign a Dependency Mode and an Execution Mode to every operation type. Using these classifications the driver runtime then knows how each operation should be executed. These modes, as well as what they mean to the driver runtime, are described below.

3.5.1 Dependency Modes

While executing a workload the driver treats operations differently, depending on their **Dependency Mode**. In the previous section operations were categorized by whether or not they are in the sets Dependencies and/or Dependents.

Another way of communicating the same categorization is by assigning a Dependency Mode to operations - every operation type generated by a workload definition must be assigned to exactly one Dependency Mode. Dependency modes define dependencies, constraints on operation execution order. The driver supports a number of different **Dependency Modes**: *None*, *Read Only*, *Write Only*, *Read Write*. During workload execution, operations of each type are treated as follows:

- **None**
  Depended On (NO): operations do not introduce dependencies with other operations (i.e., the correct execution of no other operation depends on these operations to have completed executing)
  - **Prior Execution**: do nothing
  - **After Execution**: do nothing
- **Read Only**
  Depended On (NO): operations do not introduce dependencies with other operations (i.e., the correct execution of no other operation depends on these operations to have completed executing)
  Dependent On (YES): operation execution does depend on GCT to have advanced sufficiently (i.e., correct execution of these operations requires that certain operations have completed execution)
  - **Prior Execution**: wait for *GCT >= operation.DepTime*
  - **After Execution**: do nothing
- **Write Only**
  Depended On (YES): operations do introduce dependencies with other operations (i.e., the correct execution of certain other operations requires that these operations to have completed executing, i.e., to advance GCT)
  Dependent On (NO): operation execution does not depend on GCT to have advanced sufficiently (i.e., correct execution of these operations does not depend on any other operations to have completed execution)
  - **Prior Execution**: add operation to *Initiated Operations*
  - **After Execution**: remove operation from *Initiated Operations*, add operation to *Completed Operations*
- **Read Write**
  Depended On (YES): operations do introduce dependencies with other operations (i.e., the correct execution of certain other operations requires that these operations to have completed executing, i.e., to advance GCT)
  Dependent On (YES): operation execution does depend on GCT to have advanced sufficiently (i.e., correct execution of these operations requires that certain operations have completed execution)

- **Prior Execution**: add operation to *Initiated Operations*, wait for *GCT < operation.DepT*
- **After Execution**: remove operation from *Initiated Operations*, add operation to *Completed Operations*

3.5.2 Execution Modes

Execution Modes relate to how operations are scheduled, when they are executed, and what their failure conditions are.

Each operation type in a workload definition must be assigned to exactly one Execution Mode. The driver supports a number of different Execution Modes: *Asynchronous*, *Synchronous*, *Partially Synchronous*. It splits the single operation stream generated by a workload into multiple streams, one per Execution Mode. During workload execution, operations from each of these streams are treated as follows.

- **Asynchronous**: operations are executed individually, when their Due Time arrives.

  Motivation: This is the default execution mode, it executes operations as true to the workload definition as possible.

  - **Re-scheduling Before Execution**: None: *operation.DueT* not modified by scheduler
  - **Execute When** *time >= operation.DueT (and GCT >= operation.DepT)*
  - **Max Concurrent Executions**: unbounded
  - **Max Execution Time**: unbounded
  - **Failure**: operation execution starts later than: *operation.DueT + Tolerated Delay*

- **Synchronous**: operations are executed individually, sequentially, in blocking manner.

  Motivation: Some dependencies are difficult to capture efficiently with SafeT and GCT alone. For example, social applications often support conversations via posts and likes, where likes depend on the existence of posts. Furthermore, posts and likes also depend on the existence of the users that make them. However, users are created at a lower frequency than posts and likes, and it can be assumed they do not immediately start creating content. As such, a reasonably long *SafeT* can be used between the creation of a user and the first time that user creates posts or likes. Conversely, posts are often replied to and/or liked soon after their creation, meaning a short SafeT would be necessary to maintain the ordering dependency. Consequently, maintaining the dependencies related to conversations would require a short SafeT, and hence a small window. This results in windows containing fewer operations, leading to less potential for parallelism within windows, less freedom in scheduling, more synchronization, and greater likelihood of bursty behavior - all negative things.

