Full Disclosure Report of the 108T LDBC Social Network Benchmark

LDBC Social Network Benchmark's Business Intelligence Workload over TigerGraph

February 10, 2023

Disclaimer: the benchmark results on this report are not official LDBC benchmark results, as they have not been audited.

Executive Summary

Scalability is a critical factor in the design and implementation of graph databases, especially given the ever-increasing volume of data generated by businesses. In this context, TigerGraph's recent performance in the LDBC SNB (Social Network Benchmark) BI (Business Intelligence) workload on the 108T dataset is remarkable, as it demonstrates the company's ability to scale its graph database seamlessly to accommodate the growth of data. This builds upon the success reported last year (2022), where TigerGraph completed the LDBC SNB BI benchmark on the SF-30000 dataset. The results of this benchmark serve to highlight the importance of scalability in graph databases and confirm TigerGraph's position as a leading provider in the graph database space.

This report details the full execution of the LDBC SNB BI workload for TigerGraph at the 108TB scale. The benchmark was conducted in accordance with the official benchmark driver, data, and substitution parameter generator as reported in the LDBC 1T Audit FDR over TigerGraph and LDBC SF30k FDR over TigerGraph. The 108TB data set was inflated based on the SF30k dataset and the queries were accordingly modified to activate the entire graph. The benchmark involved the execution of the queries using five substitution parameters in each batch, whereas the official benchmark uses 30 different parameters. The power and throughput benchmark metrics were reported in accordance with the guidelines specified in the LDBC SNB specification.

TigerGraph is a massively parallel processing (MPP) graph database management system designed for handling hybrid transaction/analytical processing (HTAP) query workloads. It is a distributed platform using a native graph storage format with an edge-cut partitioning strategy. Within this, each graph partition holds a similar amount of vertices and edges and processes requests in parallel. TigerGraph offers GSQL, a Turing-complete query language that provides both declarative features (e.g., graph patterns) as well as imperative ones (e.g. for expressing iterative graph algorithms with loops and accumulator primitives).

The focus of this benchmark test is TigerGraph's performance on Business Intelligence (BI) workloads over a sequence of batch-refreshed big graphs. The BI workload includes:

- **20 Read Queries**—the majority of OLAP-style iterative and deep-link graph queries were answered in sub-minute to a couple of minutes. The queries include explosive and redundant multi-joins and multi-source shortest path problems on a weighted graph.

- **Incremental Batch Updates**—the graph is mutated by a set of insert and delete operations. The data to be inserted or deleted are batched for a period of one day.

The following numerical novelties are highlighted in this high-level summary:

- The full source dataset is about 108TB, with a total of **1.619 trillion** relationships and **218 billion** vertices.
- The benchmarking process, which included an initial data loading phase, one power batch run, and one throughput batch run (with each batch executing all BI queries five times), took a total of **35.4** hours.
- The hardware cost for the benchmarking was **$843/hr**, and the setup consisted of **72 AWS r6a.48xlarge** machines and **432TB** GP3 SSD volumes.
- The 108TB data set was an inflation of the LDBC SNB official SF-30000 dataset. The LDBC SNB schema and BI workload queries were adjusted using advanced techniques to ensure that the BI queries would activate the whole 108TB data set.

1 Experiment Overview

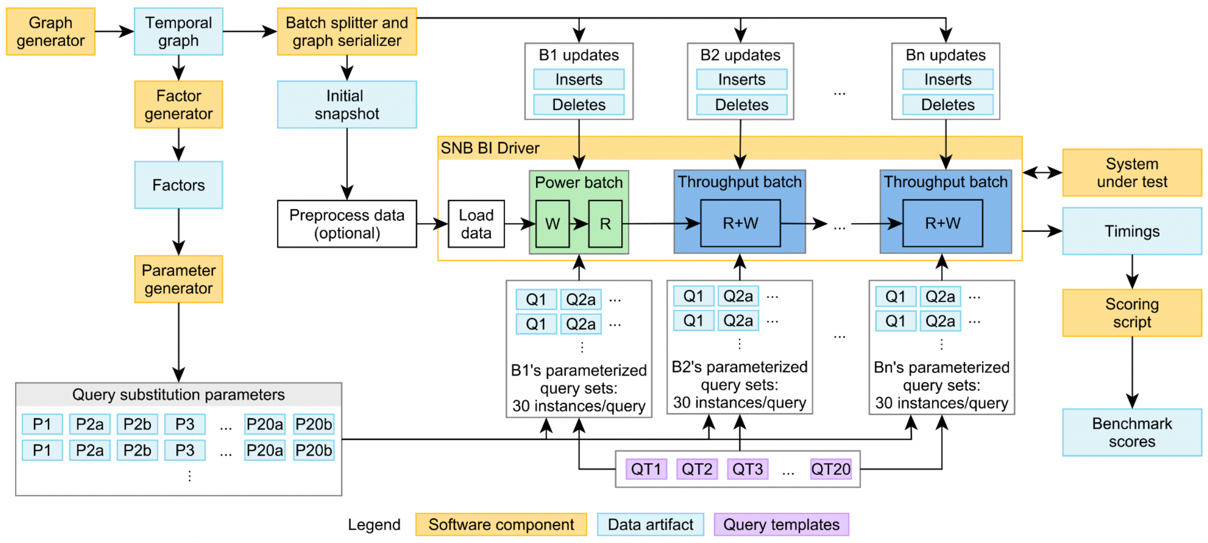


Fig. 1.1 Overview of the LDBC SNB BI workload. The SNB BI driver prompts TigerGraph to load data, perform one power batch and several throughput batches sequentially. Each power/throughput batch consists of write (W) and read (R) operations, where write operations consume inserts/deletes data and read operations run query templates (QT) on substitution parameters (P1, P2a, P2b, etc.) (source: [The LDBC Social Network Benchmark version 2.2.1](#))

The experiment was performed in accordance with the specifications of the Social Network Benchmark (SNB). However, the number of substitution parameters used for executing the queries was reduced to 5 instead of the standard 30, and the 108TB data set was inflated based on the SF30k dataset. The queries were correspondingly adjusted to activate the entire graph.

