LDBC

Cooperative Project

FP7 – 317548

D2.2.4 Benchmarking Complex Queries

Coordinator: Andrey Gubichev

With contributions from: Peter Boncz (VUA), Orri Erling (OGL)

1st Quality Reviewer: Irini Fundulaki
2nd Quality Reviewer: Norbert Martinez

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M24
Actual delivery date:	M24
Version:	1.0
Total number of pages:	38
Keywords:	query optimization, parameter selection

Abstract

This document describes the design principles of the LDBC SNB Business Intelligence Workload. We base the query workload definition on a set of technical challenges (*choke points*) that the workload addresses. The second part of the document presents the technique to select parameters for benchmark queries that guarantees a stable and understandable runtime behavior of complex queries.

EXECUTIVE SUMMARY

The deliverable describes the Business Intelligence Workload of the LDBC Social Network Benchmark. The queries in the workload were designed with the following choke points (technical challenges) in mind:

- aggregation choke points
- cardinality estimation choke points
- subquery unnesting choke points
- navigational choke points
- query processing choke points

The document details specific technical difficulties within every group, and presents a set of 21 queries that cover the difficulties. It is expected that the query engine that solves all the presented challenges will excel at performing a wide spectrum of analytical queries (including the LDBC BI workload)

The second part of the deliverable solves a general problem of selecting parameters for benchmark queries such that the resulting runtime (and therefore the benchmark score) would be stable and predictable across different benchmark runs. The state-of-the-art benchmarks do not encounter this problem in its general form, since they are using synthetic data with very simple distributions (uniform or step-wise) and feature very little correlations between data attributes. In this situation it is enough to pick value bindings for parameters uniformly at random from the value domain. We show that for the realistically looking datasets this strategy does not work. We present our *Parameter Curation* solution, and demonstrate its efficiency and effectiveness on the highly correlated LDBC Social Network Benchmark (SNB) dataset.

Document Information

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

Deliverable	**Number**	D2.2.4	**Title**	Benchmarking Complex Queries
Work Package	**Number**	WP2	**Title**	Query Processing

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

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

Abstract (for dissemination)	This document describes the design principles of the LDBC SNB Business Intelligence Workload. We base the query workload definition on a set of technical challenges (*choke points*) that the workload addresses. The second part of the document presents the technique to select parameters for benchmark queries that guarantees a stable and understandable runtime behavior of complex queries.
Keywords	query optimization, parameter selection

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
15/09/2014	0.1	Andrey Gubichev	First version
26/09/2014	1.0	Andrey Gubichev	Final version

TABLE OF CONTENTS

1 Introduction

This deliverable addresses two questions. First, it presents a design rationale for the LDBC Social Network Business Intelligence benchmark. Namely, we describe the technical challenges (coined *choke points*) that a mature DBMS needs to solve in order to efficiently execute a workload of complex queries. These technical challenges overlap with the choke points of the classical TPC-H benchmark, but also contain a number of graph-specific difficulties, such as cardinality estimation in traversals and pattern matching queries.

The second issue that the deliverable aims to solve is selection of parameters for benchmark queries. Our scope here is broader than just Business Intelligence queries of LDBC SNB. In fact, this task occurs in various workloads, and it becomes non-trivial for real-world data or realistically looking synthetic datasets when the queries go beyond simple lookups (i.e., when there is more than one operator in the query plan). We demonstrate that parameter values picked uniformly at random from the parameter domain *does not* yield a stable, repeatable and easy-to-interpret runtime behavior of queries, and thus it is not a satisfactory strategy for parameter selection. We then formulate the problem of *Parameter Curation* as following: *find the set of parameter bindings that minimize the variance of the amount of intermediate results produced during a query plan execution*. Our solution to this problem is presented for several important classes of query parameters, e.g. single parameter in a query, multiple correlated discrete parameters, a discrete and a continuous parameter etc. The deliverable contains a description of experiments with *curated parameters* for LDBC SNB Workload that demonstrate advantages of Parameter Curation over uniform parameter selection.

Outline. The deliverable is structured as follows. In Chapter 2 we describe the design of Business Intelligence query workload; Section 2.2 gives a detailed discussion of related choke points, and Section 2.3 contains query descriptions with choke points coverage. In Chapter 3 we formulate and solve the *Parameter Curation* problem. Sections 3.1 and 3.2 motivate the need for a special parameter selection procedure with examples from LDBC SNB Interactive workload. In Section 3.4 we give a formal definition of a problem, and then describe algorithms to solve it in Section 3.4. Section 3.5 contains the description of experiments performed on the LDBC Interactive workload with curated parameters.

2 Design of the Business Intelligence Benchmark

This chapter describes a principle design of the BI workload for the LDBC Social Network Benchmark. We will follow an approach based on the *choke points* (specific technical challenges). Namely, we will first outline the set of choke points that a mature DBMS needs to address in order to efficiently run a graph analytical workload. Then, we describe a set of queries that cover these choke points. Some of our choke points are common with the ones described for the TPC-H benchmark (see [2]), while others are very graph-specific. We will formulate the queries for the generated data of the Social Network Benchmark, and use its properties (correlations, data skew); the data schema and properties are described in detail in the SNB Task Force Report[1]

2.1 Background

Business Intelligence (analytical) queries on social networks are common among users of graph/RDF technologies. As part of the Technical User Community meetings LDBC has hosted talks from a number of companies and universities that cover multiple aspects of social media analytics. Among others, the following use cases and talks influenced our BI queries design:

- Mediapro: Graph-based User Modelling through Social Streams [2]
- Dshini: Neo4J in Dshini [2]
- East China Normal University: Elastic and realistic social media generation[3]
- Shapespace Use Case [3]

BI queries has of course been the traditional subject of benchmarking efforts. The list of most prominent analytical benchmarks include

- TPC-H: a classical and influential analytical benchmark. Some of our choke points were directly taken from the TPC-H analysis, see [2]
- TPC-DS: an analytical relational benchmark that is meant to replace TPC-H. Currently because of its size (99 queries) there are no officially scored implementations.
- BSBM-BI, the analytical RDF benchmark which is still rather limited in scope: although it addresses basic SPARQL 1.1 features in its queries (aggregation and subqueries), it is still primitive in comparison with TPC-H and DS

2.2 Choke Points of Complex Query Processing

The technical difficulties that arise in complex query processing can be divided in three groups:

- choke points that test the **algebraic power** of the optimizer, i.e. the ability to consider the entire (or sufficiently large) search space of various algebraically equivalent reformulations of the query and pick the cheapest one. Examples of such choke points include subquery unnesting and non-inner join reordering.
- triggered by the problem of **cardinality estimation**. Cardinality of intermediate results influences various choices of the optimizer, most notably join order selection and physical operator selection (index vs hash). Presence of data skew and correlations in the LDBC SNB dataset makes this fundamental problem extremely challenging.

[1]see http://www.ldbc.eu:8090/download/attachments/4325436/LDBC_SNB_Report_Nov2013.pdf
[2]http://ldbc.eu:8090/display/TUC/Second+TUC+meeting%2C+April+2013
[3]http://ldbc.eu:8090/display/TUC/Third+TUC+Meeting%2C+November+2013

- choke points that address the **evaluation power** of the query processing system, such as arithmetical computations, evaluation of conditional expressions. These choke points will force the DBMS (especially the main-memory systems) to consider vectorization or query compilation in order to reduce overhead of expression interpretation.

In the rest of the section we describe concrete choke points. Some of them are identical to the ones of TPC-H workload; this is very natural, since we are designing an analytical workload. Other choke points, however, are unique to graph workloads, they appear in Subsections 2.2.3 and 2.2.5.