  The alternative offered by **Synchronous Execution** is that, when practical, operations of certain types can be partitioned (e.g. posts and likes could be partitioned by the forum in which they appear), and partitions assigned to driver processes. Using the social application example from above, if all posts and likes were partitioned by forum the driver process that executes the operations from any partition could simply execute them sequentially. Then the only dependency to maintain would be on user operations, reducing synchronization dramatically, and parallelism could still be achieved as each partition would be executed independently, in parallel, by a different driver process.

  - **Re-scheduling Before Execution**: None: *operation.DueT* not modified by scheduler
  - **Execute When** *time >= operation.DueT and previousOperation.completed == true (and GCT >= operation.DepT)*
  - **Max Concurrent Executions**: 1
  - **Max Execution Time**: *nextOperation.DueT - operation.DueT*

  - **Failure**: operation execution starts later than: *operation.DueT + Tolerated Delay* E.g., if *previousOperation* did not complete in time, forcing current operation to wait for longer than *toleratedDelay*

- **Partially Synchronous** (**Windowed Execution**, described in Section 3.4 in more details), groups of operations from the same time window are executed together

  - **Re-scheduling Before Execution**: Yes, as long as the following still holds:
    window.startTime <= operation.DueT < window.startTime + window.duration
    Operations within a window may be scheduled in any way, as long as they remain in the window from which they originated: their *Due Times*, and therefore ordering, may be modified
  - **Execute When** *time >= operation.DueT (and GCT >= operation.DepT)*
  - **Max Concurrent Executions**: number of operations within window
  - **Max Execution Time**: *(window.startTime + window.duration) - operation.DueT*
  - **Failure**: operation execution starts later than: *window.startTime + window.duration*
    operation execution does not finish by: *window.startTime + window.duration*

3.6 Measurements

3.6.1 Measuring Latency

There are essentially two ways of reporting measurements for a transactional workload:

- try to get the maximum throughput, and report that number.
- fix the load at a certain level (say, 100K transactions per second), and run the workload reporting *latency-at-load*.

Following the TPC-C example, LDBC SNB Interactive workload is run by fixing the certain load (by adjusting the simulation time to real time ratio in the generated social network).

When measuring system performance the objective is to gain insight into how that system would perform in a known, realistic scenario. Data obtained from such testing is useful for capacity planning, setting of service level agreements (SLA), software selection, etc. In all cases the desired information is: how quickly (latency) will the system under test (SUT) respond to a request, when subjected to that scenario. Knowing the expected latency for an SUT under some given load is almost always more valuable than knowing the maximum throughput that SUT could serve.

Given that *latency-at-load* is the metric of interest, the testing methodology and tooling must be designed with this in mind. That is, in addition to the types and numbers of queries to execute as part of a workload, the load tester (LT) must also control the frequency at which those queries are sent to the SUT. Specifying frequencies allows an LT to measure how an SUT performs under some well defined load, it shifts the emphasis from measuring maximum throughput to measuring response times at a given throughput.

Testing in this manner requires a number of concepts to be understood and challenges overcome, specifically those related to measurement errors like *coordinated omission*, and to ensuring that test runs are repeatable and comparable. The following sections provide a summary of these topics, providing motivation for certain design decisions taken during development of the workload driver, e.g., the rationale for assigning operations Due Times and only executing them at those Due Times, rather than as fast as the database would allow possible.

3.6.2 Repeatability

In the context of load generation, repeatability means the loads generated by different test runs, all of which use the same workload definition, should be equal (within some bounds).

There are several reasons that make it desirable for test runs to be comparable. It makes it possible to compare the performance of one SUT to another, e.g., if the test results for one SUT differ from those of another, and it is known that both results under the same workload, it is safe to assume that any differences in the results are due to SUT differences. Another, perhaps more important, reason is it enables test results to be validated, e.g., by third parties.

