1 Business Intelligence Workload

The Business Intelligence (BI) workload is the SNB's analytical (OLAP) workload. As such, it defines complex read queries that touch a significant portion of the data (see Section 1.4). Additionally, it defines daily batches of updates over a 33-day period (see Section 1.5 for inserts and Section 1.6 for deletes).

Related Publications

The BI workload was published in PVLDB 2022 [17].

Related Software Components

- Datagen (Spark-based): `https://github.com/ldbc/ldbc_snb_datagen_spark`
- Driver and reference implementations: `https://github.com/ldbc/ldbc_snb_bi`

1.1 Overview

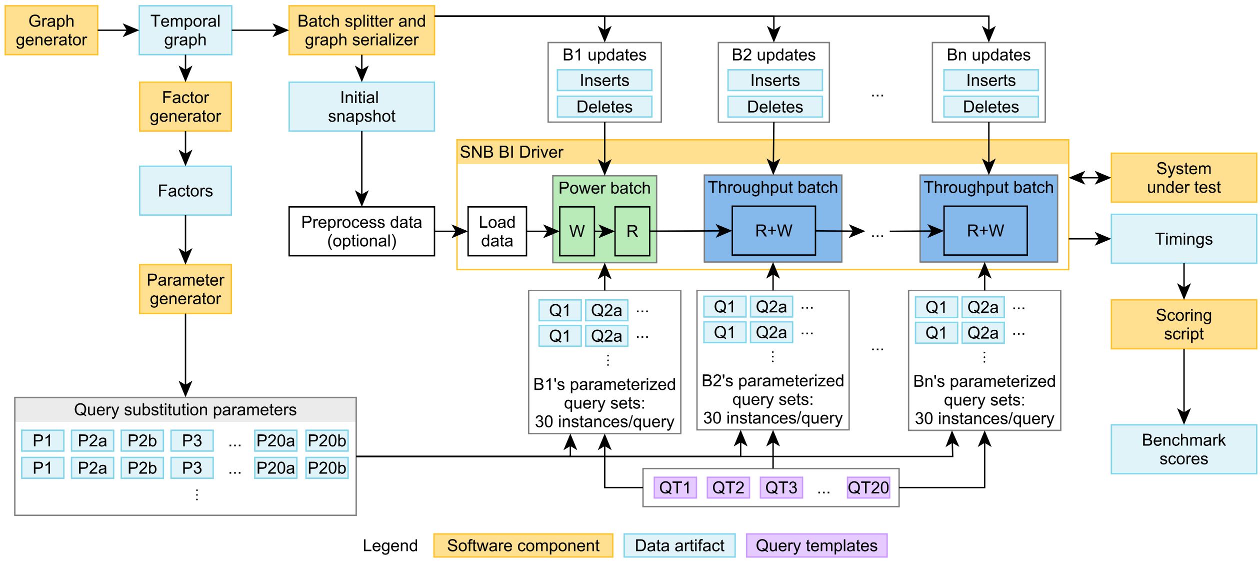


Figure 1.1: Main software components and data artifacts of the benchmark and their connection to the workflow executed by the BI benchmark driver.

An overview of the BI workload is shown in Figure 1.1. The rules for auditing workload implementations are given in **??**.

1.2 Read Query Templates

SNB BI consists of 20 parameterized *read query templates*, referred to as *queries*. These search for graph patterns (often implying join-heavy operations on many-to-many edges), traverse hierarchies, and compute cheapest paths (a.k.a. weighted shortest paths). Additionally, they include filtering, grouping, aggregation, and sorting operators. While all queries explore a large portion of the graph, they only return the top-k (typically 20 or 100) results, keeping their result sizes compact to avoid emphasizing the client-server network protocol's role in the benchmark [13].

1.2.1 Choke Point-Based Design Methodology

LDBC's query design process relies on the use of *choke points* (**??**), i.e. challenging aspects of query processing. SNB BI includes 38 choke points divided into 9 categories: aggregation performance, join performance, data access locality, expression calculation, correlated subqueries, parallelism and concurrency, graph specifics, language features, and update operations. Their coverage is shown in **??**. In the following, we discuss two challenges that are particularly prevalent in graph workloads.

1.2.1.1 Explosive and redundant multi-joins

In recent years it has become clear that graph pattern matching, or equivalent multi-join queries over many-to-many relationships, typically generate very large intermediate results when executed with traditional join algorithms. This is especially the case for cyclical join graphs (corresponding to cyclic graph queries). It was proven in theory [11] and shown in practice [18, 9, 5] that "worst-case optimal" *multi*-join algorithms can avoid these large intermediates and outperform traditional joins. Following this, there has been increased attention on *redundancy* in join results (even when produced by worst-case optimal joins), which can be eliminated using *factorized* query processing techniques [2, 12, 8]. Graph pattern matching queries that contain large join patterns will trigger these phenomena.

1.2.1.2 Expressive path finding

SNB BI contains queries that require an efficient implementation of shortest path finding between many pairs. Expressing such queries requires a query language which supports either path finding or recursion. The underlying system implementation must then handle this with an optimized execution strategy, as recursing to try all paths will not scale. As some of this path finding includes on-the-fly computed edges (joins) between nodes, the queries can benefit from *path expressions*, as proposed in Oracle's PGQL language [15] and as part of the GQL and SQL/PGQ languages [3]. The path finding required by SNB BI not only tests connectivity (as supported in SPARQL), but also requires returning the *cheapest cost* along weighted paths (necessitating SPARQL extensions [10]).

1.2.2 Analysis of Selected Queries

In order to defeat trivializing complex query performance by query caching, benchmarks can use both frequent updates (which require invalidating caches or maintaining cached intermediates) as well as parameterized query templates. The BI workload features update batches, so parametrized *read query templates* are necessary to guard against this between the batches. In this section, we analyze four read query templates.

Notation: We denote the query parameters with the `$` symbol and discuss their generation in Section 1.3.

1.2.2.1 Q11: Friend triangles

`BI 11` imposes two key difficulties. First, systems should efficiently filter the knows edges based on the location of their endpoint Persons (Country `$country`) and the date range. Second, given a large number of knows edges even after filtering, efficient enumeration of `personA`–`personB`–`personC` triangles (a cyclic subgraph query) requires worst-case optimal multi-joins.

1.2.2.2 Q14: International dialog

`BI 14` imposes different challenges depending on whether Countries `$country1` and `$country2` are correlated or anti-correlated (Section 1.3.3.1). For the ranking, *top-k pushdown* can be exploited: once a result for a City in `$country1` is obtained, extra restrictions in a selection can be added based on the value of this element. As the score of two Persons does not depend on any query parameters, precomputing and maintaining it as an attribute on the knows edge can be beneficial.

1.2.2.3 Q18: Friend recommendation

BI 18 is inspired by Twitter's recommendation algorithm [7]. Implementations of this query can exploit factorization: systems can count the number of mutual friends without explicitly enumerating all <person1, personM, person2> tuples.

1.2.2.4 Q20: Recruitment

BI 20 performs *graph projection* [1]. Instead of materializing this graph in the database, systems may represent it using a compact in-memory structure such as CSR (Compressed Sparse Row) [16]. To perform the cheapest path computation, a single-source shortest path algorithm (starting from `$person2`), such as Dijkstra's algorithm, can be used. As the projected graph is independent of query parameters, precomputing and maintaining it can be beneficial.

1.3 Parameter Curation for BI Queries

1.3.1 The Need for Parameter Curation

A disadvantage of executing the same read query template with different parameters is that the intermediate results and runtimes can be severely influenced by the parameter values. This is particularly the case in SNB BI with its explosive joins, skewed out-degrees, skewed value distributions, correlated value distributions, and structural correlations. Moreover, the updates (including cascading deletes) can significantly change the portion of the graph reached by the same query executed at different times. In order to keep query performance understandable we need to actively *curate* parameters, such that different parameters executed at different logical times still lead to stable and, therefore, understandable results. We achieve this through *parameter curation* [6, 4], a data mining process of looking for parameter values with suitably similar characteristics.

1.3.2 Parameter Generation Steps

Our parameter curation process is a two-step process: we first generate *factors* followed by the *parameters* (Figure 1.1). These components are executed for each scale factor and are independent of the serialization format/layout of the data set.

1.3.2.1 Factor Generator

The factor generator produces 21 *factor tables* containing summary statistics from the temporal graph, e.g. the number of Persons per City or the number of Messages per day for each Tag.

1.3.2.2 Parameter Generation

To find suitable substitution parameters that (presumably) lead to the same amount of data access and thus similar runtimes, we first identify the factor table containing the summary statistics of the query's parameters. For example, Q14's template uses the parameters Country `$country1` and Country `$country2`. Therefore, we use the `countryPairsNumFriends` factor table which contains `$country1`, `$country2` pairs and the number of friendships on Person lives in `$country1` and the other lives in `$country2`. Using this table, we select the pth percentile from the distribution as the *anchor*, then rank the rest of the distribution based on their absolute difference from the anchor and take the top-k values. We shuffle the values using a hash function to avoid introducing artificial locality, where e.g. subsequent queries start in nodes from the same ID range. Listing 1.1 shows the SQL query implementing the parameter generation for Q14*a*.

1.3.3 Parameter Curation for Graph Queries

We discuss two parameter curation cases that are particularly important in graph data management.

1.3.3.1 Correlated vs. Anti-Correlated Parameters

Our parameter curation provides a straightforward way of selecting start entities which are affected by (structural or attribute-level) correlation vs. anti-correlation: corresponding parameters can be found by selecting a high vs. low percentile as the anchor in the parameter generation query. For example, for Q14 (Section 1.4), we selected variant a to $p = 0.98$ (correlated) and variant b to $p = 0.03$ (anti-correlated).