2.2.1 Choke Points on Grouping

Grouping and aggregation operations are ubiquitous in analytical workloads. This group of choke points stresses the ability of optimizer to speed up evaluation of group-by's in commonly occurring situations such as dependencies in the grouping keys or small domain size of the keys. Choke points of this group belong to the first two classes of choke points in our classification (*algebraic power* and *cardinality estimation*).

CP-1.1: Dependencies in Group-By Keys Systems usually use hash-tables to store group-by keys. Multiple grouping keys therefore result in a significant CPU and memory overhead (multiple comparisons and storing large amounts of attributes). Frequently, however, there are (functional) dependencies among keys (such as when grouping on *person_id*, *person_name* and *person_email*, where the latter two attributes functionally depend on the first – primary key – attribute). In such cases, it is enough to group on the primary key only. The optimizer should be able to detect such situations based on a declared schema (relational systems, property graph databases) or detected foreign keys (using, for example, the *characteristic sets* approach in RDF systems).

CP-1.2: Ordered Aggregation When all the equal group-by keys appear consecutively in the input tuple stream consumed by the aggregation operator, the system should employ *ordered aggregation* instead of hash-based one. This happens in several situations, such as when keys come from clustered indexes or from underlying order-preserving joins. An example in LDBC SNB dataset would be a query grouping by *post_id* and *liker*: once the aggregation operator has seen the post with a certain ID, it will never encounter it again.

CP-1.3: Group-By key domain size: small vs large If the grouping key domain is small (such as gender or first name), a specialized version of the grouping operator should be applied: it can use an array rather than a hash table, to keep the aggregate function values. Extremely large domain size (together with large input) may lead to *spilling* aggregation. Detection of both cases naturally requires accurate cardinality estimation.

2.2.2 Subquery Choke Points

Efficiently handling subqueries implies the ability to rewrite the query in an equivalent way (most frequently, by unnesting the subquery). This group of choke points emphasizes *algebraic power* of the optimizer.

CP-2.1: Subquery Rewriting Flattening the subqueries (i.e., rewriting them using an equi-, outer-, or anti-join) is the key skill of a query optimizer for executing complex queries efficiently [6]. There is usually a variety of ways of subquery decorrelation. In order to choose between different options, the optimizer needs to correctly estimate the amount of intermediate results, which is a challenge in itself for SNB dataset that features multiple correlations and data skew.

CP-2.2: Joining Derived Tables with aggregates . This is an important case of a more general *Reusing Computation* which will be described later (see CP-6.1). When two derived tables contain aggregate on the same attribute, and there is a join in the outer query between derived tables, the hash tables constructed for aggregation should be reused for hash join execution. Another option is to push the join between two derived tables below the second group-by, although it is not always beneficial; detecting which situation applies is part of the challenge.

CP-2.3: Moving Predicates Between Subqueries This choke point occurs when the subquery computes an aggregate (*average number of friends of users in China*) which is used in a selection in outer query. In such situations some additional constraints from the outer query can be transferred to subquery (if the outer query, for example, restricts the age group of users). Identical choke point appears in multiple TPC-H queries (TPCH2, TPCH17, TPCH20).

2.2.3 Cardinality Estimation Choke Points

Cardinality estimation of intermediate query results is a fundamental problem of query optimization. Despite years of academical and industrial attention, cardinality estimation in real-world datasets (i.e., datasets that have non-uniform data distribution and correlations) is an extremely tough problem (and overall, an open research question).

CP-3.1: Choosing Between Index and Hash *Index vs Hash* is a fundamental choice between physical join operator implementations that the optimizer needs to make based on cardinality estimates: if the left input of the join produces a very few tuples and there is an index on the right side, then the join can be efficiently performed by looking up values from the left into the index on the right. Otherwise, the hash join is probably the most efficient implementation.

CP-3.1a: Estimations in Trees This choke point applies in (sub)queries that contain hierarchy traversals, e.g. *find all posts and reply trees that are tagged with "Franz Kafka"*. Correct cardinality estimates are later used by the optimizer to re-order traversal with other parts of the query. Additionally, estimates influence the physical operator selection, such as: (i) from which side of the traversal should one start expansion? (ii) if a hash join used to split the traversal evaluation, which end serves as a build side? Moreover, recognizing that a traversal is performed on a tree structure (and not on a general graph) is itself a challenging problem.

CP-3.1b: Estimations in Graphs A generalization of the previous choke point. Requires (approximate) counting of paths (and more complex structures) with restrictions on edges and nodes in a graph, which is a very challenging problem in a general-shape graphs. In addition, the graph itself can be implicit (e.g., *people that frequently talk to each other*).

CP-3.1c: Estimations in Full Text Search Cardinality estimation for full text search is only feasible when there is a specialized index for text search. Again, the estimation influences the physical operator choice. Consider, for example, (part of) the query: *find popular posts that do not contain certain words (bag of words)*. Depending on how selective the condition on the text is and how many popular posts are there, the optimizer has basically two options: (i) lookup the given pattern in the text index, and then continue execution by joining the results with the popular posts, (ii) find the popular posts and filter out those that do not contain given words, ignoring the text index.

CP-3.2: Estimations on Foreign Key joins Most of the time joins involving two foreign keys are assumed to be non-selective and expensive. An important example of the opposite situation is *triangle matching* in graphs: matching a triangle pattern of a form $A \to B \to C \to A$ involves two joins, first one on B and second one on C, which is a very selective Foreign Key – Foreign Key join. Recognizing this is a very tricky challenge for the optimizer. After that, specific tricks for sparse joins (Bloom filters) should be applied. Despite the "foreign key" term, this choke point is not an RDBMS-specific: for graph and RDF systems it boils down to detecting that the size of an intersection (join) of two neighborhoods $A \to B \to C$ and $C \to A$ is very small.

2.2.4 Choke Points of Query Processing

CP-4.1: Arithmetic and Conditional Expressions Choke points that arise during evaluation of various expressions are described in [2]. In short, interpretation overhead of a tuple-at-a-time evaluation leads system to adoption of vectorization, Just-in-Time compilation, or GPU- and FPGA-based approaches.

CP-4.2: Full Text Search String manipulations are much more expensive then arithmetic computations. A frequent corner case of prefix search can be re-written into a less expensive string range comparison. General full text search is best done with a specialized index.

CP-4.3: Parallel Execution In the era of many-core architectures, the query parallelization is an extremely important topic, both for query optimization and query execution. Further, some systems may focus on scaling out; this usually involves partitioning over many nodes in a cluster.

2.2.5 Navigational Choke Points

CP-5.1: Materializing Results During Traversals While computing aggregate scores over a densely connected graph, same edges are visited multiple times. A typical example is a PageRank-style discovery of most liked (i.e.,most popular) person, where likes from other popular users weigh more. Here, the weighted graph structure is implicit and a beneficial approach would be to materialize it into a hash table, thus improving access locality and saving CPU time.

CP-5.2: Trees vs Graphs Telling tree-like structures (or DAGs) from graphs of general shape is important for the following reasons: (i) cardinality estimations may be more accurate for trees, (ii) this, in turn, helps to determine the start and end points of traversals, as well as physical operator details (hash vs index, how many traversal steps to put on build side of the hash table).

CP-5.3: Transitivity on Small Dimensions When the query involves traversing small hierarchies (represented as *dimension tables* in the relational domain), the optimal strategy is to put the result of the entire traversal on a build side of a hash join. For example, when looking for *average number of replies of posts written in Germany*, the traversal on geographical hierarchy (*written in Germany*) yields a (relatively) small number of locations; all of them should be put into a hash table, and then posts are probed on this hash table. In addition to detecting a tree-like structure, the optimizer should correctly estimate the size of the tree.