To be capable of producing repeatable test runs an LT must fulfill various requirements. Generally, the LT must adhere to the workload definition, given the same workload definition the LT will generate the same load. To do that the LT must be aware of (i.e., monitoring) the load that it is generating. Given this knowledge an LT can ensure the load it generates is independent of both SUT (see section 3.6.3 on Coordinated Omission) and hardware. Regardless of the hardware on which an LT runs, the load it generates should be the same: running on faster hardware does not result in the generation of a heavier load and running on slower hardware does not result in a lighter load. To ensure that test runs are repeatable, an LT must detect if the load it generates deviates from the workload definition and act accordingly if it does. E.g., if the hardware on which the LT runs does not provide sufficient resources for it to generate the defined workload the LT should recognize this, terminate the test, and report an appropriate error message.

An LT must guaranty that multiple test executions using the same workload definition will generate the same workload: same query types, same query counts, same query frequencies, in the same order - across executions, the same things happen at the same relative times.

3.6.3 Coordinated Omission

Coordinated omission (CO) is a measurement methodology problem that applies to performance measuring systems (e.g. an LT), rather than to an SUT. It is a common problem, leading to such errors as the incorrect reporting of (especially higher) percentile data.

The problem occurs when an observed response time is longer than the interval that the LT is trying to maintain between requests; when an LT that waits for a response before issuing the next request misses the time window in which it would have sent that next request, measurements are omitted. Whenever the blocking of a thread in an LT is somehow coordinated with an SUT event, CO is likely to occur. The term 'coordinated' is used in the sense that the LT pauses at the same time the SUT experiences heavy load and, with high probability, the SUT would take longer than usual to respond to other requests if the LT was able to (was not paused) send them at that time. This equates to the LT allowing the SUT to say 'I'm heavily loaded right now, please let me catch up before sending more requests'. In effect, the LT omits slow responses that would likely occur if it sent the requests as intended, resulting in incorrect higher percentile measurements because the calculations do not include the slow responses that were 'omitted'.

The outcome is analogous to removing a large percentage of slow responses from the results, leaving almost only faster responses (typically, only the first slow response, in each sequence of slow responses, survives the removal). This skews statistics (e.g. percentiles) that are calculated from the recorded results. Percentiles, for example, can be orders of magnitude off, or the reported 99.9th percentile may appear better than the real 99th percentile.

Consider the following scenario. A test is created to measure the performance of an SUT under real world conditions, and the metric deemed most important is 90th percentile latency. In real world conditions the SUT would receive 1 request every 1 s, so the LT is configured to do the same. The LT repeats the following until the end of the workload, then reports response times as percentiles: send request to SUT, wait for response from SUT, measure response time. Note that while waiting for a response the LT does nothing, it sends no new requests during this time.

Normally the SUT responds in 900 ms or less, but every 900 s it takes as long as 100 s due to the overhead of some periodic system maintenance (e.g. garbage collection). Given this scenario, every 1000 s period the LT would receive 900 x 900 ms responses and 1 x 100 s response, and it would report a 90th percentile of 900 ms. This report is erroneous. Consider the following: during the 100 s pause SUT performance appears to be markedly ($\sim$100 times) worse; during the first 900 s of every 1000 s period 900 responses are recorded, but during the last 100 s only 1 is recorded; 10% of the time (when SUT performance is at its worst) is represented

by only ∼0.1% (1/901) of the recorded results. Had the LT continued sending requests every 1 s during the 100 s slow period, results for each 1000 s period would likely contain 900 x 900 ms responses and 100 x 100 s responses, translating to a 90th percentile of 100 s, 2 orders of magnitude higher that what was actually reported.

Observe, these results are clearly invalid, but it is only because the omitted requests were due to pauses in the SUT that the cause of error is CO, i.e., the omissions are coordinated with SUT events. Had the LT omitted requests at random the results would likely have remained statistically valid.

3.7 Driver and Social Network Benchmark

LDBC Driver, described in this chapter, was designed to execute the workload of the Social Network Benchmark. As we have mentioned in Section 3.1, the main challenge of running queries in parallel on the graph-shaped data stem from dependencies introduced by the graph structure. In other words, workload partitioning becomes as hard as graph partitioning.