1.3.3.2 Path Queries

SNB BI queries Q15, Q19, Q20 include cheapest path finding queries computed between given (sets of) Persons. These queries are particularly challenging for parameter curation: if there is no path between the two endpoints, query runtimes are significantly higher as the search has to traverse an entire connected component to ensure that no path exists. Moreover, the presence of a path between two nodes *at a given time* does not guarantee that it will always exist during the benchmark execution as deletions can render the endpoints of a path unreachable.

1.3.4 Query Variants

12 queries have a single variant, while 8 queries have two variants, yielding a total of 28 query variants. As a rule of thumb, variants a are expected to produce a longer runtime while variants b are expected to be simpler. Variants of Q2, Q8, Q16 are parametrized with a flashmob vs. a non-flashmob date. Variants of Q14 and Q19 select correlated vs. non-correlated Countries/Cities. Q10's variants differ in degree (a start Person with an average number of friends vs. only a few friends), while Q15's variants have different path lengths and time intervals (4 hops and one week vs. 2 hops and one month). Q20a selects endpoints where it is guaranteed that *no path exists*, while Q20b selects ones where there is guaranteed that a path exists.

1.3.5 Scalability and Reproducibility

1.3.5.1 Scalability

The *factor generator* is part of the SNB Datagen and runs after the *temporal graph* has been created. It is implemented in Spark for distributed execution. While its computations use expensive, aggregration-heavy queries, the derived factor tables are *compact*, e.g. SF10 000 has only 20 GiB of factors in compressed Parquet format, the equivalent of approximately 100 GiB in CSV format, i.e. 1% of the total data set size. The *parameter generator* queries are executed in DuckDB [14], which supports vertical scalability and is capable of running the parameter generation for SF10 000 using less than 512 GiB memory.

```sql
SELECT country1, country2
FROM (
  SELECT
    country1,
    country2,
    abs(frequency - (
      SELECT percentile_disc(0.98) WITHIN GROUP (ORDER BY frequency) AS anchor FROM countryPairsNumFriends
    )) AS diff
  FROM countryPairsNumFriends
  ORDER BY diff, country1, country2
)
ORDER BY md5(concat(country1, country2))
```

```
LIMIT 50
```

Listing 1.1: Parameter generation SQL query for Q14*a*.

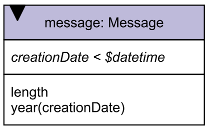
1.3.5.2 Reproducibility

It is important to guarantee that the parameter curation process is reproducible. To this end, we leverage that the Datagen and, consequently, the factor generator are reproducible. To ensure that the parameter generation queries yield deterministic results we define a total ordering in each query. To provide deterministic shuffling we base the ordering on MD5 hashes (instead of the actual attribute values), see Listing 1.1.

1.4 Reads

BI / read / 1

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 1
title	Posting summary
pattern	

description

Given a `$datetime`, find all Messages created before that moment. Group them by a 3-level grouping:

1. by year of creation
2. for each year, group into Message types: is Comment or not
3. for each year-type group, split into four groups based on length of their content
   - `0`: $0 \le$ length < 40 (short)
   - `1`: $40 \le$ length < 80 (one liner)
   - `2`: $80 \le$ length < 160 (tweet)
   - `3`: $160 \le$ length (long)

params

1	`$datetime`	DateTime

result

1	`year`	32-bit Integer	R	`year(message.creationDate)`
2	`isComment`	Boolean	M	`True` for Comments, `False` for Posts
3	`lengthCategory`	32-bit Integer	C	`0` for short, `1` for one-liner, `2` for tweet, `3` for long
4	`messageCount`	64-bit Integer	A	Total number of Messages in that group
5	`averageMessageLength`	32-bit Float	A	Average length of the Message content in that group
6	`sumMessageLength`	64-bit Integer	A	Sum of all Message content lengths
7	`percentageOfMessages`	32-bit Float	A	Number of Messages in group as a percentage of all messages created before the given date

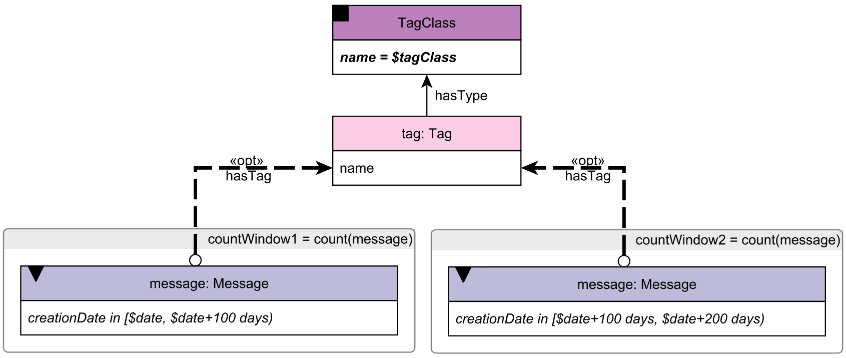
sort

1	`year`	↓	
2	`isComment`	↑	`False` < `True`, i.e. Posts come first and Comments second
3	`lengthCategory`	↑	

limit	n/a
CPs	1.2, 3.2, 4.1, 4.2, 8.5

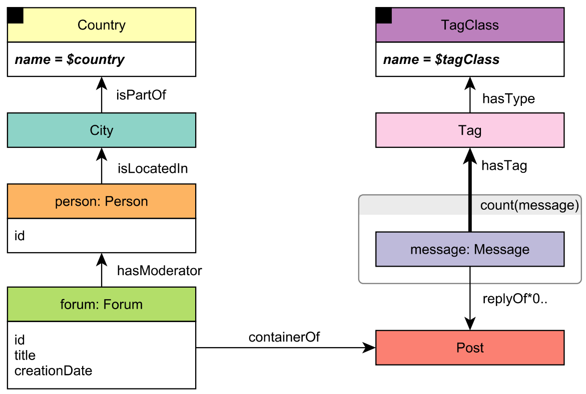
BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

BI / read / 2

query	BI / read / 2
title	Tag evolution
pattern	
description	Find the Tags under a given `$tagClass` that were used in Messages during in the 100-day time window starting at `$date` and compare it with the 100-day time window that follows. For the Tags and for both time windows, compute the count of Messages.
params	1 · `$date` · Date · Based on the creation day – TagClass – number of Messages factor table: (a) A flashmob date (b) A non-flashmob date 2 · `$tagClass` · Long String · For both (a) and (b), TagClasses with a similar amount of Messages are selected
result	1 · `tag.name` · Long String · R 2 · `countWindow1` · 32-bit Integer · A · Occurrences of the `tag` during the first time window 3 · `countWindow2` · 32-bit Integer · A · Occurrences of the `tag` during the second time window 4 · `diff` · 32-bit Integer · A · Absolute difference of `countWindow1` and `countWindow2`
sort	1 · `diff` · ↓ 2 · `tag.name` · ↑
limit	100
CPs	2.4, 3.1, 3.2, 4.1, 4.2, 4.3, 5.3, 6.1, 8.2, 8.5

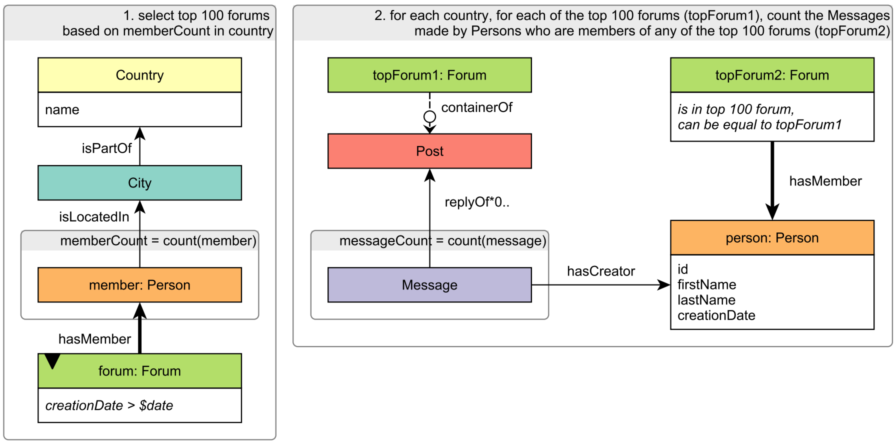
BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 3

<table>
<tr><td>query</td><td>BI / read / 3</td></tr>
<tr><td>title</td><td>Popular topics in a country</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>Given a <code>$tagClass</code> and a <code>$country</code>, find all the Forums created in the given <code>$country</code>, containing at least one Message with Tags belonging directly to the given <code>$tagClass</code>, and count the Messages by the Forum which contains them.
The location of a Forum is identified by the location of the Forum's moderator.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td><code>$tagClass</code></td><td>Long String</td><td>TagClasses with a similar amount of Messages are selected</td></tr>
<tr><td>2</td><td><code>$country</code></td><td>Long String</td><td>Big Countries are selected</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td><code>forum.id</code></td><td>ID</td><td>R</td><td></td></tr>
<tr><td>2</td><td><code>forum.title</code></td><td>Long String</td><td>R</td><td></td></tr>
<tr><td>3</td><td><code>forum.creationDate</code></td><td>DateTime</td><td>R</td><td></td></tr>
<tr><td>4</td><td><code>person.id</code></td><td>ID</td><td>R</td><td></td></tr>
<tr><td>5</td><td><code>messageCount</code></td><td>32-bit Integer</td><td>A</td><td></td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td><code>messageCount</code></td><td>↓</td><td></td></tr>
<tr><td>2</td><td><code>forum.id</code></td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>1.1, 1.2, 1.3, 2.1, 2.2, 2.4, 3.3, 8.2</td></tr>
</table>