Table 1.1: Benchmark Overview

Artifact	**Version**	**URL**
Specification	2.2.0	https://arxiv.org/pdf/2001.02299v7.pdf
Data generator	0.5.0	https://github.com/ldbc/ldbc_snb_datagen_spark/releases/tag/v0.5.0
Driver and implementations	1.0.2	https://github.com/ldbc/ldbc_snb_bi/releases/tag/v1.0.2

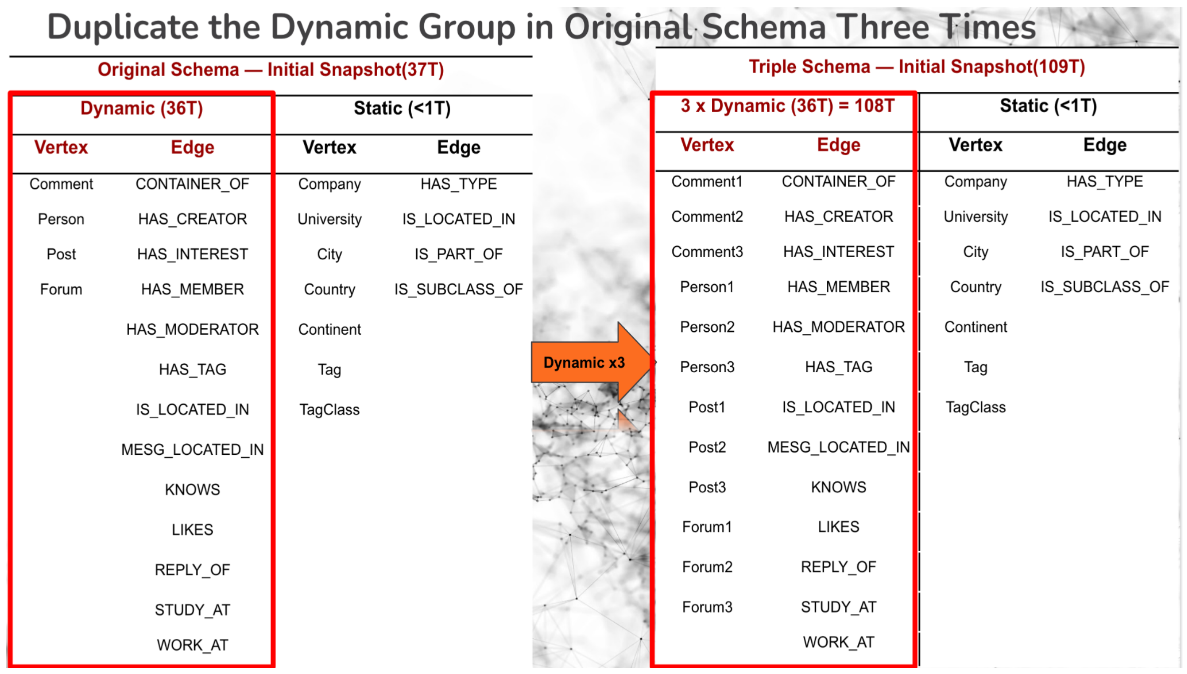
2 Methodology for 108TB Graph

2.1 Data inflation

The LDBC_SNB SF-30k data generator produces 44TB of raw data, consisting of 36.5TB in the initial snapshot and 7TB in the form of inserts and deletes. The initial snapshot includes 36TB of dynamic data, which can be altered through daily batch updates, and less than 1TB of static data, which remains unchanged, as shown in Fig.2.1.

To increase the volume of data, the 36TB of dynamic vertex and edge types were replicated three times, including the Comment, the Person, the Post, the Forum, and their associated edges. This resulted in a total of more than 108TB of raw data loaded into TigerGraph. For instance, the new graph schema includes Comment1, Comment2, and Comment3. All dynamic vertex types are connected to the original static vertex types, forming a single, cohesive graph. As a result, this methodology preserves the realism of data generated from the LDBC SF30k data generator while ensuring that the adjusted queries can effectively access the entire 108TB data set.

All implementations can be found here in the GitHub repository.



Duplicate the Dynamic Group in Original Schema Three Times

Original Schema — Initial Snapshot(37T)

Dynamic (36T)		Static (<1T)	
Vertex	Edge	Vertex	Edge
Comment	CONTAINER_OF	Company	HAS_TYPE
Person	HAS_CREATOR	University	IS_LOCATED_IN
Post	HAS_INTEREST	City	IS_PART_OF
Forum	HAS_MEMBER	Country	IS_SUBCLASS_OF
	HAS_MODERATOR	Continent	
	HAS_TAG	Tag	
	IS_LOCATED_IN	TagClass	
	MESG_LOCATED_IN		
	KNOWS		
	LIKES		
	REPLY_OF		
	STUDY_AT		
	WORK_AT		

Triple Schema — Initial Snapshot(109T)

3 x Dynamic (36T) = 108T		Static (<1T)	
Vertex	Edge	Vertex	Edge
Comment1	CONTAINER_OF	Company	HAS_TYPE
Comment2	HAS_CREATOR	University	IS_LOCATED_IN
Comment3	HAS_INTEREST	City	IS_PART_OF
Person1	HAS_MEMBER	Country	IS_SUBCLASS_OF
Person2	HAS_MODERATOR	Continent	
Person3	HAS_TAG	Tag	
Post1	IS_LOCATED_IN	TagClass	
Post2	MESG_LOCATED_IN		
Post3	KNOWS		
Forum1	LIKES		
Forum2	REPLY_OF		
Forum3	STUDY_AT		
	WORK_AT		

Fig. 2.1 Methodology of graph inflation from 36TB to 108TB

2.2 Schema change

We tripled the 4 dynamic vertex types, such as the Comment, the Person, the Forum, and the Post vertex types, and therefore, the 108TB graph schema contains indexed dynamic vertex types. Use the Comment vertex type as an example, the following DDL tripled the comments graph elements in three containers, respectively.

```
        CREATE VERTEX Comment1 (PRIMARY_ID id UINT, creationDate INT, locationIP STRING, browserUsed
STRING, content STRING, length UINT) WITH primary_id_as_attribute="TRUE"
        CREATE VERTEX Comment2 (PRIMARY_ID id UINT, creationDate INT, locationIP STRING, browserUsed
STRING, content STRING, length UINT) WITH primary_id_as_attribute="TRUE"
        CREATE VERTEX Comment3 (PRIMARY_ID id UINT, creationDate INT, locationIP STRING, browserUsed
STRING, content STRING, length UINT) WITH primary_id_as_attribute="TRUE"
```