The LDBC SNB data can in fact be seen as a union of two parts:

- "core" – relatively small and dense friendship graph (not more than 10% of the data). Updates on this part are very hard to partition among driver threads, since the graph is essentially a single dense strongly connected component.
- user activity: posts, replies, likes; this is by far the biggest part of the data. Updates on this part are easily partitioned as long as the dependencies with the "core" part are satisfied (i.e., users don't post things before the profiles are created, etc.).

In order to avoid friendship graph partitioning, the LDBC Driver introduces the concept *SafeT* – the minimal simulation time that should pass between two dependant events. This property is enforced by the data generator, i.e. the driver does not need to change or delay some operations in order to guarantee dependency safety. Respecting dependencies now means globally communicating the advances of the *Global Completion Time*, and making sure the operations do not start earlier than *SafeT* from their dependants.

On the other hand, the driver exploits the fact that some of the dependencies in fact do not hinder partitioning: although replies to the post can only be sent after the post is created, these kinds of dependencies are satisfied if we partition workload by forums. This way, all (update) operations on posts and comments from one forum are assigned to one driver thread. Since there is typically a lot of forums, each driver thread gets multiple ones. Updates from one forum are then run in *Synchronous mode* (see Section 3.5), and parallelism is achieved by running many distinct forums in parallel. By doing so, we can add posts and replies to forums at very high frequency without the need to communicate the GCT across driver instances (i.e. we efficiently create the so-called flashmob effects in the posting/replying workload).

4 CONCLUSION

The deliverable describes the query mix of the transactional SNB workload. We have set the frequency distribution of operations to be 10% updates, more than 50% of long reads, and the rest (less than 40%) are short reads. Moreover, we have defined the frequency of updates and short reads via the distribution of long read running times. As a result, we have a complete definition of the workload, where each of the 14 SNB queries corresponds to a number of updates and short reads that have to be included in the operation stream. We have presented some runtime characteristics of the workload run measured on Virtuoso 7 Column Store.

For the LDBC transactional workload we have developed a workload driver that satisfies two properties: (i) it executes the workload as is, in a repeatable way for any SUT, (ii) it respects the dependencies between operations in the workload. The second part of the deliverable described main concepts and principles of the driver architecture.

A SET OF QUERIES

For completeness, we give the full text of query templates of the LDBC Social Network Benchmark Interactive workload in SPARQL and SQL (short reads). Implementations in other languages/APIs will be made available on Github: `https://github.com/ldbc/ldbc_snb_implementations`