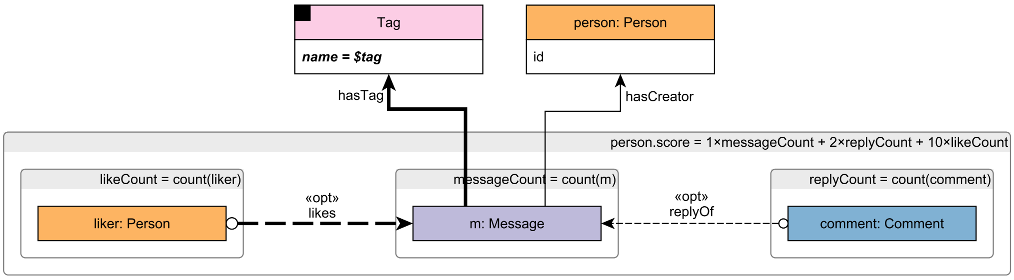
BI / read / 4

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 4
title	Top message creators by country
pattern	
description	Find the most popular Forums by Country, where the popularity of a Forum is measured by the number of members that Forum has from a given Country and the Forum was created after a given `$date`. Calculate the top 100 most popular Forums. If a Forum is popular in multiple countries, it should only be calculated once with its largest membership. In case of a tie, the Forum with the smaller id value should be selected. For each member Person of the 100 most popular Forums, count the number of Messages (`messageCount`) they made in any of those (most popular) Forums. Also include those member Persons who have not posted any Messages (have a `messageCount` of 0).
params	1 | `$date` | Date | Selected from the first 30 days of the network
result	1 | `person.id` | ID | R 2 | `person.firstName` | String | R 3 | `person.lastName` | String | R 4 | `person.creationDate` | DateTime | R 5 | `messageCount` | 32-bit Integer | A
sort	1 | `messageCount` | ↓ 2 | `person.id` | ↑
limit	100
CPs	1.2, 1.3, 2.1, 2.2, 2.3, 2.4, 3.3, 5.3, 6.1, 8.2, 8.4

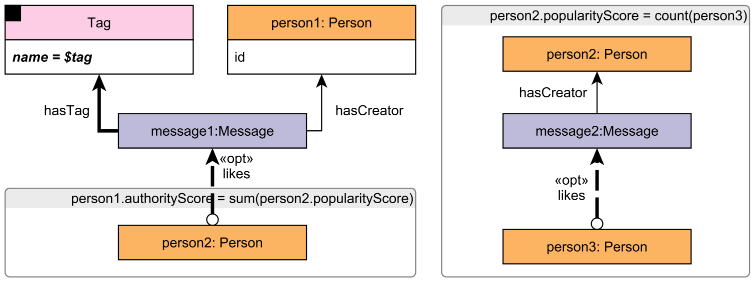
BI / read / 5

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 5
title	Most active posters of a given topic
pattern	
description	Get each Person (person) who has created a Message (message) with a given $tag (direct relation, not transitive). Considering only these Messages, for each Person node: • Count its Messages (messageCount). • Count likes (likeCount) to its Messages. • Count Comments (replyCount) in reply to its Messages. The score is calculated according to the following formula: $1 \times$ messageCount $+ 2 \times$ replyCount $+ 10 \times$ likeCount.
params	1 | $tag | Long String | Tags with a similar amount of Messages are selected. To avoid caching, different Tags should be used than the ones in Q6 and Q7.
result	1 | person.id | ID | R 2 | replyCount | 32-bit Integer | A 3 | likeCount | 32-bit Integer | A 4 | messageCount | 32-bit Integer | A 5 | score | 32-bit Integer | A
sort	1 | score | ↓ 2 | person.id | ↑
limit	100
CPs	1.2, 2.3, 2.6, 8.2

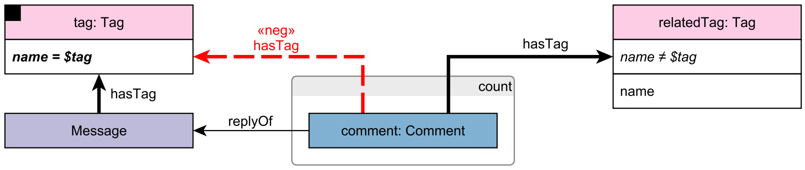
BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 6

<table>
<tr><td>query</td><td>BI / read / 6</td></tr>
<tr><td>title</td><td>Most authoritative users on a given topic</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>Given a $tag, find all Persons (person1) that ever created a Message with the $tag. For each of these Persons (person1) compute their “authority score” as follows:
• The “authority score” is the sum of “popularity scores” of the Persons (person2) that liked any of that Person’s Messages with the given $tag (same criterion as for message1).
• A Person’s (person2) “popularity score” is defined as the total number of likes (by any Person person3) on any of their Messages (message2).</td></tr>
<tr><td>params</td><td><table><tr><td>1</td><td>$tag</td><td>Long String</td><td>Tags with a similar amount of Messages are selected. To avoid caching, different Tags should be used than the ones in Q5 and Q7.</td></tr></table></td></tr>
<tr><td>result</td><td><table><tr><td>1</td><td>person1.id</td><td>ID</td><td>R</td><td></td></tr><tr><td>2</td><td>authorityScore</td><td>32-bit Integer</td><td>A</td><td></td></tr></table></td></tr>
<tr><td>sort</td><td><table><tr><td>1</td><td>authorityScore</td><td>↓</td><td></td></tr><tr><td>2</td><td>person1.id</td><td>↑</td><td></td></tr></table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.3, 2.6, 3.3, 6.1, 8.2</td></tr>
<tr><td>relevance</td><td>Computing the authority scores might involve computing the popularity score for the same Person multiple times. Implementations are advised to avoid such redundant computations.</td></tr>
</table>

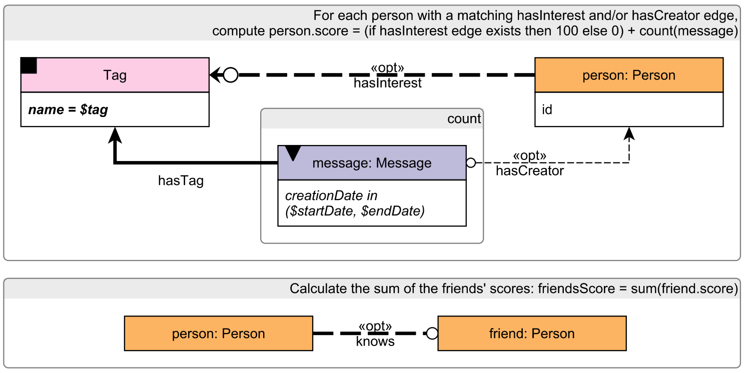
BI / read / 7

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 7
title	Related topics
pattern	
description	Find all Messages that have a given `$tag`. Find the related Tags attached to (direct) reply Comments of these Messages, but only of those reply Comments that do not have the given `$tag`. Group the related Tags by name, and get the count of replies in each group.
params	1 | `$tag` | Long String | Tags with a similar amount of Messages are selected. To avoid caching, different Tags should be used than the ones in Q5 and Q6.
result	1 | `relatedTag.name` | Long String | R 2 | `count` | 32-bit Integer | A
sort	1 | `count` | ↓ 2 | `relatedTag.name` | ↑
limit	100
CPs	1.4, 3.3, 5.2, 8.1

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 8

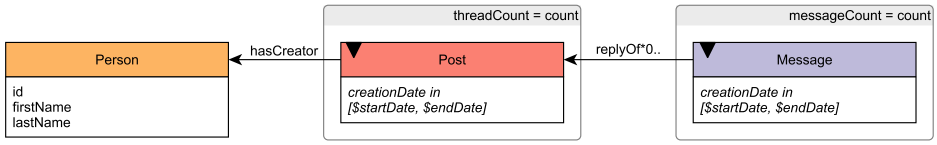
<table>
<tr><td>query</td><td>BI / read / 8</td></tr>
<tr><td>title</td><td>Central person for a tag</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>Given a $tag, find all Persons that are interested in the $tag and/or have written a Message (Post or Comment) with a creationDate after a given $startDate and that has a given $tag. For each Person, compute the score as the sum of the following two aspects:

- 100, if the Person has this $tag as their interest, or 0 otherwise
- number of Messages by this Person with the given $tag

Also, for each Person, compute the sum of the score of the Person's friends (friendsScore).</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>$tag</td><td>Long String</td><td>Tags with a similar amount of Messages are selected</td></tr>
<tr><td>2</td><td>$startDate</td><td>Date</td><td>(a): A range during which a flashmob event happened (it should yield at least a 5× difference)
(b): A regular range (does not include a flashmob event)</td></tr>
<tr><td>3</td><td>$endDate</td><td>Date</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>person.id</td><td>ID</td><td>R</td><td></td></tr>
<tr><td>2</td><td>score</td><td>32-bit Integer</td><td>A</td><td></td></tr>
<tr><td>3</td><td>friendsScore</td><td>32-bit Integer</td><td>A</td><td>The sum of the score of the person's friends</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>score + friendsScore</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>person.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.1, 2.3, 3.2, 5.3, 8.2, 8.4, 8.5</td></tr>
<tr><td>relevance</td><td>Similarly to BI 16, there are two major ways to compute this query: (1) creating an induced subgraph of the interested Persons and their friends and performing the scoring on this graph or (2) performing the scoring without creating an induced subgraph and scoring the friends of a Person on-the-fly. The first approach is more efficient as it avoids redundant computations, however, specifying it needs support for composable graph queries.</td></tr>
</table>