We multiplied the edge data without creating new edge types. Instead, we increased both the source and target endpoints, enabling us to load edge data to specific combinations of source and target endpoints. This method also facilitates modifying our queries because we can keep the edge pattern as it is. Use the directed edge CONTAINER_OF and REPLY_OF as examples, the following DDL connecting the tripled vertex types to their corresponding targeted vertex types based on type indices.

```
        CREATE DIRECTED EDGE CONTAINER_OF (FROM Forum1, TO Post1 | FROM Forum2, TO Post2 | FROM
Forum3, TO Post3) WITH REVERSE_EDGE="CONTAINER_OF_REVERSE"

        CREATE DIRECTED EDGE REPLY_OF (FROM Comment1, TO Comment1 | FROM Comment2, TO Comment2 |
FROM Comment3, TO Comment3 | FROM Comment1, TO Post1 | FROM Comment2, TO Post2 | FROM Comment3, TO
Post3) WITH REVERSE_EDGE="REPLY_OF_REVERSE"
```

2.3 Query amendment

As we only modified the dynamic vertex types in the schema, the adjusted queries must substitute the original dynamic vertex types with the multiplied vertex types. No new edge types were added, so the edge pattern remains unchanged. The static vertex types are connected to all three groups of dynamic vertex types, enabling the BI queries to access the entire 108TB data through the static vertex types. For example, the adjusted BI-5 query is:

```
CREATE OR REPLACE DISTRIBUTED QUERY bi5(STRING tag) SYNTAX v2 {
TYPEDEF TUPLE <UINT personId, UINT replyCount, UINT likeCount, UINT messageCount, UINT score>
RESULT;
HeapAccum<RESULT>(100, score DESC, personId ASC) @@result; SumAccum<UINT> @likeCount;
SumAccum<UINT> @messageCount; SumAccum<UINT> @replyCount;
T = SELECT t FROM Tag:t WHERE t.name == tag;
messages = SELECT m FROM T:t -(<HAS_TAG)- (Comment1|Comment2|Comment3|Post1|Post2|Post3):m;
tmp = SELECT m FROM messages:m -(<LIKES)- (Person1|Person2|Person3):p ACCUM m.@likeCount += 1;
tmp = SELECT m FROM messages:m -(<REPLY_OF)- (Comment1|Comment2|Comment3):c ACCUM m.@replyCount +=
1;
tmp = SELECT p FROM messages:m -(HAS_CREATOR>)- (Person1|Person2|Person3):p
ACCUM p.@replyCount += m.@replyCount, p.@likeCount += m.@likeCount, p.@messageCount += 1
POST-ACCUM @@result += RESULT(p.id, p.@replyCount, p.@likeCount, p.@messageCount,
               p.@messageCount + 2*p.@replyCount + 10*p.@likeCount);
PRINT @@result as result;}
```

Note: Most bi queries are modified based on the above implementation and can touch the whole 108TB dataset except 4 querie. Queries bi-14, bi-15, bi-17, and bi-18 only activate 36T data due to a limitation in TigerGraph v3.7.0 where VERTEX<> does not support multiple vertex type specifications. These queries only feature one singular dynamic seed vertex in the parameter field and do not traverse to the joint static vertex, thus leaving other duplicated dynamic vertices untraversed.

3 System Description and Pricing Summary

3.1 Machine overview

The hardware used for benchmarking LDBC-SNB on 108TB are 72 AWS EC2 instances of type r6a.48xlarge. Amazon EC2 R6a AMD instances are powered by 3rd Generation AMD EPYC processors and deliver up to 35% better price performance compared to R5a instances and 10% lower cost than comparable x86-based Amazon instances. These instances are ideal fit for memory-intensive workloads, such as Graph Databases and Graph Analytics.
The r6a.48xlarge instances feature 192 cores per virtual machine, a high memory depth of 1.5TB per VM, and a high EBS storage bandwidth of 40Gbps that enables outstanding performance on even the most complex graph algorithms at scale.

Table 2.1: Machine Type

Number of Virtual Machines	72
Instance Type	r6a.48xlarge
Operating System	Amazon Linux 2 AMI (HVM) - Kernel 5.10
vCPU	192 cores/node
Memory	1.536T/node

3.2 CPU details

The details below were obtained using the commands `cat /proc/cpuinfo` (Listing A.1) and `lscpu` (Listing A.2).

Table 2.2: CPU details summary

Type	AMD® EPYC® 7R13 Processor

Total number	2
Cores per CPU	48
Threads per CPU	96
Total threads	192
CPU clock frequency	3.564 GHz
Total cache size per CPU	L1d cache: 32K L1i cache: 32K L2 cache: 512K L3 cache: 32768K

3.3 Memory details

The total aggregate size of the memory installed is 110.6 TB. This information was obtained using the `cat /proc/meminfo` (Listing A.3) and `lshw -C` memory (Listing A.4) commands.

3.4 Disk and storage details

Table 2.3: Disk details summary

Disk	AWS General Purpose SSD (gp3)
Device Size	6 TB
Max IOPS	16000
Max throughput	1000 MB/s

3.5 Network details

Table 2.4: Network details summary

Instance	r6a.48xlarge
Network Bandwidth	50 Gbps
EBS bandwidth	40 Gbps

3.6 Machine pricing

Table 2.5: Pricing Summary

Item	Price
Total AWS EC2 instance cost for 3 years	$20,598,810 ($783.82/hr)
Total AWS EC2 volume cost for 3 years	$1,503,360 ($41,760/month)

3.7 System version and availability

Table 2.6: System version

System	Version	License
TigerGraph	3.7.0	Enterprise License provided by TigerGraph

4 Dataset Generation

4.1 General information

As demonstrated in the methodology chapter, the 108TB dataset was inflated based on SF-30k dataset, and therefore, the following sections focused on the data generation of SF-30k. Datasets larger than SF10k are currently not available in the official Cloudflare R2. The dataset and query substitution parameters are generated using the LDBC's official data generator v0.5.0 and stored in the GCP bucket hosted by TigerGraph (gs://ldbc_bi/sf3000and gs://ldbc_bi/parameters_sf3000). The data generation settings of the LDBC Datagen are described below.

Table 4.1: Datagen settings summary

Data format	composite-projected-fk layout, compressed CSV files
Scale factors	30000

4.2 Data loading and data schema

The data preprocessing and loading times are reported below. The column **Data preprocessing time** shows how much time it took to preprocess the CSV files. For this benchmark execution, the preprocessing only consisted of decompressing the .csv.gz files. The column **Data loading time** shows how long it took to create a graph from the input CSV files and perform the initial indexing of vertices and edges, including schema setup, initial data loading, query installation, pre-computation, etc. The initial data loading alone took 45900 s. The column **Total time** contains the sum of the data preprocessing and loading times. The TigerGraph topology data size 44.4 TB and compression ratio are shown in Fig. 3.1. The TigerGraph data schema is shown in Listing D.1 in Section 9.