2.2.6 Miscellaneous Choke Points

CP-6.1: Touching Multiple Attributes Following this principle, we will include multiple queries that cover the same choke points, if they address different attributes of tables

CP-6.2: Non-Inner Join Ordering The join ordering problem is further complicated with operators that are not freely reorderable, like non-inner joins or joins involving more than two tables. Optimizers that are able to systematically (and efficiently) explore the entire search space for various types of operators will find the optimal query plan for complex queries.

CP-6.3: Union of Views Similar to subqueries, views present multiple possibilities for query rewriting. Given the join between several views (each view may contain unions), the optimizer should choose between options like join of unions or union of joins.

CP-6.4: Top-k Pruning Top-k clause usually accompanies the group-by of high cardinality, where the number of groups is too high to display. If ordering that is used for top-k is on the grouping column, the query processor should generate exactly k groups, and then use values of the kth group to prune some of the further computation (i.e., to prune values that certainly will not make it into top-k). Similarly, if there is an ordering by some other metric (e.g. date), the Top-k can be pushed down to the selection as well.

2.3 Query Set

In this section we give a set of 21 queries that cover choke points from Section 2.2. For every query we will list the following description items: (i) a business question that the query solves, (ii) choke point(s) that are relevant to the query, (iii) input parameters, (iv) a short commentary that outlines how a well-designed optimizer should compile the query, or what choices an optimizer has while choosing a good execution plan. Full SQL formulations of all queries are given in Appendix A and on Github (http://github.com/ldbc). Note that the given query set is preliminary, i.e. some query definitions may be modified later, but the choke points coverage is meant to stay the same. In particular, the precise definition of every query in English (as opposed to current definitions in SQL only) is subject of future work. Our goal here is to describe the general choke-based design of the query workload.

Query 1: top 100 popular topics in each country, age and gender group

Query: across all the posts published in the given period of time, find top 100 most popular tags (i.e. tags with largest number of posts), grouped by age group, gender and the country of the post creator
Business Question: what are the popular topics depending on demographics (age, gender, country) of authors?
Parameter(s): Start date, duration. Usually user activity increases with time.
Choke Point(s): CP-1.3

Query 2: find new tags that appeared during last month

Query: for a given month, find the tags that were used in posts during that period, and the tags that were used in the month before the given month. In both cases compute the count of posts that use the tags. Report the tags that maximize the difference in post counts between the given month and the preceding month.
Business Question: what are the emerging popular topics?
Parameter(s): Time interval (the month in question)
Choke Point(s): CP-2.2
Comments: Popularity of tags is expressed as two derived tables with aggregation, which meant to be independently executed and result in two hash tables on the same grouping key. These two tables are then joined, thus effectively re-using the hash structures from aggregation.

Query 3: find most relevant forums on a given topic in a given country

Query: for a given class of topics (e.g., *Musical instruments*) and a given country, find the most popular forums on this topic. The way we identify the topic of the forum is by tags of posts in this forum; the location of the forum is identified by the location of the forum's moderator (in SNB, forums are similar to Facebook's walls). The result is grouped by forum's id, title, creation date and moderator, and ordered by the count of messages with the given topic class.
Business Question: where should the product be advertised?
Parameter(s): Tag Class and Country
Choke Point(s): CP-1.1, CP-4.3
Comments: classical example of the group by attributes with functional dependencies. Additionally, this is a large cardinality group by with possibilities for parallelization.

Query 4: top posters in 100 top forums in China

Query: Among the participants of the 100 most popular forums (i.e., forums with the largest number of posts) in a given country, find the top 100 posters (users that created the largest amount of posts in one of these forums). Order the result by number of created posts per returned user.
Business Question: who are the famous or influential users? (as in: users that post a lot in very popular forums in a given country)
Parameter(s): Country
Choke Point(s): CP-2.1
Comments: Flattening of subqueries. Additionally, the query stresses the implementation of Top-K.

Query 5: top posters on a given topic

Query: get top 100 users who post on a given subject (tag). Users are sorted by a score that includes count of likes and replies to their posts with a given tags, aggregated with different weights. Return users, counts of the posts, replies and likes and their aggregated score, ordered by the score.
Business Question: who is the expert on a given topic? Authority of the expert here is defined as amount of likes and replies his posts receive.
Parameter(s): Tag name (topic)
Choke Point(s): CP-2.1, CP-3.1, CP-4.1
Comments: Multiple transformation of correlated subqueries, depending on the selectivity of the tag: by index (selective tag), by hash (hash table with likes and replies for an unselective tag; we could migrate conditions on the build side), alternatively group-join.

Query 6: most authoritative user posting on a given topic

Query: Find 100 most authoritative users posting on a given topic. The authority is a number of likes received to the users's post on a given topic, such that likes from much liked posters weigh more.
Business Question: who is the expert on a given topic? PageRank-style computation of authority of users (unlike Query 5)
Parameter(s): Tag name (topic)
Choke Point(s): CP-2.1, CP-3.1, CP-5.1
Comments: A trick here is to avoid computing likes of the same people multiple times. Some materialization of like count table is required.

Query 7: tags that are most frequently mentioned together with the given tag

Query: Find top 100 tag names that get mentioned in replies to posts with the given tag. Order by decreasing number of such co-occurences.
Business Question: find top brand competitors: something that gets mentioned together with a tag, but is not that tag.
Parameter(s): Tag name
Choke Point(s): CP-2.1
Comments: Rewriting EXIST and NOT EXIST subqueries. If the EXIST condition is selective, one can transform it into derived table with distinct, and perform it first in the join order. Same thing can not be done with NOT EXIST, though.

Query 8: anti-correlation between two tags' occurrence

Query: Find the forums where that contain posts with either of two given tags t_1 and t_2. Order the forums by the difference of count of posts with tags t_1 and t_2, respectively.
Business Question: where should we advertise the product? Where are competitor brands discussed but not ours?
Parameter(s): Two tag names; will be picked to be anticorrelated, i.e. there are forums that have posts with one tag, but not with another
Choke Point(s): CP-2.1, CP-3 (general cardinality estimation)
Comments: Unnesting subqueries in presence of several predicates on tags (which implies the need for accurate cardinality estimation)

Query 9: find the most central user for the given tag

Query: find the top 100 users that talk about a given tag, such that their friends also talk about the same tag. Order users by score, which is in this case the amount of posts by the user and her friends on a given topic.
Business Question: who is the expert on a given topic? Yet another way to define authority of a user on a given subject: most of the friends talk about the same tag.
Parameter(s): Tag name (topic)
Choke Point(s): CP-2.1, CP-3.1
Comments: depending on the tag cardinality, the solution should start with either people or tags

Query 10: find people that successfully introduce a new topic

Query: finds users that talk about a new tag, their posts get liked and replied to, and replies do not contain "garbage" (given as a bag of words). Order the users by the number of times they introduced a new tag.
Business Question: who are the thought leaders?
Parameter(s): Country (origin of the user)
Choke Point(s): CP-3.1c, CP-4.2
Comments: Among other issues, this is a full-text search query.

Query 11: most liked content of the network in the given period

Query: return 100 most recent posts (made after a certain date) that were liked more than a certain number of times. Group by the post id, creator, date.
Business Question: find the current popular content of the social network
Parameter(s): Time interval
Choke Point(s): CP-1.2
Comments: Group by key is a post id, so it is an ordered aggregation: tuples come in as a stream, and once we have seen a certain post, we will never see it again.