BI / read / 9

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 9
title	Top thread initiators
pattern	
description	For each Person, count the number of Posts they created in the time interval [$startDate, $endDate] (equivalent to the number of threads they initiated) and the number of Messages in each of their (transitive) reply trees, including the root Post of each tree. When calculating Message counts only consider Messages created within the given time interval. Return each Person, number of Posts they created, and the count of all Messages that appeared in the reply trees (including the Post at the root of tree).

params

1	$startDate	Date	Selected around the same date
2	$endDate	Date	80-100 days after the $startDate

result

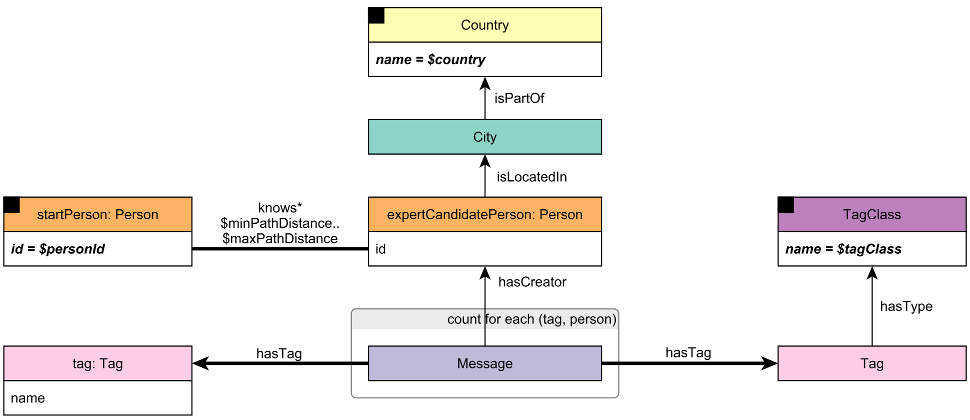
1	person.id	ID	R	
2	person.firstName	String	R	
3	person.lastName	String	R	
4	threadCount	32-bit Integer	A	The number of Posts created by that Person (the number of threads initiated)
5	messageCount	32-bit Integer	A	The number of Messages created in all the threads this Person initiated

sort

1	messageCount	↓	
2	person.id	↑	

limit	100
CPs	1.2, 2.2, 2.3, 2.6, 3.2, 7.2, 7.3, 7.4, 8.1, 8.5

BI / read / 10

query	BI / read / 10
title	Experts in social circle
pattern	
description	Given a Person `startPerson` with ID `$personID`, find all other Persons (`expertCandidatePerson`) that live in a given `$country` and are connected to the `startPerson` on a *shortest path* with length in range `[$minPathDistance, $maxPathDistance]` through the knows relation. For each of these `expertCandidatePerson` nodes, retrieve all of their Messages that contain at least one Tag belonging to a given `$tagClass` (direct relation not transitive). For each Message, retrieve all of its Tags. Group the results by Persons and Tags, then count the Messages by a certain Person having a certain Tag.

params

1	`$personId`	ID	(a) Persons with an average degree of knows edges are selected (b) Persons who have only one friend and that Person has two friends in total (including the original Person)
2	`$country`	String	Select mid-sized Countries
3	`$tagClass`	Long String	TagClasses with a similar degree of hasType edges are selected
4	`$minPathDistance`	32-bit Integer	3
5	`$maxPathDistance`	32-bit Integer	4

result

1	`expertCandidatePerson.id`	ID	R	
2	`tag.name`	Long String	R	
3	`messageCount`	32-bit Integer	A	Number of Messages created by that Person containing that Tag

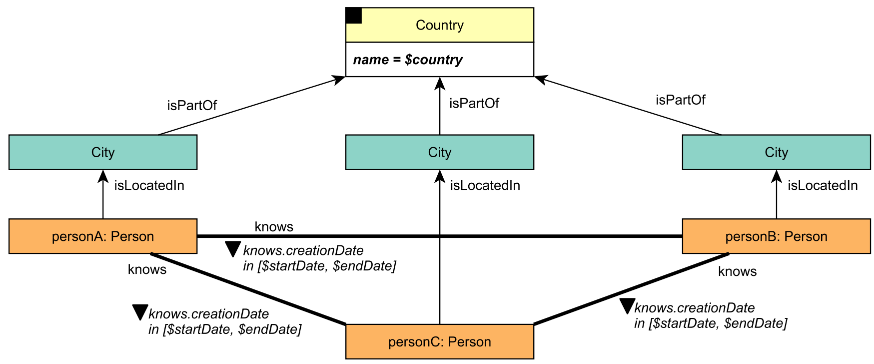
sort

1	`messageCount`	↓	
2	`tag.name`	↑	
3	`expertCandidatePerson.id`	↑	

limit	100
CPs	1.2, 1.3, 2.3, 2.4, 2.6, 3.3, 5.3, 7.1, 7.2, 7.3, 8.1, 8.6

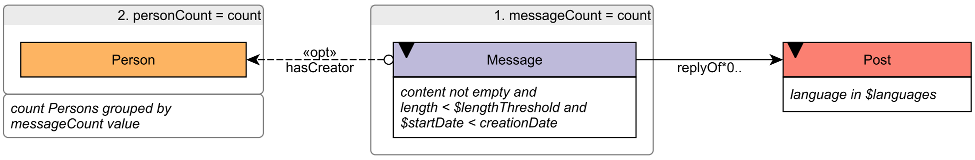
BI / read / 11

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

<table>
<tr><td>query</td><td>BI / read / 11</td></tr>
<tr><td>title</td><td>Friend triangles</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>For a given <code>$country</code>, count all the distinct triples of Persons such that:
• <code>personA</code> is friend of <code>personB</code>,
• <code>personB</code> is friend of <code>personC</code>,
• <code>personC</code> is friend of <code>personA</code>,
and these friendships were created in the range <code>[$startDate, $endDate]</code>.
Distinct means that given a triple t_1 in the result set R of all qualified triples, there is no triple t_2 in R such that t_1 and t_2 have the same set of elements.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>$country</td><td>Long String</td><td>Selected from the largest Countries (India, China)</td></tr>
<tr><td>2</td><td>$startDate</td><td>Date</td><td>Selected from a 30-day interval towards the end of the simulation time</td></tr>
<tr><td>3</td><td>$endDate</td><td>Date</td><td>Selected to yield around a 100-day interval</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>count</td><td>64-bit Integer</td><td>A</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>n/a</td></tr>
<tr><td>CPs</td><td>2.3, 2.5, 3.2</td></tr>
</table>

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 12

<table>
<tr><td>query</td><td>BI / read / 12</td></tr>
<tr><td>title</td><td>How many persons have a given number of messages</td></tr>
<tr><td>pattern</td><td>


</td></tr>
<tr><td>description</td><td>
For each Person, count the number of Messages they made (messageCount). Only count Messages with the following attributes:

- Its content is not empty (and consequently, the imageFile attribute is empty for Posts).
- Its creationDate is after $startDate (exclusive, equality is not allowed).
- Its length is below the $lengthThreshold (exclusive, equality is not allowed).
- It is written in any of the given $languages.
  - The language of a Post is defined by its language attribute.
  - The language of a Comment is that of the Post that initiates the thread where the Comment replies to.

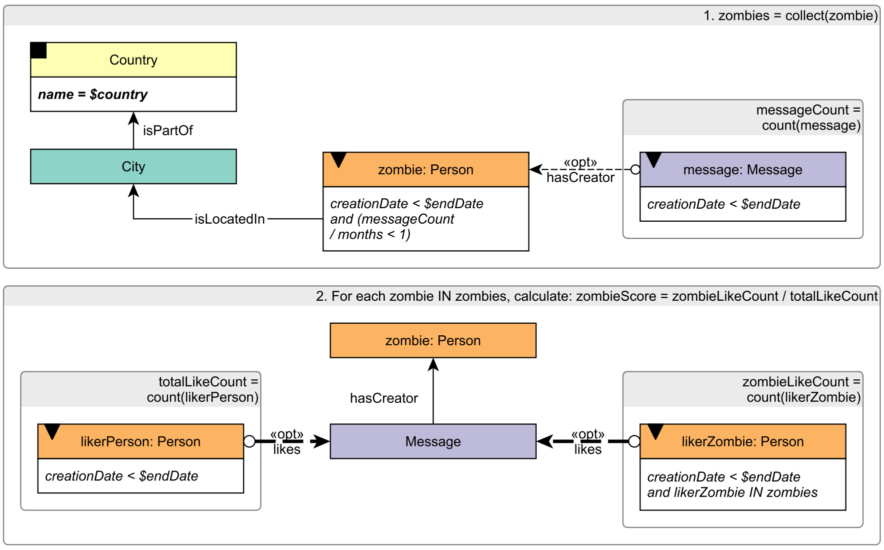
  The Post and Comments in the reply tree's path (from the Message to the Post) do not have to satisfy the constraints for content, length, and creationDate.