The following configurations were updated on top of the default configuration:

```
gadmin config entry GPE.BasicConfig.Env
    ●   Add “MVExtraCopy=0;”  //default is 1; this turns off backup copy
    ●   Add “MaxNumberSegments=1310720;” //default is 131072 which is too small for 108T

gsql
    ●   set segsize_in_bits = 24  //to increase the segment size

gadmin config entry GPE.BasicConfig.Env - gadmin config group timeout
    ●   FileLoader.Factory.DefaultQueryTimeoutSec: 16 -> 6000
    ●   KafkaLoader.Factory.DefaultQueryTimeoutSec: 16 -> 6000
    ●   RESTPP.Factory.DefaultQueryTimeoutSec: 16 -> 6000
```

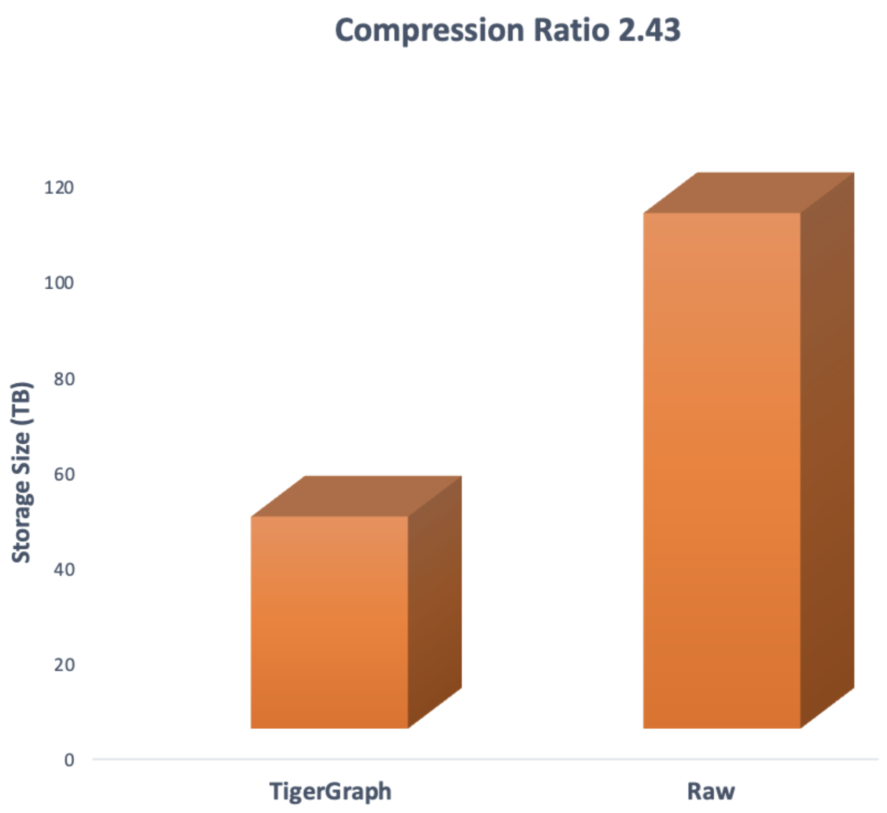


Fig. 4.1 The disk space consumed by the dataset was 2.43 times smaller once loaded into TigerGraph

Table 4.2: Data preprocessing and loading times for TigerGraph on 108TB

Data size	Data preprocessing time	Data loading time	Total time
108 TB	8 min (457 s)	12hr45min(45 900s)	12hr53min(46 357s)

4.3 Data statistics

The statistics of the initial state for each vertex and edge type is shown in the following table.

Table. 4.3 Cardinality for each vertex type (total **217.86B** vertices)

Vertex	Count
Comment1	58,666,958,815
Comment2	58,666,958,815
Comment3	58,666,958,815
Post1	13,148,296,221

Post2	13,148,296,221
Post3	13,148,296,221
Forum1	728,629,666
Forum2	728,629,666
Forum3	728,629,666
Person1	74,689,437
Person2	74,689,437
Person3	74,689,437
Company	4,725
University	19,140
City	4,029
Country	333
Continent	18
Tag	48,240
TagClass	213
Subtotal	217,855,799,115

Table. 4.4 Cardinality for each edge type (total **1.62T** edges)

Edges	**Count**
CONTAINER_OF	39,444,888,663
HAS_CREATOR	215,445,765,108
HAS_INTEREST	5,243,002,503
HAS_MEMBER	271,956,270,042
HAS_MODERATOR	2,185,888,998
HAS_TAG	304,603,732,866
IS_LOCATED_IN	224,076,266
MESG_LOCATED_IN	215,445,765,108
KNOWS	17,203,410,066
LIKES	370,276,474,926
REPLY_OF	176,000,876,445
STUDY_AT	179,275,377
WORK_AT	487,556,766
HAS_TYPE	16,080
IS_PART_OF	1,454
IS_SUBCLASS_OF	70
Subtotal	1,618,697,000,738

5 Benchmark workflow and implementation

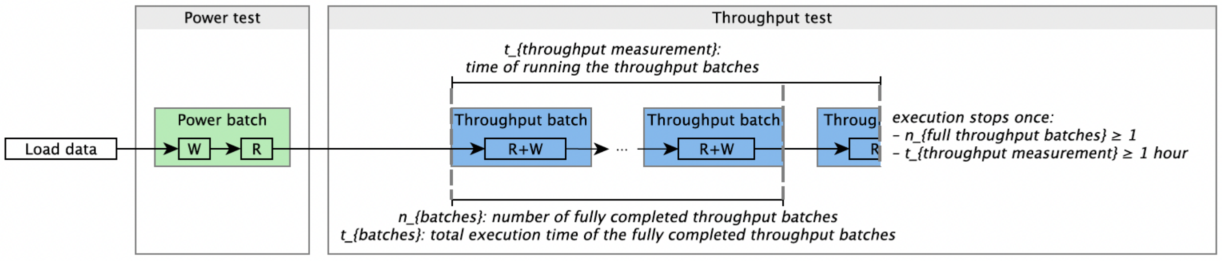

Power test
Throughput test
t_{throughput measurement}:
time of running the throughput batches
Load data
Power batch
W
R
Throughput batch
R+W
...
Throughput batch
R+W
Throug
R
execution stops once:
- n_{full throughput batches} ≥ 1
- t_{throughput measurement} ≥ 1 hour
n_{batches}: number of fully completed throughput batches
t_{batches}: total execution time of the fully completed throughput batches

Fig. 5.1 Tests and batches (power and throughput) executed in the BI workload's workflow (source: The LDBC Social Network Benchmark version 2.2.1)

In Fig.5.1, the benchmark consists of load data, power test and throughput test. The power test, which runs a single power batch, first executes the write operations and then a sequential execution of individual read query variants. The official benchmark requires 28 query variants x 30 substitution parameters = 840 read queries to be executed, but in this benchmark, we executed 5 substitution parameters, and thus a total of 28 x 5 = 140 queries.