Query 12: top thread initiators

Query: Return top 100 users that started discussion threads (i.e., posts followed by reply trees). All replies and posts should be created in a certain time interval.
Business Question: find the thread initiators
Parameter(s): Time interval
Choke Point(s): CP-3.1a, CP-5.2
Comments: Tree traversals and cardinality estimations in trees with conditions on nodes.

Query 13: top 100 people with the number of friends higher than average in China

Query: For a given country, find the users that have more than an average number of friends in that country.
Business Question: find better-than-average-connected users in a given country
Parameter(s): Country
Choke Point(s): CP-2.3
Comments: Correlated subquery with average, similar to TPCH17. We can transfer conditions between outer query and subquery

Query 14: find people that are connected to each other and talk about one topic

Query: for a given country and topic, find all the people from that country that know each other (transitive closure of the *knows* relationship) and talk about this topic
Business Question: find experts in some domain linked via contact chain, all working for Chinese companies
Parameter(s): Country, Topic
Choke Point(s): CP-3.1b
Comments: Transitive query in a general shape graph: both cardinality estimation and execution are hard problems

Query 15: triangle counting

Query: for a given country, count all the triangles formed by friendship relationship between users in that country.
Business Question: count small communities of people in one country
Parameter(s): Country
Choke Point(s): CP-3.2
Comments: End points of edges in the graph are identifiers; for RDF implementation comparing URIs is very hard, implementation-specific syntax should be allowed. As discussed in section 2.1, the Foreign Key – Foreign Key join here is actually very selective (see CP-3.2 description)

Query 16: distribution of number of posts

Query: construct the distribution of number of posts created after a given timestamp, i.e. compute how many people created certain number of posts. Users with 0 posts are also considered (with left outer join)
Parameter(s): Start time
Choke Point(s): CP-6.3
Comments: optimizer should be able to re-write the query with the right outer join (similar to TPCH 13)

Query 17: communication between strangers

Query: Find all the users that were born after a certain date (e.g., teenagers) that are frequently replying to posts from strangers. These strangers should be members of two forums with specified tags. Order the results by the amount of replies exchanged.
Business Question: detect unusual communication patterns in the network
Parameter(s): Start time
Choke Point(s): CP-4.1
Comments: evaluation of conditional expression (not exists)

Query 18: frequency of topics

Query: Find the frequency of the topics that are one level below the most general topic (root of the hierarchy)
Business Question: frequency of generic topics across all posts
Parameter(s): –
Choke Point(s): CP-5.2
Comments: query optimizer needs to figure if it is a graph or a tree? (in this case a tree)

Query 19: Zombies: People who are liked by people who produce nothing

Query: "Person who produces nothing"; somebody who has been on-line for more than 1 month, has less than 5 posts. "Zombie score" is calculated as number of likes of the content of the author divided by the likes which come from people that produce nothing. The query finds all the zombies from a given country with the score bigger than a threshold.
Business Question: find the "zombies": fake profiles that like each other's activities.
Parameter(s): Country and time period
Choke Point(s): CP-5.1
Comments: likes have high cardinality. When one looks at posts of the person, the same likers are visited multiple times. Good opportunity for materialization

Query 20: correspondence between two countries

Query: We define relationship *related* as a union of: (i) two users reply to each others posts, (ii) two users know each other, (iii) two users like each others posts; these three cases have different weights. Given two countries (e.g., USA and Yemen), return people that are related to each other, ordered by relatedness weight.
Business Question: "mole hunt"
Parameter(s): Two countries. These will be anti-correlated
Choke Point(s): CP-6.4
Comments: multiple possibilities to re-write the union from the *related* definition.

Query 21: tag timeline

Query: Return a histogram of post count for posts with a tag belonging to a general tagclass (e.g., *Politician*). Group by month, year, continent (the latter can be NULL) of the post.
Parameter(s): Tag name
Choke Point(s): CP-5.3
Comments: Star transformation: the entire transitive dimension (hierarchy of tags) goes onto the build side of hash.

2.4 Roadmap

The following open issues in workload design will be settled in the future:

- **Updates**: In case of TPC-H, the updates are just appends to two tables. In TPC-DS the updates are more involved since the data is delivered in raw format with respect to values, and then first the dimension tables need to be queried; this is more realistic for a data warehousing workload.
- **Power vs Throughput experiments** are the two possible options for running the workload. In Power experiments, the queries are issued one after another, and the system has to explore intra-query parallelism. In Throughput experiments, the system gets many queries in parallel.
- **Metrics**: Usually, the benchmark score is a function of power and throughput experiments. This function can take either geometric mean or sum of the times of all queries. First option encourages system's architects to improve all queries, while the second one in fact stimulates improvement in the worst (the longest) query.

In order to issue many different queries (i.e., queries with the same template but different parameter bindings), and have comparable runtime for all these queries, we need to mine many parameter bindings for each template from the corresponding value domains. The second part of the deliverable describes our work on this question.

3 Parameter Curation

3.1 Motivation

A typical benchmark consists of two parts: (i) the dataset, which can be either real-world or synthetic, and (ii) the workload generator that issues queries against the dataset based on the pre-defined *query templates*. A query template is an expression in the query language (e.g., SQL or SPARQL) with *substitution parameters* that have to be replaced with real bindings by the workload generator. For example, a template of a query that asks for all the movie producing companies from the country *%Country%* that have released more that 20 movies, looks like:

Query 3.1: IMDB Query

```
SELECT cn.name, COUNT(t.id) cnt
FROM title t, movie_companies mc, company_name cn
WHERE t.id = mc.movie_id AND cn.id = mc.company_id
        AND cn.country_code = '%Country%' AND t.kind_id = 1
GROUP BY mc.company_id, cn.name
HAVING COUNT(*) > 20
ORDER BY cnt DESC
LIMIT 20
```

In a query workload, the workload driver would execute this query template in one experiment potentially multiple times (e.g., 10) with different bindings for the `%Country` parameter. It would report an aggregate value of the observed runtime distribution per query (usually, the average runtime per query template). This aggregated score serves two audiences: First, the users can evaluate how fit a specific system is for their use-case (choosing, for example, between systems that are good in complex analytical processing and those that have the highest throughput for lookup queries). Second, database architects can use the score to analyze their systems' handling of certain technical challenges (" choke points" [2]), like handling multiple interesting orders or sparse foreign key joins.

In "throughput" experiments, the benchmark driver may also execute the above experiment multiple times in multiple concurrent query streams. For each stream, a different set of parameters is needed.

Desired Properties. In order for the aggregate runtime to be a useful measurement of the system's performance, the selection of parameters for a query template should guarantee the following properties of the resulting queries:

P1: the query runtime has a bounded variance: the average runtime should correspond to the behavior of the majority of the queries

P2: the runtime distribution is stable: different samples of (e.g., 10) parameter bindings used in different query streams should result in an identical runtime distribution across streams

P3: the optimal logical plan (optimal operator order) of the queries is the same: this ensures that a specific query template tests the system's behavior under the well-chosen technical difficulty (e.g., handling voluminous joins or proper cardinality estimation for subqueries etc.)

The conventional way to get the parameter bindings for `%Country` is to sample the values (uniformly, at random) from all the possible country names in the dataset (the "domain"). This is, for example, how the TPC-H benchmark creates its workload. Since the TPC-H data is generated with simple uniform distribution of values, the uniform sample of parameters trivially guarantees the properties **P1-P3**. The TPC-DS benchmark moved away from uniform distributions and uses "step-shaped" frequency distributions instead [5, 7], where there are large differences in frequency between steps, but each step in the frequency distribution contains multiple values all having the same frequency. This allows TPC-DS to obtain parameter values with exactly the same frequency, by choosing them all from the same step.