For each messageCount value, count the number of Persons with exactly messageCount Messages (with the required attributes).
</td></tr>
<tr><td>params</td><td>
<table>
<tr><td>1</td><td>$startDate</td><td>Date</td><td>Selected randomly from a 60-day interval.</td></tr>
<tr><td>2</td><td>$lengthThreshold</td><td>32-bit Integer</td><td>Balanced against startDate to filter around 30% of the Messages within a language and keep the variance low.
The selection of this parameter uses a factor table of bucketed Message lengths and creation dates.</td></tr>
<tr><td>3</td><td>$languages</td><td>{String}</td><td>Only the most frequently used languages</td></tr>
</table>
</td></tr>
<tr><td>result</td><td>
<table>
<tr><td>1</td><td>messageCount</td><td>32-bit Integer</td><td>A</td><td>Number of Messages created</td></tr>
<tr><td>2</td><td>personCount</td><td>32-bit Integer</td><td>A</td><td>Number of Persons with messageCount Messages</td></tr>
</table>
</td></tr>
<tr><td>sort</td><td>
<table>
<tr><td>1</td><td>personCount</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>messageCount</td><td>↓</td><td></td></tr>
</table>
</td></tr>
<tr><td>limit</td><td>n/a</td></tr>
<tr><td>CPs</td><td>1.1, 1.2, 1.4, 2.6, 3.2, 4.2, 4.3, 8.1, 8.2, 8.3, 8.4, 8.5</td></tr>
</table>

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 13

<table>
<tr><td>query</td><td>BI / read / 13</td></tr>
<tr><td>title</td><td>Zombies in a country</td></tr>
<tr><td>pattern</td><td>



</td></tr>
<tr><td>description</td><td>

Find zombies within the given `$country`, and return their zombie scores. A `zombie` is a Person created before the given `$endDate`, which has created an average of `[0, 1)` Messages per month, during the time range between profile's `creationDate` and the given `$endDate`. The number of months spans the time range from the `creationDate` of the profile to the `$endDate` with partial months on both end counting as one month (e.g. a `creationDate` of Jan 31 and an `$endDate` of Mar 1 result in 3 months).
For each `zombie`, calculate the following:

- `zombieLikeCount`: the number of likes received from other zombies.
- `totalLikeCount`: the total number of likes received.
- `zombieScore`: `zombieLikeCount` / `totalLikeCount`. If the value of `totalLikeCount` is 0, the `zombieScore` of the `zombie` should be 0.0.

For both `zombieLikeCount` and `totalLikeCount`, only consider likes received from profiles that were created before the given `$endDate`.

</td></tr>
<tr><td>params</td><td>

1	`$country`	Long String	Selected from the largest Countries (India, China)
2	`$endDate`	Date	Selected from the last days of the initial data set

</td></tr>
<tr><td>result</td><td>

1	`zombie.id`	ID	R	
2	`zombieLikeCount`	32-bit Integer	A	
3	`totalLikeCount`	32-bit Integer	A	
4	`zombieScore`	32-bit Float	A	Determined as `zombieLikeCount` / `totalLikeCount`

</td></tr>
<tr><td>sort</td><td>

1	`zombieScore`	↓	
2	`zombie.id`	↑	

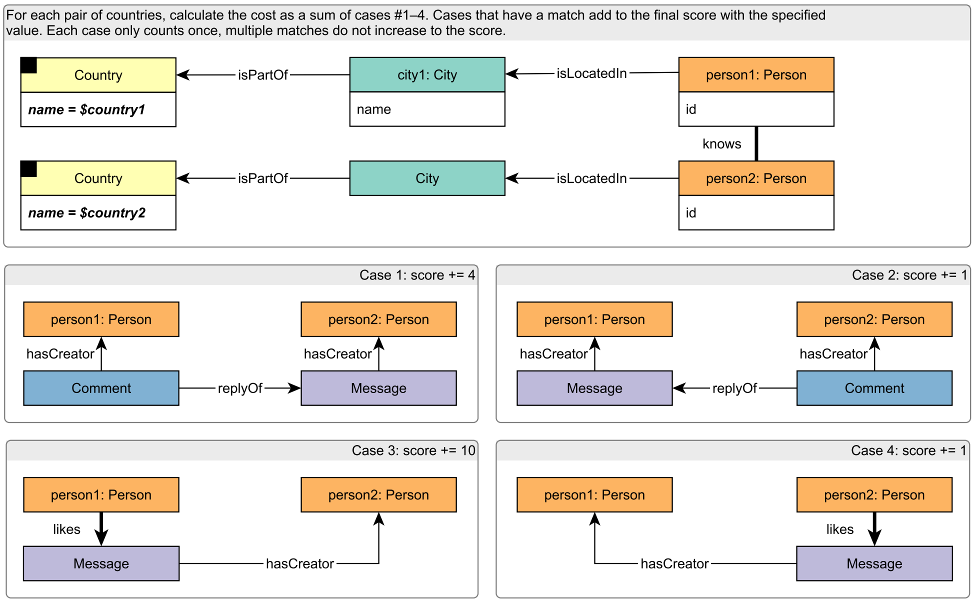
</td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.1, 2.3, 2.4, 2.6, 3.2, 3.3, 4.2, 5.1, 5.3, 8.2, 8.4, 8.5</td></tr>
</table>

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

BI / read / 14

query	BI / read / 14
title	International dialog

pattern



description

Consider all pairs of people (`person1, person2`) such that (1) they know each other, (2) one is located in a `City` of `$country1`, and (3) the other is located in a `City` of `$country2`. For each `City` of `$country1`, return the highest scoring pair. If there are multiple top-scoring pairs in a city, return the pair with the lowest (`person1.id`, `person2.id`) using a lexicographical ordering.
The score of a pair is defined as the sum of the subscores awarded for the following kinds of interaction. The initial value is `score = 0`.

1. `person1` has created a reply `Comment` to at least one `Message` by `person2`: `score += 4`
2. `person1` has created at least one `Message` that `person2` has created a reply to: `score += 1`
3. `person1` liked at least one `Message` by `person2`: `score += 10`
4. `person1` has created at least one `Message` that was liked by `person2`: `score += 1`

Consequently, the maximum score a pair can obtain is: `4 + 1 + 10 + 1 = 16`.

params

1	$country1	Long String	(a) Correlated with parameter `country2`, i.e. the `Countries` are close and there are many `Persons` knowing each other (b) Uncorrelated with parameter `country2`, i.e. the `Countries` are afar and there are few `Persons` knowing each other
2	$country2	Long String	

result

1	person1.id	ID	R	
2	person2.id	ID	R	
3	city1.name	Long String	R	
4	score	32-bit Integer	C	