The throughput test consists of multiple throughput batches. The type and number of operations in each throughput batch are the same as the power batch. The only difference is that the throughput test allows the operations executed concurrently. In the current implementation, the throughput batch is implemented in the same way as power batch where the write and read operations are performed sequentially.

The implementation is similar to in the previous LDBC audited SF-1000 benchmark. However, the number of substitution parameters used for executing the queries was reduced to 5 instead of the standard 30, and the 108TB data set was inflated based on the SF30k dataset. The queries were correspondingly adjusted to activate the entire graph.

6 Performance Results

Based on the Sec 7.5.4 Scoring Metrics in Specification, the *power score*, *throughput score* and their price adjusted variants *per-$ power score* and *per-$ throughput score* are calculated. The benchmark time shown in Table 6.1 indicates the total elapsed time for both the power and throughput batch. The power score is calculated as

$$power@SF = \frac{3\,600}{\sqrt[29]{w \cdot q_1 \cdot q_{2a} \cdot q_{2b} \cdot \ldots \cdot q_{18} \cdot q_{19a} \cdot q_{19b} \cdot q_{20a} \cdot q_{20b}}} \cdot SF$$

where $w = 15160$ is the time in second to perform the writes and $q_1, q_{2a}, q_{2b} \ldots q_{20b}$ are the time in second for executing each variant with 30 different substitution parameters. In this benchmark, $q_1, q_{2a}, q_{2b} \ldots q_{20b}$ are extrapolated by applying a factor of 6 to the time spent on 5 substitution parameters (i.e., the sum column in Table 6.2).
The throughput score is calculated as

$$throughput@SF = (24 \text{ hours} - t_{load}) \cdot \frac{n_{batches}}{t_{batches}} \cdot SF$$

where t_{load} is the load time and is 12.88 hr, $n_{batches} = 1$ is the number of throughput batch in this benchmark, $t_{batches}$ is the time spent in a throughput batch with 30 substitution parameters and is 37.9 hr.

The price-adjusted score are

$$power@SF/\$ = power@SF \times \frac{1000}{Cost},$$
$$throughput@SF/\$ = throughput@SF \times \frac{1000}{Cost},$$

where the *Cost* only includes the hardware cost, whereas the specification uses the *total cost of ownership (TCO)* that includes both software and hardware cost.

Table 6.1: Summary of results for TigerGraph at 108TB dataset.
The benchmark time includes both the power batch (inserts/deletes+precompute+read) and the throughput batch (inserts/deletes+precompute+read) elapsed time.

Benchmark time	Power@SF	Power@SF/$	Throughput@SF	Throughput@SF/$
22.67 hr	106 034.76	9.78	26 398.01	2.43

Table 6.2: Detailed power test results for TigerGraph at 108TB dataset. Execution times for five runs with different substitution parameters per query are reported in seconds.

Query	**Sum (5 runs)**	**Max.**	**Min.**	**Mean**	**P_{50}**	**P_{90}**	**P_{95}**	**P_{99}**
1	190.85	41.77	36.17	38.17	37.20	40.50	41.13	41.64
2a	1356.09	322.91	153.44	271.22	312.56	320.04	321.47	322.62
2b	688.63	161.29	125.84	137.73	136.32	151.55	156.42	160.31
3	1569.73	642.00	195.34	313.95	221.75	507.38	574.69	628.54
4	86.07	18.79	14.77	17.21	17.66	18.45	18.62	18.76
5	276.02	70.10	46.62	55.20	55.43	64.92	67.51	69.58
6	266.63	65.31	47.81	53.33	50.19	61.16	63.24	64.90
7	1068.72	223.63	199.79	213.74	221.30	223.32	223.48	223.60

8a	395.34	85.51	73.94	79.07	78.81	83.22	84.36	85.28
8b	227.08	50.60	41.31	45.42	44.49	49.32	49.96	50.47
9	790.82	162.24	150.31	158.16	159.20	161.46	161.85	162.17
10a	867.05	204.22	146.86	173.41	162.77	203.64	203.93	204.16
10b	353.49	100.48	32.47	70.70	76.94	93.09	96.79	99.74
11	211.01	45.90	40.27	42.20	40.61	45.06	45.48	45.82
12	759.12	168.89	136.31	151.82	147.61	166.70	167.79	168.67
13	1780.49	362.29	351.08	356.10	356.33	361.14	361.71	362.17
14a	667.76	138.77	124.42	133.55	135.44	138.23	138.50	138.71
14b	284.12	66.48	46.09	56.82	57.61	63.25	64.87	66.16
15a	897.11	183.40	166.82	179.42	182.41	183.01	183.20	183.36
15b	3004.23	664.72	509.15	600.85	627.38	652.41	658.57	663.49
16a	1728.39	411.64	245.30	345.68	359.15	406.70	409.17	411.15
16b	418.36	86.47	79.16	83.67	84.10	86.28	86.38	86.45
17	548.34	113.44	104.09	109.67	110.01	112.79	113.12	113.38
18	1156.12	234.36	227.99	231.22	231.33	233.42	233.89	234.27
19a	262.11	55.64	49.01	52.42	52.31	54.65	55.14	55.54
19b	260.12	56.13	48.59	52.02	51.82	55.33	55.73	56.05
20a	77.12	36.20	8.87	15.42	9.49	27.06	31.63	35.29
20b	86.69	27.06	11.59	17.34	16.09	23.19	25.12	26.67

Table 6.3: Operations in the **power test** for TigerGraph at 108TB dataset. Execution times are reported in seconds. ROOT_POST[1] pre-computations are performed for each Comment insertion and deletion operation. Therefore, they are reported as part of the writes.

Operation	**Time (hh:mm:ss)**	**Time (seconds)**
Total read time (5 runs)	5:37:57	20,277.70
Total write time	4:12:40	15,160.01
Precomputation for Q4	0:06:22	382.08
Precomputation for Q6	0:15:59	959.31
Precomputation for Q14 and Q19	2:04:53	7,493.26
Precomputation for Q20	0:06:17	376.55

7 Conclusion

The benchmark in this report showcases TigerGraph's scalability in graph analytics and business intelligence on graph-structured data at a scale of hundreds of terabytes. The new LDBC SNB BI workloads include two challenges: a micro-batch of insert and delete operations to modify the current graph, and complex read queries that access a substantial portion of the data. These queries were designed based on choke points and challenging aspects of query processing, such as multi-joins that are both explosive and redundant, and expressive pathfinding.