However, these techniques do not work for benchmarks that use real-world datasets (IMDB in our example, or DBPedia etc.), or generate datasets with skewed value distribution and close-to-realistic correlations between values (LDBC Social Network Benchmark, which is based on S3G2 generator [3]). In our example above, the behavior of the query changes significantly depending on the selection of the parameter. We present a detailed analysis of its behavior in Section 3.2, but most notably, if `%Country` is '[US]', the query features a voluminous join between `movie_companies` and `movie`, while for smaller countries (like '[FI]') the join is very sparse. As we see, two very different scenarios are tested for these two parameter choices, and they should ideally be reported separately. The country parameter bindings for these two scenarios would be drawn from two buckets of countries, with large number of movies ('[US]', '[UK]', '[FR]' etc) and with a few movies ('[HK]','[DK]' etc). The recently proposed LDBC Social Network benchmark is another example where one would need to carefully select parameters in order to avoid large variability of plans and execution times.

We clarify that our intention is not to obviate the interesting query optimization problems related to the real-world distributions and correlations in the dataset, but to make the results within one query template predictable by choosing the parameters that satisfy properties **P1**-**P3**, in order to guarantee that the behavior of the System Under Test (SUT) and of the benchmark results is *understandable*. In case different parameters have very different runtimes and optimal query plans (e.g. due to skew or correlations) this can still be tested in a benchmark by having multiple query *variants*, e.g., one variant with countries where many movies are made, another with countries where rarely movies are made. The different variants would behave very differently and test whether the optimizer makes good decisions, but within the same query variant the behavior should be stable and understandable regardless the substitution parameter.

Parameter Curation. In this chapter we present an approach to generate parameters that yield similar behavior of the query template, which we coin "Parameter Curation". We consider a setup with a fixed set of query templates and a dataset (either real-world or synthetic) as input for the parameter generator. Our approach consists of two parts:

- for each query template for all possible parameter bindings, we determine the size of intermediate results in the *intended* query plan. Intermediate result size heavily influences the runtime of a query, so two queries with the same operator tree and similar intermediate result sizes at every level of this operator tree are expected to have similar runtimes. This analysis on result sizes versus parameter values is done once for every query template (remember that we consider benchmarks with a *fixed* set of queries).
- we define a greedy algorithm that selects ("curates") those parameters with similar intermediate result counts from the dataset.

Note that Parameter Curation depends on data generation in a benchmark: we are mining the generated data for suitable parameters to use in the workload. As such, Parameter Curation constitutes an new phase that follows data generation in a typical database benchmarking process.

The astute reader may remark that `%Country` in the previous example has the limitation that the country domain is rather limited. Thus, a need to select e.g., 100 parameter values would imply using a large part of the domain, and in case of skewed frequency distribution would lead to unavoidable large variance. This does not invalidate our approach to select parameters in an as stable manner as possible, and we note that benchmark queries tend to have (or can be made to have) multiple parameters, so the amount of parameter combinations is the product of the parameter domain sizes, thus grows explosively, so limited parameter choices should not be an issue in general.

Our Parameter Curation algorithms are implemented as part of the LDBC Social Network data generator [1]. The generator outputs sets of curated parameter bindings for every query of SNB Interactive workload, and the the workload generator uses these bindings to issue queries against the system under test. Extending the generator to produce parameters for Business Intelligence workload is subject of future work.

Outline. The rest of the chapter is organized as follows. In Section 3.2 we demonstrate in examples that the straightforward approach of generating parameter bindings uniformly at random fails to deliver predictable and

[1] See http://github.com/ldbc and http://ldbcouncil.org

stable results. Section 3.3 formalizes the problem of *curating parameters* that would yield runtime distribution satisfying properties **P1** - **P3**. In Section 3.4 we present our implementation of Parameter, used in the LDBC Social Network Benchmark (SNB). Section 3.5 describes the set of experiments we conducted on SNB Interactive queries.

3.2 Examples

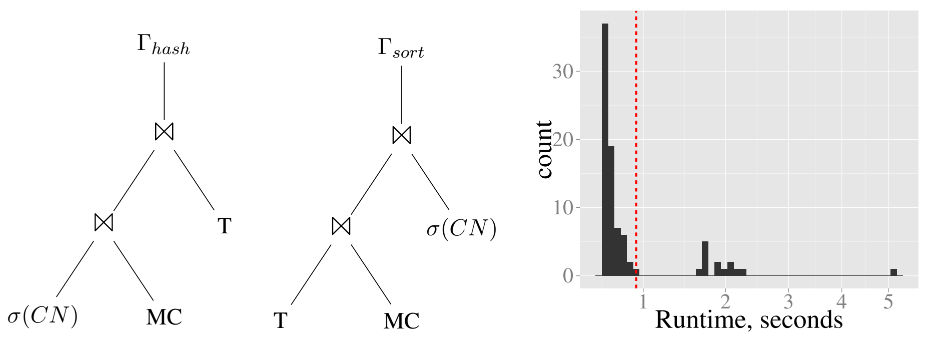
We use the LDBC Social Network Benchmark [1] (Interactive Workload) and a query on IMDB dataset from Query 3.1. For the LDBC SNB Interactive, we generated a social network with 50.000 users (ca. 5 GB of CSV files). For both datasets we use Virtuoso 7 database (Column store) and run our experiments on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 64 Gb RAM, 1 TB SAS-HD, Redhat Enterprise Linux (2.5.37).

In the following examples (**E1-E4**), we illustrate our statement that uniform selection of parameters leads to unpredictable behavior of queries, which makes interpretation of benchmark results difficult.

E1: Runtime distribution has high variance. When drawing parameters uniformly at random, we encounter a very skewed runtime distribution for queries over real-world datasets. The runtime of the query from Query 3.1, for example, has a variance of $17 \cdot 10^4$. This is caused by the fact that the majority of the movies is produced in a single country, US; additionally, the top 10 countries produce 3 times more movies than all the other countries together. This translates into highly variable amount of data that the query needs to touch depending on the parameter, which in turn influences the runtime.

This issue is also important for the LDBC benchmark, where the data generator seeks to mimic some of the properties of the real-world data: the generated data has correlations and skewed data distributions. In this case, naturally, the randomly generated parameter bindings result in a very skewed runtime distribution.

E2: Different plans for different parameters. The uniformly generated parameter bindings can lead to completely different plans for the same query template. This happens because the cardinalities of the subqueries naturally depend on the parameter bindings, and sometimes on the combination of the parameters. For example, two optimal plans for Query 3.1 (as found by the PostgreSQL database) are depicted in Figure 3.1a) and b), where leaves are marked with table aliases from the query listing. Picking 'US' as a parameter not only changes the join order, as compared with the 'UK' parameter, but also results in applying a different group-by method (by sorting as opposed to hash-based grouping for the 'UK' parameter).



(a) %Country = 'UK' (b) %Country = 'US' (c) Runtime distribution, red line marks the mean

Figure 3.1: IMDB Query 3.1 plans and runtime distribution for different parameters

As another example, we consider LDBC SNB Interactive Query 3 that *finds the friends and friends of friends that have been to countries X and Y*. The optimal plan for this query can start either with finding all the friends within two steps from the given person, or from extracting all the people that have been to countries X and Y: if X and Y are Finland and Zimbabwe, there are supposedly very few people that have been to both, but if X and Y are USA and Canada, this intersection is very large. In the LDBC benchmark, correlations that might not even be detected by the optimizer aggravate the execution picture beyond plain frequency differences. There is a correlation between the location of each user and her friends (they often live in the same country) and travel destinations are correlated so that nearby travel is more frequent. Hence combinations of countries far from home are extremely rare and combinations of neighboring countries frequent.