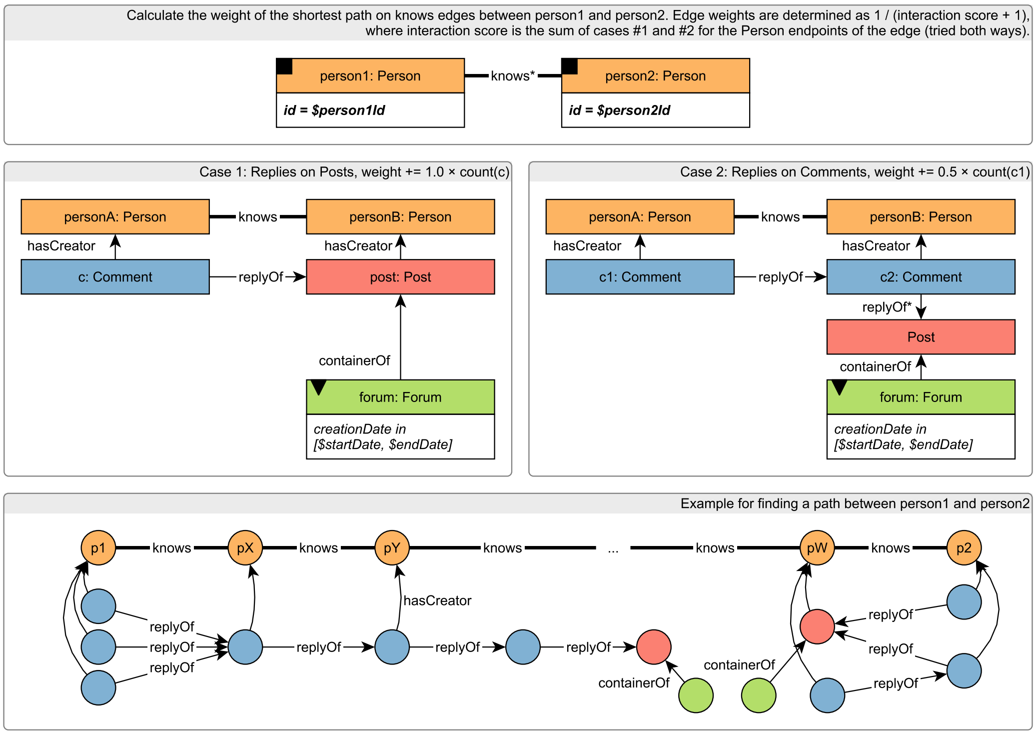
sort

1	score	↓	
2	person1.id	↑	
3	person2.id	↑	

limit	100
CPs	1.3, 1.4, 2.1, 3.1, 3.3, 5.1, 5.2, 5.3, 8.3, 8.4

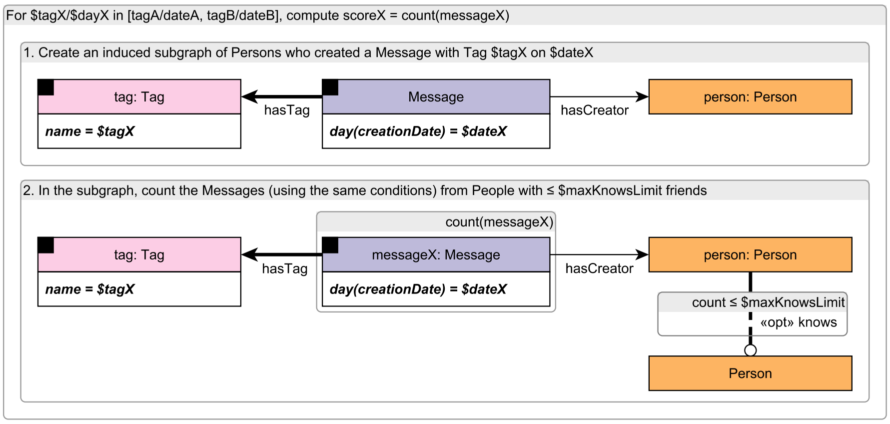
BI / read / 15

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

<table>
<tr><td>query</td><td>BI / read / 15</td></tr>
<tr><td>title</td><td>Trusted connection paths through forums created in a given timeframe</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>Given two Persons with IDs <code>$person1Id</code> and <code>$person2Id</code>, calculate the cost of the weighted shortest path between these two Persons, in the subgraph induced by the knows relationship. The interaction score of a knows edge is calculated based on the interactions of its Person endpoints:
• Every direct reply (by one of the Persons) to a Post (by the other Person) is 1.0 point.
• Every direct reply (by one of the Persons) to a Comment (by the other Person) is 0.5 points.
Only consider Messages that were created in a Forum that was created within the timeframe (interval) <code>[$startDate, $endDate]</code>. Note that for Comments, the containing Forum is that of the Post that the comment (transitively) replies to. Also note that interactions are counted both ways.
The weight for the shortest path algorithm is determined as $\frac{1}{\textit{interaction score}+1}$.
The result of the query is a single number, the cost of the weighted shortest path. If no such path exists, the query should return -1.0.</td></tr>
<tr><td>params</td><td>
<table>
<tr><td>1</td><td><code>$person1Id</code></td><td>ID</td><td>(a) <code>$person1Id</code> – <code>$person2Id</code> pair with a distance of 4 hops
(b) <code>$person1Id</code> – <code>$person2Id</code> pair with a distance of 2 hops</td></tr>
<tr><td>2</td><td><code>$person2Id</code></td><td>ID</td><td></td></tr>
<tr><td>3</td><td><code>$startDate</code></td><td>Date</td><td>(a) Small interval (approx. one week)
(b) Big interval (approx. one month)</td></tr>
<tr><td>4</td><td><code>$endDate</code></td><td>Date</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td>
<table>
<tr><td>1</td><td><code>weight</code></td><td>32-bit Float</td><td>C</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>n/a</td></tr>
<tr><td>CPs</td><td>1.2, 2.1, 2.2, 2.4, 3.3, 5.1, 5.3, 7.2, 7.3, 7.6, 7.7, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6</td></tr>
</table>

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 16

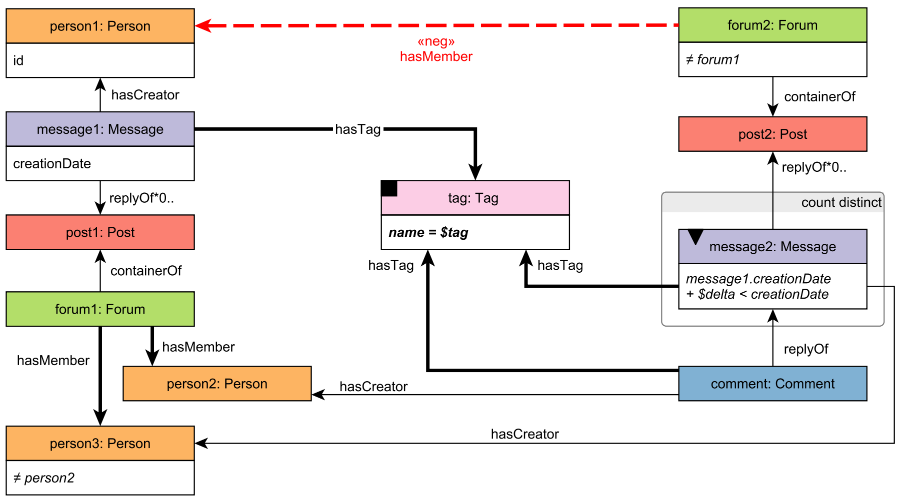
<table>
<tr><td>query</td><td>BI / read / 16</td></tr>
<tr><td>title</td><td>Fake news detection</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>Given two Tag/date pairs (<code>$tagA</code>/<code>$dateA</code> and <code>$tagB</code>/<code>$dateB</code>), for each pair <code>$tagX</code>/<code>$dateX</code>:

Create an induced subgraph between Persons where for each pair of Persons <code>person1</code>/<code>person2</code>, both have created a Message on the day of <code>$dateX</code> with Tag <code>$tagX</code>.
In the induced subgraph, only keep pairs of Persons who have at most <code>maxKnowsLimit</code> friends (in the induced subgraph).
For these Persons, count the number of Messages created on <code>$dateX</code> with Tag <code>$tagX</code>.

Return Persons who had at least one Messages for both <code>$tagA</code>/<code>$dateA</code> and <code>$tagB</code>/<code>$dateB</code> ranked by their total number of Messages (descending).</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>$tagA</td><td>Long String</td><td>(a) $tagA/$dateA, $tagB/$dateB are both selected to be a flashmob Tag/date combination
(b) $tagA/$dateA, $tagB/$dateB are both selected to be a non-flashmob Tag/date combination</td></tr>
<tr><td>2</td><td>$dateA</td><td>Date</td><td></td></tr>
<tr><td>3</td><td>$tagB</td><td>Long String</td><td></td></tr>
<tr><td>4</td><td>$dateB</td><td>Date</td><td></td></tr>
<tr><td>5</td><td>$maxKnowsLimit</td><td>32-bit Integer</td><td>Selected between 3 and 6</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>person.id</td><td>ID</td><td>R</td><td></td></tr>
<tr><td>2</td><td>messageCountA</td><td>32-bit Integer</td><td>A</td><td>Message count for $tagA/$dateA</td></tr>
<tr><td>3</td><td>messageCountB</td><td>32-bit Integer</td><td>A</td><td>Message count for $tagB/$dateB</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>messageCountA + messageCountB</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>person.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>5.3, 8.4, 8.5</td></tr>
<tr><td>relevance</td><td>There are two major ways to compute this query: (1) create the induced subgraph as suggested by the specification (either as a view or in materialized form), or (2) skip creating the induced subgraph and perform on-the-fly check for the number of friends (who also posted at least one Message with the given Tag on the given date). The latter approach is easier to express in systems which do not provide graph views but might result in redundant computations (the query engine might repeatedly check whether a Person has at least one Message that satifies the conditions).</td></tr>
</table>

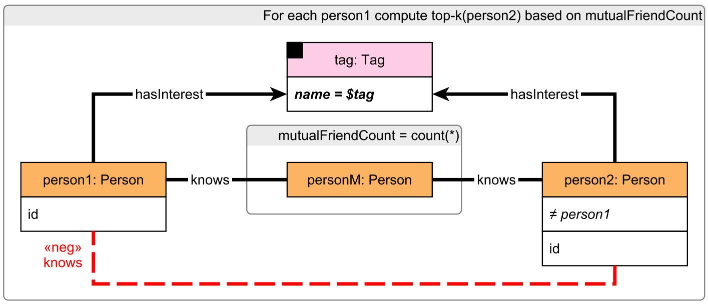
BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

BI / read / 17

<table>
<tr><td>query</td><td>BI / read / 17</td></tr>
<tr><td>title</td><td>Information propagation analysis</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>This query aims to identify instances of "information propagation" when a Person (person1) submits a Message (message1) with a given $tag to a Forum (forum1). This is read by other members of forum1, Persons person2 and person3 (who must be different Persons). Some time later (specified by the $delta parameter), these persons have a discussion with the same $tag in a different Forum (forum2) where person1 is not a member. The discussion consists of a Message (message2) by person2 and a direct reply Comment (comment) by person3.
Return IDs of person1 with the number of interactions their Messages (might have) caused.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>$tag</td><td>Long String</td><td>Tags with a similar amount of Messages are selected</td></tr>
<tr><td>2</td><td>$delta</td><td>32-bit Integer</td><td>Measured in hours, selected to be between 8 and 16 hours.</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>person1.id</td><td>ID</td><td>R</td><td></td></tr>
<tr><td>2</td><td>messageCount</td><td>32-bit Integer</td><td>A</td><td></td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>messageCount</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>person1.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>10</td></tr>
<tr><td>CPs</td><td>2.1, 2.3, 2.5, 2.6, 8.1</td></tr>
</table>

BI / read / 18

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

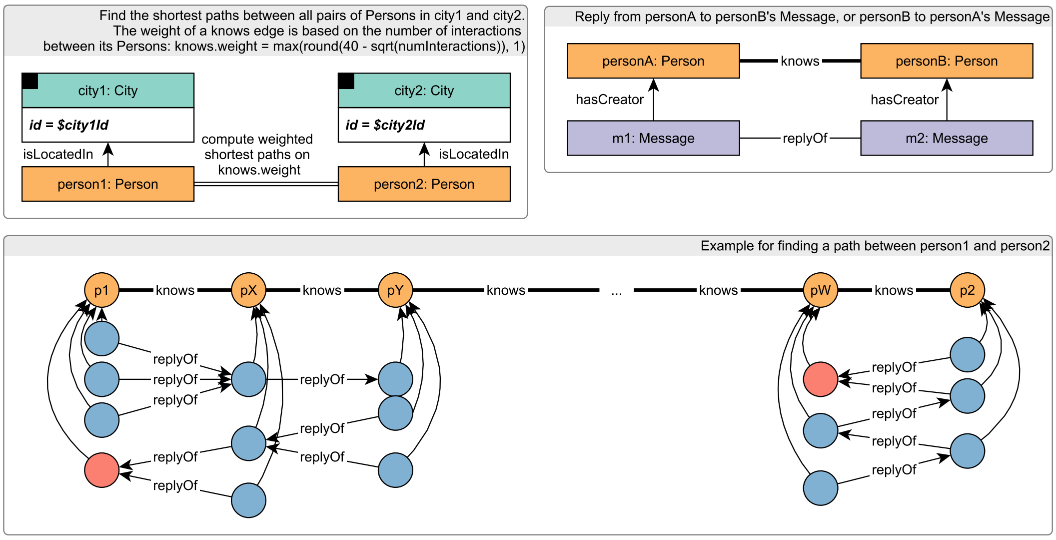
<table>
<tr><td>query</td><td>BI / read / 18</td></tr>
<tr><td>title</td><td>Friend recommendation</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>For a given <code>$tag</code>, for each <code>person1</code> interested in <code>$tag</code>, recommend new friends (<code>person2</code>) who

do not yet know <code>person1</code>
have at least one mutual friend with <code>person1</code>
are also interested in <code>$tag</code>.