A.1 Long Read Queries

Query 1: friends with a certain name

```
sparql SELECT ?fr ?last min(?dist) as ?mindist  ?bday
        ?since ?gen ?browser ?locationIP #Q1
    ((SELECT group_concat (?email, ", ")
        WHERE {
            ?frr snvoc:email ?email .
        FILTER (?frr = ?fr) .
        }
        group BY ?frr)) as ?email
    ((SELECT group_concat (?lng, ", ")
        WHERE {
            ?frr snvoc:speaks ?lng .
        FILTER (?frr = ?fr) .
        }
        group BY ?frr)) as ?lng
    ?based
    ((SELECT group_concat ( bif:concat (?o_name, " ", ?year, " ", ?o_country),
                ", ")
        WHERE {
        ?frr snvoc:studyAt ?w .
        ?w snvoc:classYear ?year .
            ?w snvoc:hasOrganisation ?org .
            ?org snvoc:isLocatedIn ?o_countryURI .
        ?o_countryURI foaf:name ?o_country .
            ?org foaf:name ?o_name .
        FILTER (?frr = ?fr) .
        }
        group BY ?frr)) as ?studyAt
    ((SELECT group_concat ( bif:concat (?o_name, " ", ?year, " ", ?o_country),
                ", ")
        WHERE {
        ?frr snvoc:workAt ?w .
        ?w snvoc:workFrom ?year .
            ?w snvoc:hasOrganisation ?org .
            ?org snvoc:isLocatedIn ?o_countryURI .
        ?o_countryURI foaf:name ?o_country .
            ?org foaf:name ?o_name .
        FILTER (?frr = ?fr) .
        }
        group BY ?frr)) as ?workAt
    {
        ?fr A snvoc:Person .
```

```
        ?fr snvoc:firstName "%Name%" .
    ?fr snvoc:lastName ?last .
        ?fr snvoc:birthday ?bday .
        ?fr snvoc:isLocatedIn ?basedURI .
    ?basedURI foaf:name ?based .
        ?fr snvoc:creationDate ?since .
        ?fr snvoc:gender ?gen .
    ?fr snvoc:locationIP ?locationIP .
    ?fr snvoc:browserUsed ?browser .

        {
          { SELECT DISTINCT ?fr (1 as ?dist)
            WHERE {
              sn:pers%Person% snvoc:knows ?fr.
            }
          }
      UNION
          { SELECT DISTINCT ?fr (2 as ?dist)
            WHERE {
              sn:pers%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr.
              FILTER (?fr != sn:pers%Person%).
            }
          }
      UNION
          { SELECT DISTINCT ?fr (3 as ?dist)
            WHERE {
              sn:pers%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr3.
              ?fr3 snvoc:knows ?fr. FILTER (?fr != sn:pers%Person%).
            }
          } .
        }
    }
    group BY ?fr ?last ?bday ?since ?gen ?browser ?locationIP ?based
    ORDER BY ?mindist ?last ?fr
    LIMIT 20
```

Query 2: Recent posts and comments by your friends

```
sparql SELECT ?fr ?first ?last ?post ?content ?date
FROM <sib>
WHERE {
  sn:pers%Person% snvoc:knows ?fr.
  ?fr snvoc:firstName ?first. ?fr snvoc:lastName ?last .
  ?post snvoc:hasCreator ?fr.
  { {?post snvoc:content ?content } UNION
  { ?post snvoc:imageFile ?content }} .
  ?post snvoc:creationDate ?date.
  FILTER (?date <= "%Date0%"^^xsd:date).
}
ORDER BY DESC (?date) ?post
LIMIT 20
```

Query 3: Friends and friends of friends that have been to countries X and Y

```
sparql SELECT ?fr ?first ?last ?ct1 ?ct2 (?ct1 + ?ct2) as ?sum
FROM <sib>
WHERE {
    {SELECT DISTINCT ?fr ?first ?last
        (((SELECT count (*)
        WHERE {
            ?post snvoc:hasCreator ?fr .
            ?post snvoc:creationDate ?date .
            FILTER (?date >= "%Date0%"^^xsd:date
             && ?date < bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date)) .
            ?post snvoc:isLocatedIn dbpedia:%Country1%
        }))
        as ?ct1)
        ((SELECT count (*)
        WHERE {
            ?post2 snvoc:hasCreator ?fr .
            ?post2 snvoc:creationDate ?date2 .
            FILTER (?date2 >= "%Date0%"^^xsd:date
             && ?date2 < bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date)) .
            ?post2 snvoc:isLocatedIn dbpedia:%Country2%
        })
        as ?ct2)
    WHERE {
        {sn:pers%Person% snvoc:knows ?fr.} UNION
        {sn:pers%Person% snvoc:knows ?fr2.
                ?fr2 snvoc:knows ?fr. FILTER (?fr != sn:pers%Person%)} .
        ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last .
        ?fr snvoc:isLocatedIn ?city .
        FILTER(!exists {?city snvoc:isPartOf dbpedia:%Country1%}).
        FILTER(!exists {?city snvoc:isPartOf dbpedia:%Country2%}).
    }
    }.
    FILTER (?ct1 > 0 && ?ct2 > 0) .
}
ORDER BY DESC(6) ?fr
LIMIT 20
```