We note that the plan variability is not a bad property *per se*: indeed, this query forces the query optimizer to accurately estimate the cardinalities of subqueries depending on input parameters. However, the generated parameters should be sampled independently for two different variants (countries that are rarely and frequently visited together), to allow a fair and complete comparison of different query optimization strategies.

E3: Average runtime is not representative. In addition to being far from uniform (**E1**), the query runtime distribution can also be "clustered": depending on the parameter binding, the query runs either extremely fast or surprisingly slow, and the average across the runtimes does not correspond to any actual query performance. To illustrate this issue, we consider again the IMDB Query 3.1. Figure 3.1c shows the runtime distribution of that query over the entire domain of %Country parameter bindings. We see that the average runtime (red line on the plot) falls outside of the larger group of parameter bindings, so in fact very few actual queries have the runtime close to the mean.

E4: Sampling is not stable. A single query in the benchmark is typically being executed several times with different randomly chosen parameter bindings. It is therefore interesting to see how the reported average time changes when we draw a different sample of parameters. In order to study this, we take Query 2 of the LDBC SNB that *finds the newest 20 posts of the given user's friends*. We sample 4 independent groups of parameter bindings (100 user parameter bindings in each group), run the query with these parameters and report the aggregated runtime numbers within individual groups (q_{10} and q_{90} are the 10th and the 90th percentiles, respectively).

Time	Group 1	Group 2	Group 3	Group 4
q_{10}	0.14 s	0.07 s	0.08 s	0.09 s
Median	1.33 s	0.75 s	0.78 s	1.04 s
q_{90}	4.18 s	3.41 s	3.63 s	3.07 s
Average	1.80 s	1.33 s	1.53 s	1.30 s

We see that uniform at random generation of query parameters in fact produces unstable results: if we were to run 4 workloads of the same query with 100 different parameters in each workload, the deviation in reported average runtime would be up to 40%, with even stronger deviation on the level of percentiles and median runtime (up to 100%). When TPC benchmark record results are improved, this often only concerns minor difference with the previous best (e.g. 5%). Hence, the desired stability between different parameter runs of a benchmark should ideally have a variance below that ballpark.

3.3 Problem Definition

Here we define the problem of generating the parameter binding for benchmark queries. In order to compare two query plans formulated in logical relational algebra, we use the classical logical cost function that takes into account the sum of intermediate results produced during the plan's execution [4]:

$$C_{\text{out}}(T) = \begin{cases} |R_x| & \text{if } T \text{ is a scan of relation } R_x \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

The above formula is incomplete and just here for argumentation; a more complete version of this logical cost formula naturally should include all relational operators (hence also selection, grouping, sorting, etc). The main idea is that for every relational operator T_y it holds the amount of tuples that pass through it.

In our experiments, the cost function C_{out}, which is computed using the de-facto result sizes (not the estimates!), strongly correlates with query running time (ca. 85% Pearson correlation coefficient). Therefore, if two query plan instances have the same C_{out}, or even better if all operators in the query plan have the same C_{out}, these plans are expected to have very similar running time.

In order to find k parameter bindings that yield identical runtime behavior of the queries, we could:

a: enumerate the set of all equivalent logical query plans L_Q for a query template Q.

b: for each possible parameter p from domain P, and each subplan T_{lq} of L_Q compute $C_{out}(T_{lq}(p))$.

c: find subset $S \subset P$, with size $|S| = k$, such that the sum of all variances $\sum_{\forall T_{lq} \in L_Q} \mathit{Variance}_{\forall p \in S}\ C_{out}(T_{lq}(p))$ is minimized.

Note that this generic problem of parameter curation is infeasibly hard to solve. The amount of possible query plans is exponential in the amount of operators (e.g. $2^{|L_Q|}$, just for leftdeep-only plans, and $|L_Q|$ being the amount of operators in plan L_Q), and all these plan costs would have to be calculated very many times: for each possible set of parameter bindings (whose size is $2^{|P|}$, where $|P|$ is the product of all parameter domain sizes – a typically quite large number), and for all $|L_Q|$ subplans of L_Q.

Instead, we simplify the problem by focusing on a single *intended* logical query plan. Since we are designing a benchmark, which consists of a relatively small set of query templates (the intended benchmark workload), and in this benchmark design we have certain intentions, this is feasible to do manually. We can, therefore, formulate a more practical problem of Parameter Curation as follows:

PARAMETER CURATION: For the Intended Query Plan QI and the parameter domain P, select a subset $S \subset P$ of size k such that $\sum_{\forall T_{qi} \in QI} \mathit{Variance}_{\forall p \in S}\ C_{out}(T_{qi}(p))$ is minimized.

Since the cost function correlates with runtime, queries with identical optimal plans w.r.t. C_{out} and similar values of the cost function are likely to have close-to-normal distribution of runtimes with small variance. Therefore, the properties **P1-P3** from Section 3.1 hold within the set of parameters S and effects mentioned in Section 3.2 are eliminated.

The Parameter Curation problem is still not trivial. A possible approach would be to use query cardinality estimates that an EXPLAIN feature provides. For each query template Q we could fix the operator order to the intended order QI, run the query optimizer for every parameter p and find out the estimated $C_{out}(QI(p))$, and then group together parameters with similar values. However, it seems unsatisfactory for this problem, since even the state-of-the-art query optimizers are often very wrong in their cardinality estimates. As opposed to estimates we will therefore use the de-facto amounts of intermediate result cardinalities (which are otherwise only known after the query is executed).

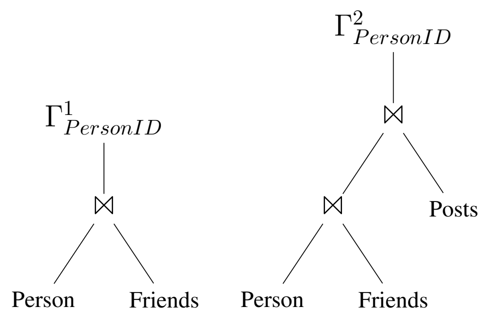
3.4 Implementation of Parameter Curation

In this section we demonstrate how the problem of Parameter Curation for a given query plan is solved in several important cases, namely:

- a query with a single parameter
- a query with two (potentially correlated) parameters, one from discrete and another from continuous domain. Such a combination of parameters could be: *Person* and *Timestamp* (of her posts, orders, etc).
- multiple (potentially correlated) parameters, such as *Person*, her *Name* and the *Country* of residence.

Note that our solution easily generalizes to the cases of multiple parameters (such as two *Timestamp* parameters etc); we consider the simplest cases merely for the purposes of presentation.

Our solution is divided into two stages. First, we perform *data analysis* that aims at computing the amount of intermediate results produced by the given query execution plan across the entire domain of parameter(s).



(a) Step 1:
# Friends per Person

(b) Step 2:
# Posts of Friends

PersonID	$\|\Gamma^1\|$	$\|\Gamma^2\|$
...	...	...
1542	60	99
1673	60	102
7511	60	103
958	61	120
1367	61	101
...	...	...

(c) Parameter-Count table

Figure 3.2: Preprocessing for the query plan with a single parameter

The output of the analysis is a set of parameter(s) values and the corresponding intermediate result sizes produced by every join of the query plan. Second, the output of the data analysis stage is processed by the *greedy algorithm* that selects the subset of parameters resulting in the minimal variance across all intermediate result sizes.

3.4.1 Single Parameter

Data Analysis The goal of this stage is to compute all the intermediate results in the query plan for each value of the parameter. We will store this information as a *Parameter-Count* (*PC*) table, where rows correspond to parameter values, and columns – to a specific join's result sizes.