TigerGraph was able to effectively handle deep-link OLAP style queries on a big graph of 217.9 billion vertices and 1.6 trillion edges. Eleven of the data-intensive read queries returned results within 1 minute, while the rest took between 1 and 10 minutes.

The R6a instances have proven capable of meeting the objectives of the 108TB SNB BI benchmark, and they exhibit excellent scalability as dataset sizes increase. As such, they are highly recommended for large-scale deployments that demand powerful and adaptable computing infrastructure.

This benchmark demonstrates TigerGraph's capability to handle big graph workloads in a real-world production environment, where daily or hourly incremental updates of tens of terabytes of connected data are common. To the best of our knowledge, no other graph database or relational database vendor has demonstrated similar analytical and operational capabilities on such a large-scale, updatable graph.

[1]In the precomputation phase, an auxiliary edge is added between each comment and its root Post. This edge is called ROOT_POST

8 Acknowledgement

The cloud computing cost used in this benchmark is funded by AMD and is gratefully acknowledged.

9 Supplemental Materials

A CPU and Memory Details

Listing A.1: Output of the `cat /proc/cpuinfo` command for a single CPU core

```
processor       : 191
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 1
model name      : AMD EPYC 7R13 Processor
stepping        : 1
microcode       : 0xa001173
cpu MHz         : 3358.650
cache size      : 512 KB
physical id     : 1
siblings        : 96
core id         : 55
cpu cores       : 48
apicid          : 239
initial apicid  : 239
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc
cpuid extd_apicid aperfmperf tsc_known_freq pni pclmulqdq monitor ssse3 fma cx16 pcid sse4_1 sse4_2
x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a
misalignsse 3dnowprefetch topoext perfctr_core invpcid_single ssbd ibrs ibpb stibp vmmcall fsgsbase bmi1
avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 clzero
xsaveerptr rdpru wbnoinvd arat npt nrip_save vaes vpclmulqdq rdpid
bugs            : sysret_ss_attrs null_seg spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 5299.97
TLB size        : 2560 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management:
```

Listing A.2: Output of the `lscpu` command

```
    Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              192
On-line CPU(s) list: 0-191
Thread(s) per core:  2
Core(s) per socket:  48
Socket(s):           2
NUMA node(s):        4
Vendor ID:           AuthenticAMD
CPU family:          25
Model:               1
Model name:          AMD EPYC 7R13 Processor
Stepping:            1
CPU MHz:             3563.865
BogoMIPS:            5299.97
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            32768K
NUMA node0 CPU(s):   0-23,96-119
NUMA node1 CPU(s):   24-47,120-143
NUMA node2 CPU(s):   48-71,144-167
NUMA node3 CPU(s):   72-95,168-191
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good
nopl nonstop_tsc cpuid extd_apicid aperfmperf tsc_known_freq pni pclmulqdq monitor ssse3 fma cx16
pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy
cr8_legacy abm sse4a misalignsse 3dnowprefetch topoext perfctr_core invpcid_single ssbd ibrs ibpb
stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni
xsaveopt xsavec xgetbv1 clzero xsaveerptr rdpru wbnoinvd arat npt nrip_save vaes vpclmulqdq rdpid
```

Listing A.3: Output of the `cat /proc/meminfo` command

```
MemTotal:       1552189116 kB
MemFree:        494708720 kB
MemAvailable:   1362323316 kB
Buffers:            2704 kB
Cached:         873227972 kB
SwapCached:            0 kB
Active:         47234164 kB
Inactive:       1005192256 kB
Active(anon):     177428 kB
Inactive(anon): 179360420 kB
Active(file):   47056736 kB
Inactive(file): 825831836 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:            241444 kB
Writeback:             0 kB
```

```
AnonPages:      179185916 kB
Mapped:            448420 kB
Shmem:             415480 kB
KReclaimable:     2434316 kB
Slab:             3200796 kB
SReclaimable:     2434316 kB
SUnreclaim:        766480 kB
KernelStack:       109472 kB
PageTables:        468372 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:    776094556 kB
Committed_AS:   45633820 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       540292 kB
VmallocChunk:           0 kB
Percpu:             58368 kB
HardwareCorrupted:      0 kB
AnonHugePages:          0 kB
ShmemHugePages:         0 kB
ShmemPmdMapped:         0 kB
FileHugePages:          0 kB
FilePmdMapped:          0 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:        2048 kB
Hugetlb:                0 kB
DirectMap4k:       911208 kB
DirectMap2M:    92338176 kB
DirectMap1G:    1483735040 kB
```

Listing A.4: Output of the `lshw -C memory` command

```
    *-firmware
       description: BIOS
       vendor: Amazon EC2
       physical id: 0
       version: 1.0
       date: 10/16/2017
       size: 64KiB
       capacity: 64KiB
       capabilities: pci edd acpi virtualmachine
  *-cache:0
       description: L1 cache
       physical id: 6
       slot: L1 - Cache
       size: 3MiB
       capacity: 3MiB
       clock: 1GHz (1.0ns)
       capabilities: pipeline-burst internal write-back unified
       configuration: level=1
  *-cache:1
       description: L2 cache
       physical id: 7
       slot: L2 - Cache
       size: 24MiB
       capacity: 24MiB
```

```
        clock: 1GHz (1.0ns)
        capabilities: pipeline-burst internal write-back unified
        configuration: level=2
   *-cache:2
        description: L3 cache
        physical id: 8
        slot: L3 - Cache
        size: 192MiB
        capacity: 192MiB
        clock: 1GHz (1.0ns)
        capabilities: pipeline-burst internal write-back unified
        configuration: level=3
   *-cache:0
        description: L1 cache
        physical id: 9
        slot: L1 - Cache
        size: 3MiB
        capacity: 3MiB
        clock: 1GHz (1.0ns)
        capabilities: pipeline-burst internal write-back unified
        configuration: level=1
   *-cache:1
        description: L2 cache
        physical id: a
        slot: L2 - Cache
        size: 24MiB
        capacity: 24MiB
        clock: 1GHz (1.0ns)
        capabilities: pipeline-burst internal write-back unified
        configuration: level=2
   *-cache:2
        description: L3 cache
        physical id: b
        slot: L3 - Cache
        size: 192MiB
        capacity: 192MiB
        clock: 1GHz (1.0ns)
        capabilities: pipeline-burst internal write-back unified
        configuration: level=3
   *-memory
        description: System Memory
        physical id: c
        slot: System board or motherboard
        size: 1536GiB
      *-bank:0
           description: DIMM DDR4 Static column Pseudo-static Synchronous Window DRAM 3200 MHz (0.3 ns)
           physical id: 0
           size: 384GiB
           width: 64 bits
           clock: 3200MHz (0.3ns)
      *-bank:1
           description: DIMM DDR4 Static column Pseudo-static Synchronous Window DRAM 3200 MHz (0.3 ns)
           physical id: 1
           size: 384GiB
           width: 64 bits
           clock: 3200MHz (0.3ns)
      *-bank:2
           description: DIMM DDR4 Static column Pseudo-static Synchronous Window DRAM 3200 MHz (0.3 ns)
           physical id: 2
           size: 384GiB
           width: 64 bits
           clock: 3200MHz (0.3ns)
      *-bank:3
           description: DIMM DDR4 Static column Pseudo-static Synchronous Window DRAM 3200 MHz (0.3 ns)
           physical id: 3
```

```
          size: 384GiB
          width: 64 bits
          clock: 3200MHz (0.3ns)
```