Rank Persons <code>person2</code> based on the number of mutual friends with <code>person1</code>.</td></tr>
<tr><td>params</td><td><table><tr><td>1</td><td><code>$tag</code></td><td>Long String</td><td>Tags with a similar amount of Persons are selected</td></tr></table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td><code>person1.id</code></td><td>ID</td><td>R</td><td></td></tr>
<tr><td>2</td><td><code>person2.id</code></td><td>ID</td><td>R</td><td></td></tr>
<tr><td>3</td><td><code>mutualFriendCount</code></td><td>32-bit Integer</td><td>A</td><td></td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td><code>mutualFriendCount</code></td><td>↓</td><td></td></tr>
<tr><td>2</td><td><code>person1.id</code></td><td>↑</td><td></td></tr>
<tr><td>3</td><td><code>person2.id</code></td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>2.5, 2.6, 8.1</td></tr>
</table>

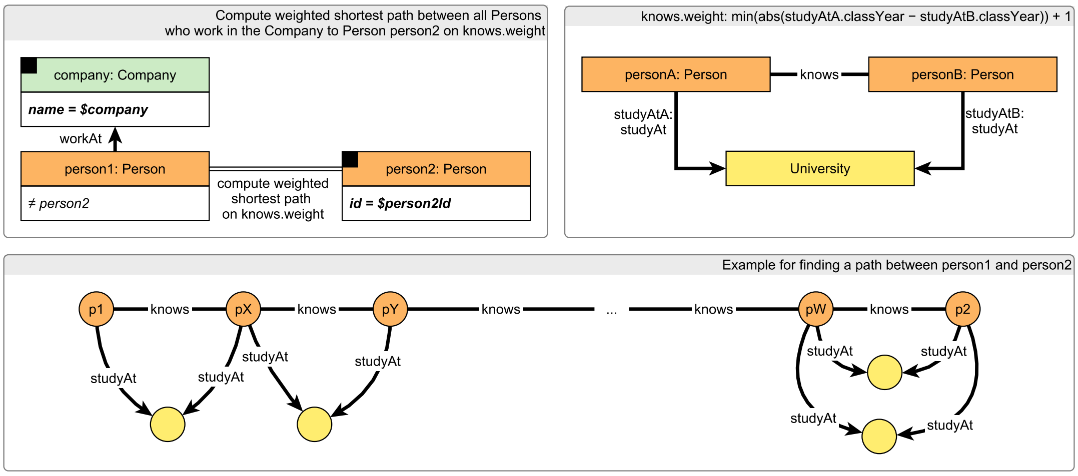
BI / read / 19

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

<table>
<tr><td>query</td><td>BI / read / 19</td></tr>
<tr><td>title</td><td>Interaction path between cities</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>description</td><td>Given two Cities with IDs <code>$city1Id</code>, <code>$city2Id</code>, find Persons <code>person1</code>, <code>person2</code> living in these Cities (respectively) with the cheapest interaction path between them.
The cheapest path is equivalent to the weighted shortest path. It is computed on a subgraph of the Person-knows-Person graph with the edge weights based on the number of interactions. An interaction is a direct reply Comments from one Person to Messages by the other Person. Only knows edges with at least one interaction between their endpoint Persons are considered. For these, the weight of a knows edge is defined as: $\max(\mathrm{round}(40 - \sqrt{numInteractions}), 1)$
If there are multiple pairs of people with cheapest paths that have the same total weight, return all of them.
Note: Interactions are counted both ways, e.g. if Alice knows Bob, Alice writes 2 reply Comments to Bob's Messages and Bob writes 3 reply Comments to Alice's Messages, their total number of interactions is 5 and the weight of the knows edge is 38.
Remark: Determinism is ensured by using square root followed by rounding. For all integers between 1 and 100 000, the square root's fractional part is more than 10e-5 from 0.5, where the rounding could be non-determinstic based on floating point inaccuracies. As 10e-5 is significantly larger than the machine epsilon of IEEE 754 floats (both 32- and 64-bit), the floating point inaccuracies have no chance to affect the derived integer edge weights.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td><code>$city1Id</code></td><td>ID</td><td>(a) Small Cities within the same Country
(b) Larger Cities from different Countries</td></tr>
<tr><td>2</td><td><code>$city2Id</code></td><td>ID</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td><code>person1.id</code></td><td>ID</td><td>R</td><td></td></tr>
<tr><td>2</td><td><code>person2.id</code></td><td>ID</td><td>R</td><td></td></tr>
<tr><td>3</td><td><code>totalWeight</code></td><td>32-bit Integer</td><td>C</td><td></td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td><code>person1.id</code></td><td>↑</td><td></td></tr>
<tr><td>2</td><td><code>person2.id</code></td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>n/a</td></tr>
<tr><td>CPs</td><td>3.3, 7.6, 7.7, 8.4, 8.6</td></tr>
<tr><td>relevance</td><td>To find the weighted shortest paths efficiently, the system can use e.g. a bidirectional Dijkstra algorithm. As the edge weights do not depend on any parameter, systems can pre-compute them (if they do not interleave reads and writes).</td></tr>
</table>

BI / read / 20

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 20
title	Recruitment
pattern	
description	Consider knows edges where the endpoint Persons attended the same University and set the weight of the edge to the absolute difference between the year of enrolment plus 1. If the Persons attended multiple universities, we select the smallest (`min`) value. Formally: $$w = \min_{\text{studyAt}_A, \text{studyAt}_B} \lvert \text{studyAt}_A.\text{classYear} - \text{studyAt}_B.\text{classYear} \rvert + 1$$ Given a `$company` and a Person `person2` with ID `$person2Id` (who is not working and has not worked at `$company`), find a different Person (`person1`) who works or at some point worked in `$company` and is reachable from `person2` through people who have studied together through the shortest weighted path. If there are multiple Person `person1` nodes with the same shortest path length, return all of them.
params	1 · `$company` · Long String · Companies with a similar number of employees (former or current) are selected 2 · `$person2Id` · ID · (a) There is guaranteed to be no path between any `person1` working at company and `person2` (b) There is guaranteed to be a 2-hop path between at least one `person1` working at company and person
result	1 · `person1.id` · ID · R 2 · `totalWeight` · 32-bit Integer · C
sort	1 · `totalWeight` · ↑ 2 · `person1.id` · ↑
limit	20
CPs	3.3, 7.6, 7.7, 7.8, 8.4, 8.6
relevance	To find the weighted shortest paths efficiently, the system can use e.g. a bidirectional Dijkstra algorithm. As the edge weights do not depend on any parameter, systems can pre-compute them (if they do not interleave reads and writes).

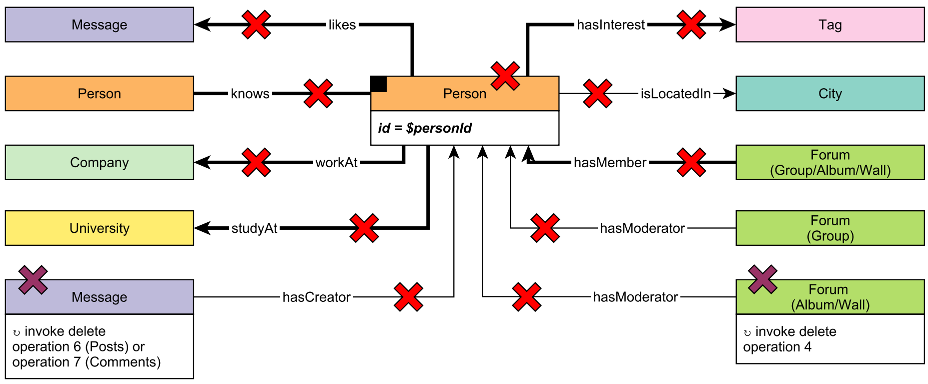
1.5 Insert Operations

Insert operations consist of individual inserts for each entity type. Implementations typically use the same format as the one for loading the initial snapshot of the data set.

1.6 Delete Operations

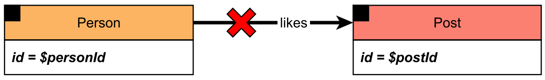
Updates / delete / 1

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 1
title	Remove person and its personal forums and message (sub)threads
pattern	
description	Remove a Person with ID `$personId` and its edges (isLocatedIn, studyAt, workAt, hasInterest, likes, knows, hasMember, hasModerator, hasCreator). Additionally, remove the Album and Wall Forums whose moderator is the Person and remove all Messages the Person has created in the rest of the Forums (Groups).
params	1 | `$personId` | ID
CPs	9.3, 9.4, 9.5
relevance	• Removal of a Person removes Forums of type "Walls" and "Albums" but not "Groups", which can continue if even the founder has left the network. For Groups, the hasModerator edge is deleted. We have discussed various approaches to appoint a new moderator, e.g. 1. choose member at random from the set of existing group members or 2. the member with the oldest group join date becomes the moderator. However, to keep the generator and the workload simple, currently no moderator is selected, leaving the group without a moderator. • Removal of a Person removes all Posts/Comments they are creator of this could result in the removal of a Comment in the middle of a thread.