There are two ways of computing that table. First, given the query plan tree we can split it into a bottom-up manner starting with the smallest subtree that contains the parameter. We will then remove the selection on the parameter value from the query, and add a Group-By on the parameter name with a Count, thus effectively aggregating the result size of that subtree across the parameter domain. In our experiments with LDBC Social Network benchmark we were generating group-by queries based on the JSON representation of the query plan.

The second way of computing the Parameter-Count table is to compute the corresponding counts as part of data generation. Indeed, in case of the LDBC SNB, for instance, all the group-by queries boil down to counting the number of generated entities: number of friends per person, number of posts per user etc. These counts are later used to generate parameters across multiple queries.

As an example, consider a simplified version of LDBC Interactive Query 2, given in Listing 3.2, which extracts 20 posts of the given user's friends ordered by their timestamps. The generated plans with Group-By's on top are depicted in Figure 3.2a and b. The first subquery plan counts the number of friends per person, the second one aggregates the number of posts of all friends by user. The resulting Parameter-Count table is given in Figure 3.2c, where columns named $|\Gamma^1|$ and $|\Gamma^2|$ correspond to the results of the first and second group-by queries, respectively. In other words, when executed with *%ParameterID* = 1542, Query 2 will generate $60 + 99 = 159$ intermediate result tuples.

Query 3.2: LDBC SNB Interactive Query 2

```
SELECT p_personid, ps_postid, ps_creationdate
FROM person, post, knows
WHERE
    person.p_personid = post.ps_creatorid AND
    knows.k_person1id = %Person% AND
    knows.k_person2id = person.p_personid
ORDER BY ps_creationdate DESC
LIMIT 20
```

Algorithm 1: PARAMETER CURATION (SINGLE PARAMETER)

```
FINDWINDOWS
Input: $PC$ – Parameter-Count table, $i$ – column, $start, end$ – offsets in the table
1  begin
2      scan the $PC$ table on the $i$th column from $start$ to $end$ rows
3      $W \leftarrow$ generate Windows of size $K$
4      merge overlapping windows with the same variance
5      return $w \in W$ with the smallest variance of $PC[i]$ values
6  PARAMETERCURATION
Input: $PC$ – Parameter-Count table, $n$ – number of count columns in $PC$
Result: $\mathbb{W}$ – window in $PC$ table with the smallest variance of counts across all columns
7  begin
8      $i \leftarrow 1$ ▷ corresponds to the column number in the table, i.e. $|\Gamma^i|$
9      $\mathbb{W} \leftarrow \text{FINDWINDOWS}(PC, 1, 0, |PC|)$ ▷ find windows on the entire first column
10     while $|\mathbb{W} > 1|$ and $i < n$ do
11         $i \leftarrow i + 1$
12         $\mathbb{W}_{new} \leftarrow$ list()
13         for $w \in \mathbb{W}$ do
14             $w' \leftarrow \text{FINDWINDOWS}(PC, i, w.start, w.end)$
15             $\mathbb{W}_{new}$.add($w'$)
16         sort $\mathbb{W}_{new}$ by variance asc
17         $\mathbb{W} \leftarrow$ all $w \in \mathbb{W}_{new}$ with the smallest variance
18     return $\mathbb{W}$
```

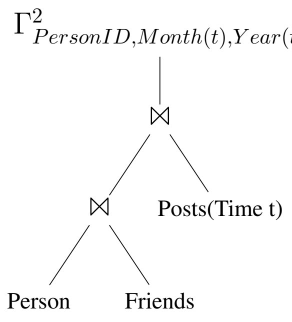
Greedy Algorithm. Now, our goal is to find the part of the Parameter-Count table with the smallest variance across all columns. Note that the order of the columns matters; in other words, variance in the first column (result size of the bottom-most join of the query plan) is more crucial to the runtime behaviour than variance in the last column (top-most join). Following this observation, we construct a simple greedy algorithm, depicted in Algorithm 7. It uses an auxiliary function `FindWindows` that finds the *windows* (consecutive rows of the table) of size at least k on a given column i with the smallest possible variance (lines 3-4). In our table in Figure 3.2c such windows on the first column ($|\Gamma^1|$) are highlighted with red and green colors (they consist of parameter sets $[1542, 1673, 7511]$ and $[958, 1367]$, respectively). Both these sets have variance 0 in the column $|\Gamma^1|$.

The algorithm starts with finding the windows $\mathbb{W}$ with the smallest variance on the entire first column (line 9). Then, in every found window from $\mathbb{W}$ we look for smaller sub-windows (but of size at least than k, see line 3) that minimize variance on the second column (lines 12-16). The found windows with the smallest variance become candidates for the next iteration, based on further columns (line 17). The process stops when we reach the last column or the number of candidate windows reduces to 1.

In the example from Figure 3.2c, the first iteration brings the two windows mentioned above (red and green). Then, in every window we look for windows of k rows, they are $[99, 102]$, $[102, 103]$ and $[120, 101]$. Out of these three candidates, $[102, 103]$ has the smallest variance (highlighted in blue), so our solution consists of two parameters, $[1673, 7511]$.

3.4.2 Two correlated parameters

Here we consider the case when a query has two parameters, discrete and continuous, e.g. *PersonID* and *Timestamp*. The continuous parameter is involved in a selection, e.g. specifying the time interval. We focus on the situation when these two are correlated, otherwise the solution of the Parameter Curation problem is a straightforward generalization of the previous case: one would follow the independence assumption and find



(a) # Posts of Friends By Month

PersonID	Jan'14	Feb'14	Mar'14	Apr'14
...	...	...	...	...
1673	0	30	30	42
7511	20	30	30	23
...	...	...	...	...

(b) Parameter-Count table with Time buckets ($PCTime$)

Figure 3.3: Preprocessing for the query plan with two correlated parameters

the bindings for the discrete parameter using Parameter-Count table, and then select intervals of the same length as bindings of the continuous parameter.

However, if parameters are correlated, the independence assumption may lead to a significant skew in the C_{out} function values. We take the LDBC Interactive Query 2 as an example again, which in its full form also includes the selection on the timestamp of the posts `ps_creationdate < %Date0%` (i.e., the query *finds the top 20 posts of friends of a user written before a certain date*). In the LDBC dataset, the *PersonID* and *Timestamp* of the user's posts are naturally correlated, since users join the modeled social network at different times; moreover, their posting activity changes over time. Therefore, if we choose the Timestamp parameter in LDBC Interactive Query 2 independently from the *PersonID*, the amount of intermediate results may vary significantly (even if ParameterIDs were curated such that the total number of posts is the same).

Data analysis. In order to capture the correlation between two parameters, we need to include the second one (Timestamp in our example) in the grouping key during the Parameter-Count table construction. Grouping by the continuous parameter may lead to a very large and sparse table, so we "bucketize" it (e.g., by months and years for Timestamp). We then store the results of the aggregation as a Parameter-Count table, along with the bucket boundaries.

Our example from Figure 3.2 is extended with the Timestamp parameter in Figure 3.3. The partial join trees are complemented with additional Group-By on Month and Year of the timestamp as soon as the corresponding table containing the Timestamp (in our case *Posts*) is added to the plan (in this example, at Step 2 when we consider the second join). Assuming that our dataset spans 4 months of 2014, the resulting table may look like Figure 3.3b.

Greedy algorithm. The first stage of the Parameter Curation for two parameters ignores the continuous parameter (e.g. Timestamp). As a result, we get the bindings for the first (discrete) parameter that have similar intermediate result sizes across the entire domain of the continuous parameter. Now for these curated parameter bindings we find the corresponding continuous parameters such that the C_{out} function values are similar across all the curated parameters.