B IO Performance

Listing B.1: Output of the `fio` command

```
    read_iops_test: (g=0): rw=randread, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=256
fio-2.14
Starting 1 process
read_iops_test: Laying out IO file(s) (1 file(s) / 3072MB)

read_iops_test: (groupid=0, jobs=1): err= 0: pid=19706: Wed Jan 25 22:12:56 2023
  read : io=3725.3MB, bw=63559KB/s, iops=15885, runt= 60017msec
    slat (usec): min=1, max=1830, avg=61.14, stdev=168.97
    clat (usec): min=2579, max=42430, avg=16050.40, stdev=536.08
     lat (usec): min=2629, max=42432, avg=16112.55, stdev=536.66
    clat percentiles (usec):
     |  1.00th=[15296],  5.00th=[15424], 10.00th=[15552], 20.00th=[15680],
     | 30.00th=[15808], 40.00th=[15936], 50.00th=[15936], 60.00th=[16064],
     | 70.00th=[16192], 80.00th=[16512], 90.00th=[16768], 95.00th=[16768],
     | 99.00th=[17280], 99.50th=[17280], 99.90th=[18816], 99.95th=[22912],
     | 99.99th=[27776]
    lat (msec) : 4=0.01%, 10=0.04%, 20=99.90%, 50=0.08%
  cpu          : usr=1.02%, sys=3.45%, ctx=115874, majf=0, minf=1
  IO depths    : 1=0.1%, 2=0.1%, 4=0.1%, 8=0.1%, 16=0.1%, 32=0.1%, >=64=105.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.1%
     issued    : total=r=953407/w=0/d=0, short=r=0/w=0/d=0, drop=r=0/w=0/d=0
     latency   : target=0, window=0, percentile=100.00%, depth=256

Run status group 0 (all jobs):
   READ: io=3725.3MB, aggrb=63559KB/s, minb=63559KB/s, maxb=63559KB/s, mint=60017msec, maxt=60017msec

Disk stats (read/write):
  nvme0n1: ios=1001100/8030, merge=0/255, ticks=15546244/82632, in_queue=15628876, util=100.00%
```

C Dataset Generation Instructions

The datasets can be generated using the LDBC SNB Datagen. To regenerate the data sets used in this benchmark, build the Datagen JAR as described in the project's README, configure the AWS EMR environment, upload the JAR to the S3 bucket (denoted as `${BUCKET_NAME}`) and run the following commands to generate the datasets used in this audit.

Note that while the datasets for TigerGraph were generated as gzip-compressed archives, they are decompressed during preprocessing.

Listing C.1: Script to generate the SF30000 dataset for TigerGraph in AWS EMR. This dataset is used for the benchmark run

```
export SCALE_FACTOR=30000
export JOB_NAME=sf${SCALE_FACTOR}-projected-csv-gz

./tools/emr/submit_datagen_job.py \
--use-spot \
--bucket ${BUCKET_NAME} \
--copy-all \
--az us-east-2c \
${JOB_NAME} \
${SCALE_FACTOR} \
csv \
bi \
-- \
--explode-edges \
--format-options compression=gzip \
--generate-factors
```