Query 4: New topics

```
sparql SELECT ?tagname count (*)
FROM <sib>
WHERE {
    ?post A snvoc:Post .
    ?post snvoc:hasCreator ?fr .
    ?post snvoc:hasTag ?tag .
    ?tag foaf:name ?tagname .
    ?post snvoc:creationDate ?date .
    sn:pers%Person% snvoc:knows ?fr .
    FILTER (?date >= "%Date0%"^^xsd:date &&
     ?date <= bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date) ) .
    FILTER (!exists {
        sn:pers%Person% snvoc:knows ?fr2 .
```

```
        ?post2 snvoc:hasCreator ?fr2 .
        ?post2 snvoc:hasTag ?tag .
        ?post2 snvoc:creationDate ?date2 .
        FILTER (?date2 < "%Date0%"^^xsd:date)})
    }
group BY ?tagname
ORDER BY DESC(2) ?tagname
LIMIT 10
```

Query 5: New groups

```
sparql SELECT ?title count (*) #Q5
FROM <sib>
WHERE {
    {SELECT DISTINCT ?fr
     FROM <sib>
     WHERE {
       {sn:pers%Person% snvoc:knows ?fr.} UNION
       {sn:pers%Person% snvoc:knows ?fr2.
        ?fr2 snvoc:knows ?fr. FILTER (?fr != sn:pers%Person%)}
     }
    } .
    ?group snvoc:hasMember ?mem .
    ?mem snvoc:hasPerson ?fr .
    ?mem snvoc:joinDate ?date .
    FILTER (?date >= "%Date0%"^^xsd:date) .
    ?post snvoc:hasCreator ?fr .
    ?group snvoc:containerOf ?post .
    ?group snvoc:title ?title.
}
group BY ?title
ORDER BY DESC(2) ?title
LIMIT 20
```

Query 6: Tag co-occurrence

```
sparql SELECT ?tagname count (*)
FROM <sib>
WHERE {
    {  SELECT DISTINCT ?fr
       FROM <sib>
       WHERE {
           {sn:pers%Person% snvoc:knows ?fr.} UNION
           {sn:pers%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr.
                       FILTER (?fr != sn:pers%Person%)}
       }
    } .
    ?post A snvoc:Post .
    ?post snvoc:hasCreator ?fr .
    ?post snvoc:hasTag ?tag1 .
    ?tag1 foaf:name ?tagname1 .
    FILTER (?tagname1 != '%Tag%') .
    ?post snvoc:hasTag ?tag .
    ?tag foaf:name ?tagname .
```

```
}
group BY ?tagname
ORDER BY DESC(2) ?tagname
LIMIT 10
```

Query 7: Recent likes

```
sparql SELECT ?liker ?first ?last ?ldt
       (if ((exists {  sn:pers%Person% snvoc:knows ?liker}), 0, 1) as ?is_new)
       ?post ?content (bif:datediff ("minute", ?dt, ?ldt) as ?lag)
FROM <sib>
WHERE {
  ?post snvoc:hasCreator sn:pers%Person% .
  {{ ?post snvoc:content ?content } UNION
  {?post snvoc:imageFile ?content}} .
  ?lk snvoc:hasPost ?post .
  ?liker snvoc:likes ?lk . ?liker  snvoc:firstName ?first .
  ?liker snvoc:lastName ?last .
  ?post snvoc:creationDate ?dt . ?lk snvoc:creationDate ?ldt .
}
ORDER BY DESC (?ldt) ?liker
LIMIT 20
```

Query 8: Recent replies

```
sparql SELECT ?FROM ?first ?last ?dt ?rep ?content
WHERE {
  { SELECT ?rep ?dt
    WHERE {
        ?post snvoc:hasCreator sn:pers%Person% .
        ?rep snvoc:replyOf ?post . ?rep snvoc:creationDate ?dt .
    }
    ORDER BY DESC (?dt)
    LIMIT 20
  } .
  ?rep snvoc:hasCreator ?FROM .
  ?FROM snvoc:firstName ?first . ?FROM snvoc:lastName ?last .
  ?rep snvoc:content ?content.
}
ORDER BY DESC(?dt) ?rep
```