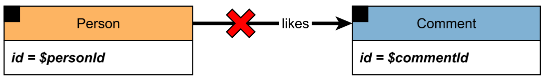
Updates / delete / 2

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 2
title	Remove post like
pattern	
description	Given a Person with ID `$personId` and a Post with ID `$postId`, remove the likes edge between them.
params	1 `$personId` ID 2 `$postId` ID
CPs	9.4
relevance	Removal of a likes edge is a rare event, e.g. people accidently liking a Post, this can be reflected by the relative frequency of the operation.

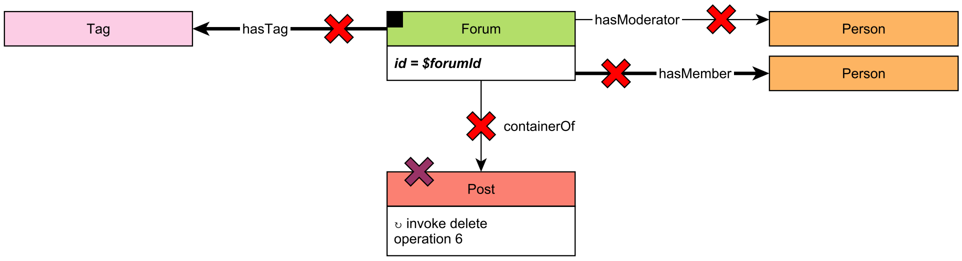
Updates / delete / 3

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 3
title	Remove comment like
pattern	
description	Given a Person with ID `$personId` and a Comment with ID `$commentId`, remove the likes edge between them.
params	1 `$personId` ID 2 `$commentId` ID
CPs	9.4
relevance	Removal of a likes edge is a rare event, e.g. people accidently liking a Comment, this can be reflected by the relative frequency of the operation.

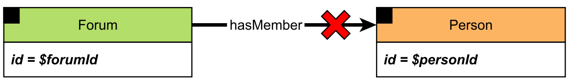
Updates / delete / 4

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 4
title	Remove forum and its content
pattern	
description	Remove a Forum with ID `$forumId` and its edges (hasModerator, hasMember, hasTag) and all Posts in the Forum (connected by containerOf edges) and their direct and transitive Comments.
params	1 `$forumId` ID
CPs	9.3, 9.4, 9.5
relevance	n/a

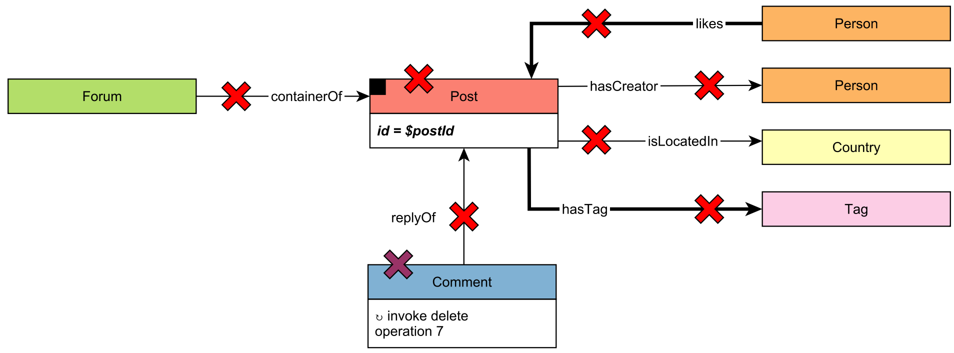
Updates / delete / 5

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 5
title	Remove forum membership
pattern	
description	Given a Forum with ID `$forumId` and a Person with ID `$personId`, remove the hasMember edge between them.
params	1 `$forumId` ID 2 `$personId` ID
CPs	9.4
relevance	n/a

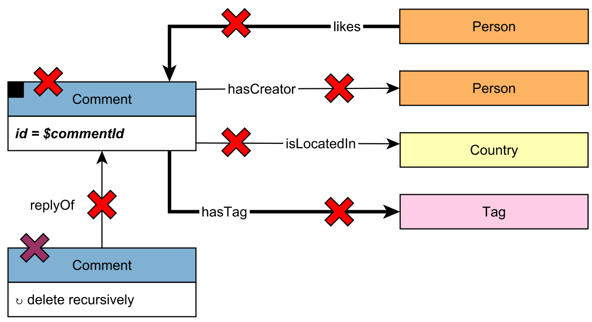
Updates / delete / 6

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 6
title	Remove post thread
pattern	
description	Remove a Post node with ID $postId and its edges (isLocatedIn, likes, hasCreator, hasTag, containerOf). Remove all replies to the Post and the connecting replyOf edges. In addition, remove all transitive reply Comments to the Post and their edges.
params	1 | $postId | ID
CPs	9.3, 9.4, 9.5
relevance	n/a

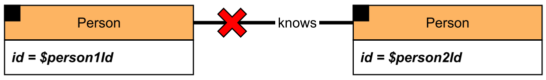
Updates / delete / 7

DEL 1 DEL 2 DEL 3 DEL 4 DEL 5 DEL 6 DEL 7 DEL 8

query	Updates / delete / 7
title	Remove comment subthread
pattern	
description	Remove a Comment node with ID $commentId and its *edges* (isLocatedIn, likes, hasCreator, hasTag). In addition, remove all replies to the Comment connected by replyOf and their *edges*.
params	1 | $commentId | ID
CPs	9.3, 9.4, 9.5
relevance	n/a

Updates / delete / 8

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	Updates / delete / 8
title	Remove friendship
pattern	
description	Given two Person nodes with IDs $person1Id and $person2Id, remove the knows edge between them.
params	1 | $person1Id | ID 2 | $person2Id | ID
CPs	9.4
relevance	n/a

Bibliography

[1] Renzo Angles et al. "G-CORE: A Core for Future Graph Query Languages". In: *SIGMOD*. ACM, 2018, pp. 1421–1432. DOI: 10.1145/3183713.3190654.

[2] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. "FDB: A Query Engine for Factorised Relational Databases". In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1232–1243. DOI: 10.14778/2350229.2350242.

[3] Alin Deutsch et al. "Graph Pattern Matching in GQL and SQL/PGQ". In: *SIGMOD*. ACM, 2022, pp. 2246–2258. DOI: 10.1145/3514221.3526057.

[4] Orri Erling et al. "The LDBC Social Network Benchmark: Interactive Workload". In: *SIGMOD*. 2015, pp. 619–630. DOI: 10.1145/2723372.2742786.

[5] Michael J. Freitag et al. "Adopting Worst-Case Optimal Joins in Relational Database Systems". In: *VLDB* 13.11 (2020), pp. 1891–1904. URL: http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf.

[6] Andrey Gubichev and Peter A. Boncz. "Parameter Curation for Benchmark Queries". In: *TPCTC*. Vol. 8904. Lecture Notes in Computer Science. Springer, 2014, pp. 113–129.

[7] Pankaj Gupta et al. "WTF: The who to follow service at Twitter". In: *WWW*. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 505–514. DOI: 10.1145/2488388.2488433.

[8] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. "Columnar Storage and List-based Processing for Graph Database Management Systems". In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2491–2504. DOI: 10.14778/3476249.3476297.

[9] Amine Mhedhbi and Semih Salihoglu. "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1692–1704. DOI: 10.14778/3342263.3342643.

[10] David Mizell, Kristyn J. Maschhoff, and Steven P. Reinhardt. "Extending SPARQL with graph functions". In: *BigData*. IEEE Computer Society, 2014, pp. 46–53. DOI: 10.1109/BigData.2014.7004371.

[11] Hung Q. Ngo, Christopher Ré, and Atri Rudra. "Skew strikes back: New developments in the theory of join algorithms". In: *SIGMOD Rec.* 42.4 (2013), pp. 5–16. DOI: 10.1145/2590989.2590991.

[12] Dan Olteanu and Maximilian Schleich. "Factorized Databases". In: *SIGMOD Rec.* 45.2 (2016), pp. 5–16. DOI: 10.1145/3003665.3003667.

[13] Mark Raasveldt and Hannes Mühleisen. "Don't Hold My Data Hostage - A Case For Client Protocol Redesign". In: *Proc. VLDB Endow.* 10.10 (2017), pp. 1022–1033. DOI: 10.14778/3115404.3115408.

[14] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An Embeddable Analytical Database". In: *SIGMOD*. ACM, 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212.

[15] Oskar van Rest et al. "PGQL: a property graph query language". In: *GRADES at SIGMOD*. 2016. DOI: 10.1145/2960414.2960421.

[16] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.

[17] Gábor Szárnyas et al. "The LDBC Social Network Benchmark: Business Intelligence Workload". In: *Proc. VLDB Endow.* 16.4 (2022), pp. 877–890. URL: https://ldbcouncil.org/docs/papers/ldbc-snb-bi-vldb-2022.pdf.

[18] Todd L. Veldhuizen. "Leapfrog Triejoin: a worst-case optimal join algorithm". In: *CoRR* abs/1210.0481 (2012). URL: http://arxiv.org/abs/1210.0481.