D Data Schema

Listing D.1: Content of the GSQL schema used by TigerGraph

```
## Message

CREATE VERTEX Comment1 (PRIMARY_ID id UINT, creationDate INT, locationIP STRING, browserUsed STRING, content STRING, length UINT) WITH
primary_id_as_attribute="TRUE"

CREATE VERTEX Comment2 (PRIMARY_ID id UINT, creationDate INT, locationIP STRING, browserUsed STRING, content STRING, length UINT) WITH
primary_id_as_attribute="TRUE"

CREATE VERTEX Comment3 (PRIMARY_ID id UINT, creationDate INT, locationIP STRING, browserUsed STRING, content STRING, length UINT) WITH
primary_id_as_attribute="TRUE"

CREATE VERTEX Post1 (PRIMARY_ID id UINT, imageFile STRING, creationDate INT, locationIP STRING, browserUsed STRING, language STRING,
content STRING, length UINT) WITH primary_id_as_attribute="TRUE"

CREATE VERTEX Post2 (PRIMARY_ID id UINT, imageFile STRING, creationDate INT, locationIP STRING, browserUsed STRING, language STRING,
content STRING, length UINT) WITH primary_id_as_attribute="TRUE"

CREATE VERTEX Post3 (PRIMARY_ID id UINT, imageFile STRING, creationDate INT, locationIP STRING, browserUsed STRING, language STRING,
content STRING, length UINT) WITH primary_id_as_attribute="TRUE"

## organisation

CREATE VERTEX Company (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"

CREATE VERTEX University (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"

## place

CREATE VERTEX City (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"

CREATE VERTEX Country (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"

CREATE VERTEX Continent (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"

## etc

CREATE VERTEX Forum1 (PRIMARY_ID id UINT, title STRING, creationDate INT,

    maxMember UINT) WITH primary_id_as_attribute="TRUE" // maxMember is for precompute in BI-4

CREATE VERTEX Forum2 (PRIMARY_ID id UINT, title STRING, creationDate INT,

    maxMember UINT) WITH primary_id_as_attribute="TRUE" // maxMember is for precompute in BI-4

CREATE VERTEX Forum3 (PRIMARY_ID id UINT, title STRING, creationDate INT,

    maxMember UINT) WITH primary_id_as_attribute="TRUE" // maxMember is for precompute in BI-4

CREATE VERTEX Person1 (PRIMARY_ID id UINT, firstName STRING, lastName STRING, gender STRING, birthday INT, creationDate INT, locationIP
STRING, browserUsed STRING, speaks SET<STRING>, email SET<STRING>,

    popularityScore UINT) WITH primary_id_as_attribute="TRUE" // popularityScore is for precompute in BI-6

CREATE VERTEX Person2 (PRIMARY_ID id UINT, firstName STRING, lastName STRING, gender STRING, birthday INT, creationDate INT, locationIP
STRING, browserUsed STRING, speaks SET<STRING>, email SET<STRING>,

    popularityScore UINT) WITH primary_id_as_attribute="TRUE" // popularityScore is for precompute in BI-6

CREATE VERTEX Person3 (PRIMARY_ID id UINT, firstName STRING, lastName STRING, gender STRING, birthday INT, creationDate INT, locationIP
STRING, browserUsed STRING, speaks SET<STRING>, email SET<STRING>,

    popularityScore UINT) WITH primary_id_as_attribute="TRUE" // popularityScore is for precompute in BI-6

CREATE VERTEX Tag (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"

CREATE VERTEX TagClass (PRIMARY_ID id UINT, name STRING, url STRING) WITH primary_id_as_attribute="TRUE"



# create edge

CREATE DIRECTED EDGE CONTAINER_OF (FROM Forum1, TO Post1 | FROM Forum2, TO Post2 | FROM Forum3, TO Post3) WITH
REVERSE_EDGE="CONTAINER_OF_REVERSE"

CREATE DIRECTED EDGE HAS_CREATOR (FROM Comment1, TO Person1 | FROM Comment2, TO Person2 | FROM Comment3, TO Person3 | FROM Post1, TO
Person1 | FROM Post2, TO Person2 | FROM Post3, TO Person3 ) WITH REVERSE_EDGE="HAS_CREATOR_REVERSE"

CREATE DIRECTED EDGE HAS_INTEREST (FROM Person1|Person2|Person3, TO Tag) WITH REVERSE_EDGE="HAS_INTEREST_REVERSE"
CREATE DIRECTED EDGE HAS_MEMBER (FROM Forum1, TO Person1 | FROM Forum2, TO Person2 | FROM Forum3, TO Person3, creationDate INT) WITH
REVERSE_EDGE="HAS_MEMBER_REVERSE"

CREATE DIRECTED EDGE HAS_MODERATOR (FROM Forum1, TO Person1 | FROM Forum2, TO Person2 | FROM Forum3, TO Person3) WITH
REVERSE_EDGE="HAS_MODERATOR_REVERSE"

CREATE DIRECTED EDGE HAS_TAG (FROM Comment1|Comment2|Comment3|Post1|Post2|Post3|Forum1|Forum2|Forum3, TO Tag) WITH
REVERSE_EDGE="HAS_TAG_REVERSE"

CREATE DIRECTED EDGE HAS_TYPE (FROM Tag, TO TagClass) WITH REVERSE_EDGE="HAS_TYPE_REVERSE"

CREATE DIRECTED EDGE IS_LOCATED_IN (FROM Company, TO Country | FROM Person1, TO City | FROM Person2, TO City | FROM Person3, TO City |
FROM University, TO City) WITH REVERSE_EDGE="IS_LOCATED_IN_REVERSE"

CREATE DIRECTED EDGE MESG_LOCATED_IN (FROM Comment1|Comment2|Comment3|Post1|Post2|Post3, TO Country) // Reverse edge of
Comment1|Comment2|Comment3/Post1|Post2|Post3 -IS_Located_IN-> Country will cause Country connected by too many edges, which makes
loading slow

CREATE DIRECTED EDGE IS_PART_OF (FROM City, TO Country | FROM Country, TO Continent) WITH REVERSE_EDGE="IS_PART_OF_REVERSE"

CREATE DIRECTED EDGE IS_SUBCLASS_OF (FROM TagClass, TO TagClass) WITH REVERSE_EDGE="IS_SUBCLASS_OF_REVERSE"

CREATE UNDIRECTED EDGE KNOWS (FROM Person1, TO Person1 | FROM Person2, TO Person2 | FROM Person3, TO Person3, creationDate INT,
weight19 UINT, weight20 UINT DEFAULT 10000)

CREATE DIRECTED EDGE LIKES (FROM Person1, TO Comment1| FROM Person2, TO Comment2 | FROM Person3, TO Comment3 | FROM Person1, TO Post1 |
FROM Person2, TO Post2 | FROM Person3, TO Post3, creationDate INT) WITH REVERSE_EDGE="LIKES_REVERSE"

CREATE DIRECTED EDGE REPLY_OF (FROM Comment1, TO Comment1 | FROM Comment2, TO Comment2 | FROM Comment3, TO Comment3 | FROM Comment1, TO
Post1 | FROM Comment2, TO Post2 | FROM Comment3, TO Post3) WITH REVERSE_EDGE="REPLY_OF_REVERSE"

CREATE DIRECTED EDGE STUDY_AT (FROM Person1|Person2|Person3, TO University, classYear INT) WITH REVERSE_EDGE="STUDY_AT_REVERSE"

CREATE DIRECTED EDGE WORK_AT (FROM Person1|Person2|Person3, TO Company, workFrom INT) WITH REVERSE_EDGE="WORK_AT_REVERSE"


CREATE DIRECTED EDGE ROOT_POST (FROM Comment1, TO Post1 | FROM Comment2, TO Post2 | FROM Comment3, TO Post3) WITH
REVERSE_EDGE="ROOT_POST_REVERSE" //FOR BI-3,9,17

CREATE DIRECTED EDGE REPLY_COUNT (FROM Person1, TO Person1 | FROM Person2, To Person2 | FROM Person3, To Person3, cnt UINT)


CREATE GLOBAL SCHEMA_CHANGE JOB addIndex {

  ALTER VERTEX Country ADD INDEX country_name ON (name);

  ALTER VERTEX Company ADD INDEX company_name ON (name);

  ALTER VERTEX University ADD INDEX university_name ON (name);

  ALTER VERTEX Tag ADD INDEX tag_name ON (name);

  ALTER VERTEX TagClass ADD INDEX tagclass_name ON (name);

}


RUN GLOBAL SCHEMA_CHANGE JOB addIndex

CREATE GRAPH ldbc_snb (*)
```