Query 9: Recent posts and comments by friends or friends of friends

```
sparql SELECT ?fr ?first ?last ?post ?content ?date
FROM <sib>
WHERE {
  {SELECT DISTINCT ?fr
   FROM <sib>
   WHERE {
       {sn:pers%Person% snvoc:knows ?fr.} UNION
       {sn:pers%Person% snvoc:knows ?fr2.
        ?fr2 snvoc:knows ?fr. FILTER (?fr != sn:pers%Person%)}
   }
  }
  ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last .
```

```
  ?post snvoc:hasCreator ?fr.
  ?post snvoc:creationDate ?date.
  FILTER (?date < "%Date0%"^^xsd:date).
  {{?post snvoc:content ?content} UNION
  {?post snvoc:imageFile ?content}} .
}
ORDER BY DESC (?date) ?post
LIMIT 20
```

Query 10: Friend recommendation

```
sparql SELECT ?first ?last
    ((( SELECT  count (DISTINCT ?post)
        WHERE {
            ?post snvoc:hasCreator ?fof .
            ?post snvoc:hasTag ?tag .
            sn:pers%Person% snvoc:hasInterest ?tag
        }
    ))
    -
    ((  SELECT  count (DISTINCT ?post)
        WHERE {
            ?post snvoc:hasCreator ?fof .
            ?post snvoc:hasTag ?tag .
            FILTER (!exists {sn:pers%Person% snvoc:hasInterest ?tag})
        }
    )) as ?score)
    ?fof  ?gender ?locationname
FROM <sib>
WHERE {
   {SELECT DISTINCT ?fof
    WHERE {
        sn:pers%Person% snvoc:knows ?fr .
        ?fr snvoc:knows ?fof .
        FILTER (?fof != sn:pers%Person%)
        minus { sn:pers%Person% snvoc:knows ?fof } .
    }
   } .
   ?fof snvoc:firstName ?first .
   ?fof snvoc:lastName ?last .
   ?fof snvoc:gender ?gender .
   ?fof snvoc:birthday ?bday .
   ?fof snvoc:isLocatedIn ?based .
   ?based foaf:name ?locationname .
   FILTER (1 = if (bif:month (?bday) = %HS0%,
         if (bif:dayofmonth (?bday) > 21, 1, 0),
         if (bif:month (?bday) = %HS1%,
         if (bif:dayofmonth(?bday) < 22, 1, 0), 0)))
}
ORDER BY DESC(3) ?fof
LIMIT 10
```

Query 11: Job referral

```
sparql SELECT ?first ?last ?startdate ?orgname ?fr
WHERE {
    ?w snvoc:hasOrganisation ?org .
    ?org foaf:name ?orgname .
    ?org snvoc:isLocatedIn ?country.
    ?country foaf:name '%Country%' .
    ?fr snvoc:workAt ?w .
    ?w snvoc:workFrom ?startdate .
    FILTER (?startdate < %Date0%) .
    {  SELECT DISTINCT ?fr
       FROM <sib>
       WHERE {
           {sn:pers%Person% snvoc:knows ?fr.} UNION
           {sn:pers%Person% snvoc:knows ?fr2.
                ?fr2 snvoc:knows ?fr. FILTER (?fr != sn:pers%Person%)}
       }
    } .
    ?fr snvoc:firstName ?first .
    ?fr snvoc:lastName ?last .
}
ORDER BY ?startdate ?fr ?orgname
LIMIT 10
```

Query 12: Expert search

```
sparql SELECT ?exp ?first ?last
         sql:group_concat_distinct(?tagname, ', ') count (*)
WHERE {
    sn:pers%Person% snvoc:knows ?exp .
    ?exp snvoc:firstName ?first .
    ?exp snvoc:lastName ?last .
    ?reply snvoc:hasCreator ?exp .
    ?reply snvoc:replyOf  ?org_post .
    FILTER (!exists {?org_post snvoc:replyOf ?xx}) .
    ?org_post snvoc:hasTag ?tag .
    ?tag foaf:name ?tagname .
    ?tag A ?type.
    ?type rdfs:subClassOf* ?type1 .
    ?type1 rdfs:label "%TagType%" .
}
group BY ?exp ?first ?last
ORDER BY DESC(5) ?exp
LIMIT 20
```

Query 13: Single shortest path

```
sparql SELECT count(*)
WHERE
  {
    {
      SELECT ?s ?o
      WHERE
        {
          ?s snvoc:knows ?o.
```

```
        }
    }
    option ( transitive,
             t_distinct,
             t_in(?s),
             t_out(?o),
             t_shortest_only,
             t_direction 3,
             t_step ('path_id') as ?path_no) .
    FILTER ( ?s = sn:pers%Person1% ).
    FILTER ( ?o = sn:pers%Person2% ).
    FILTER (?path_no = 0).
  }
```

Query 14: Weighted shortest path

```
sparql SELECT sql:path_str_sparql(?path), ?sc
WHERE
{
  SELECT ?path_no, sql:vector_agg
  (bif:vector (?via1, ?via2, ?cweight)) as ?path,
  sum (?cweight) as ?sc
  WHERE
  {
    SELECT ?via1 ?via2 ?path_no ?step_no
    sql:c_weight_sparql(?via1, ?via2) as ?cweight
    WHERE
    {
      {
        SELECT ?s bif:idn(?s) as ?via2 ?o
        WHERE
        {
          ?s snvoc:knows ?o.
        }
      }
      option ( transitive,
             t_distinct,
             t_in(?s),
             t_out(?o),
         t_shortest_only,
         t_direction 3,
         t_step (?s) as ?via1,
         t_step ('path_id') as ?path_no,
             t_step ('step_no') as ?step_no ) .
      FILTER ( ?s = sn:pers%Person1% ).
      FILTER ( ?o = sn:pers%Person2% ).
    }
  }
  group BY ?path_no
}
ORDER BY DESC(?sc)
LIMIT 10
```

A.2 Short Read Queries

S1: Profile view 1 (basic info about a user)

```
SELECT p_firstname, p_lastname, p_gender, p_birthday,
  p_creationdate, p_locationip, p_browserused, p_placeid
FROM person
WHERE
       p_personid = %personid%
```

S2: Profile view 2 (last 10 posts of a user)

```
SELECT top 10
     p1.ps_postid, p1.ps_content, p1.ps_imagefile, p1.ps_creationdate,
     p2.ps_postid, p2.p_personid, p2.p_firstname, p2.p_lastname
FROM post p1 LEFT OUTER JOIN
   (SELECT ps_postid, p_personid, p_firstname, p_lastname
      FROM post, person
      WHERE ps_creatorid = p_personid ) p2
ON p2.ps_postid = p1.ps_replyof
WHERE p1.ps_creatorid = %personid%
ORDER BY p1.ps_creationdate DESC
```

S3: Profile view 3 (list of friends of a user)

```
SELECT p_personid, p_firstname, p_lastname, k_creationdate
FROM knows, person
WHERE
         k_person1id = %personid% AND k_person2id = p_personid;
```

S4: Message view 1 (message content)

```
SELECT ps_content, ps_imagefile, ps_creationdate
FROM post
WHERE
       ps_postid = %postid%;
```

S5: Message view 2 (retrieve the author)

```
SELECT p_personid, p_firstname, p_lastname
FROM post, person
WHERE
  ps_postid = %postid% AND ps_creatorid = p_personid;
```

S6: Message view 3 (retrieve the forum)

```
SELECT f_forumid, f_title, p_personid, p_firstname, p_lastname
FROM post, person, forum
WHERE
 ps_postid = postid AND ps_forumid = f_forumid
   AND f_moderatorid = p_personid;
```

S7: Message view 4 (get the replies)

```
SELECT p2.ps_postid, p2.ps_content, p_personid, p_firstname, p_lastname,
  (CASE WHEN EXISTS (
           SELECT 1 FROM knows
                WHERE
                p1.ps_creatorid = k_person1id AND
```

```
            p2.ps_creatorid = k_person2id)
        THEN 1
        ELSE 0
        END)
FROM post p1, post p2, person
  WHERE
    p1.ps_postid = %postid% AND p1.ps_replyof = p2.ps_postid
    AND p2.ps_creatorid = p_personid;
```