For the purpose of presentation we consider the solution for the *%Date0* parameter that appears in the selection of a form $timestamp < \%Date0$. In our example from the previous section, we have found two *PersonID* parameters that have the smallest variance in C_{out}. Let $PCTime[i, j]$ denote the count in the Parameter-Count table for the parameter i in bucket j, and N be the number of buckets for continuous parameter. For example, in Figure 3.3b $PCTime[1673, Mar'14] = 30$ is the number of posts made by friends of the user 1673 in March 2014, and $N = 4$.

- We compute the partial sums of the monthly counts $Sum[i] = \sum_{j=1..N-M} PCTime[i, j]$ for all the discrete parameter bindings i for all the months except the last M (where M is typically 1..3). In the table in Figure 3.3b for $M = 1$ these partial sums are 60 and 80 for *PersonID*s 1673 and 7511, respectively.

- We determine the average $\mathcal{A}$ across these sums $Sum[i]$ (70 in our example)
- For every discrete parameter i we pick the bucket J such that $\sum_{j=1..J} PCTime[i,j]$ is as close as possible to the global average $\mathcal{A}$. More precisely, we pick the first bucket such that the sum exceeds the global average. In our example, for $i = 1673$, J is the fourth bucket (*Apr'14*)
- Finally, since our buckets represent continuous variable (time), we can split the bucket J so that the sum of counts is *exactly* $\mathcal{A}$. For $i = 1673$ we need to get 10 posts in April 2014 (60 are covered by previous months, and we need to reach the global average of 70). We pick April $\frac{42 \cdot 10}{30} = 14$ as $Date0$.

In order to perform the last step in the above computation, we have assumed that within one bucket the count is uniformly distributed (e.g., every day within one month has the same number of posts). Even when this assumption does not hold precisely, the effects are usually negligible.

The timestamp conditions of a different form, e.g. $Timestamp > Date0$, or $Timestamp \in [Date0, Date1]$ are handled in the same manner. For example, the $Timestamp \in [Date0, Date1]$ condition leads to finding for every $PersonID$ the median of its post-per-time distribution, that is the median of the $PCTable[i,j]$ for every row i. Then, the median of those medians is identified across all *PersonID*s, and finally every individual *PersonID*'s median is made as close as possible to the global median by extending/reducing the corresponding bucket.

3.4.3 Multiple correlated parameters

Parameter Curation for multiple (more than two) parameters follows the scheme of two parameters: one is selected as a primary (*PersonID*), the other ones are "bucketized". This way we get sets of bindings, each of those results in identical query plan and similar runtime behavior.

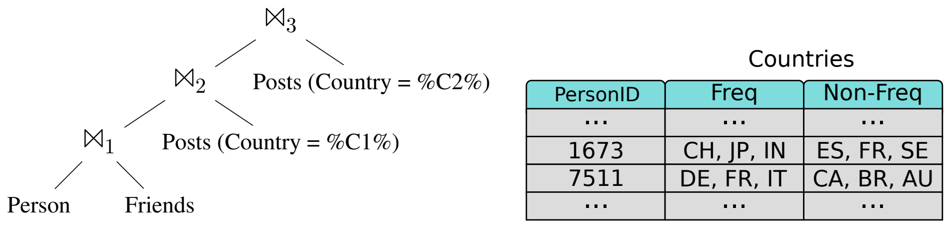
In case of correlated parameters, however, it may be interesting to find several sets of parameter bindings that would yield different query plans (but consistent within one set of bindings). Consider the simplified version of LDBC Interactive Query 3 that is *finding the friends of a user that have been to countries %C1 and %C2 and logged in from that countries (i.e., made posts)*, given in Query 3.3 and its query plan in Figure 3.4a.

Query 3.3: LDBC SNB Interactive Query 3

```
SELECT k.k_person2id, ps_postid, ps_creationdate
FROM person p, knows k, post p1, post p2
WHERE p.person_id = k.k_person1id
               AND k.k_person2id = p1.p_personid
               AND k.k_person2id = p2.p_personid
               AND p1.place = '%C1%'
               AND p2.place = '%C2%'
ORDER BY ps_creationdate DESC
LIMIT 20
```

Since in the generated LDBC SNB dataset the country of the person is correlated with the country of his friends, and users tend to travel to (i.e. post from) neighboring countries, there are essentially two groups of countries for every user: first, the country of his residence and neighboring countries; second, any other country. For parameters from first group the join denoted $\bowtie_2$ in Figure 3.4a becomes very unselective, since almost all friends of the user are likely to post from that the country. For the second group, both $\bowtie_2$ and $\bowtie_3$ are very selective. In the intermediate case when parameters are taken from the two different groups, it additionally influences the order of $\bowtie_2$ and $\bowtie_3$.

Both these groups of parameters are based on counts of posts made by friends of a user, i.e. based on the counts collected in the Parameter-Count table (with additional group-by on country of the post). Instead of keeping the buckets of all countries, we group them into two larger buckets based on their count, *Frequent* and *Non-Frequent* as shown in Figure 3.4b.



PersonID	Freq	Non-Freq
...	...	...
1673	CH, JP, IN	ES, FR, SE
7511	DE, FR, IT	CA, BR, AU
...	...	...

(a) Query Plan (b) Parameter-Count table

Figure 3.4: Case of multiple correlated parameters

Query	1	2	3	4	5	6	7
Curated	13	31	243	0.6	1300	6931	33
Random	773	2165	444174	$184 \cdot 10^6$	$52 \cdot 10^6$	278173	362

Query	8	9	10	11	12	13	14
Curated	0.18	99269	4073	1	95	2977	5107
Random	403	880287	102852	39	1535	26777	155032

Table 3.1: Variance of runtimes: Uniformly sampled parameters vs Curated parameters for the LDBC Benchmark queries

Now we can essentially split the LDBC Interactive Query 3 into three different (related) query variants (a), b) and c)), based on the combination of the two *%Country* parameters: a) *%C1* and *%C2* from the *Frequent* group, b) both from *Non-Frequent* group, c) combination of the two above.

3.5 Experiments

In this section we describe our experiments with curated parameters in the LDBC benchmark. First, we compare the runtimes of query templates with curated parameters as opposed to randomly selected ones (Section 3.5.1). Then we proceede with an experiment on curating parameters for different intended plans of the same query template in Section. All experiments are run with Virtuoso 7 Column Store as a relational engine on a commodity server.

3.5.1 Curated vs Uniformly Sampled Parameters

First experiment aims at comparing the runtime variance of the LDBC queries with curated parameters with the randomly sampled parameters. For all 14 queries we curated 500 parameters and sampled randomly the same amount of parameters for every query. We run every query template with each parameter binding for 10 times and record the mean runtime. Then, the compute the runtime variance per query for curated and random parameters. The results, given in Table 3.1, indicate that Parameter Curation reduces the variance of runtime by a factor of at least 10 (and up to several orders of magnitude). We note that some queries are more prone to runtime variability (such as Query 4 and 5), that is why the variance reduction is different accross the query set. For Query 4 we additionally report the runtime distribution of query runs with curated and random parameters in Figure 3.5.

3.5.2 Groups of Parameters for One Query

So far we have considered the scenario when the *intended query plan* needs to be supplied with parameters that provide the smallest variance to its runtime. For some queries, however, there could be multiple intended plan *variants*, especially when the query contains a group of correlated parameters. As an example, take LDBC Query 11 that *finds all the friends of friends of a given person P that work in country X*. The data generator

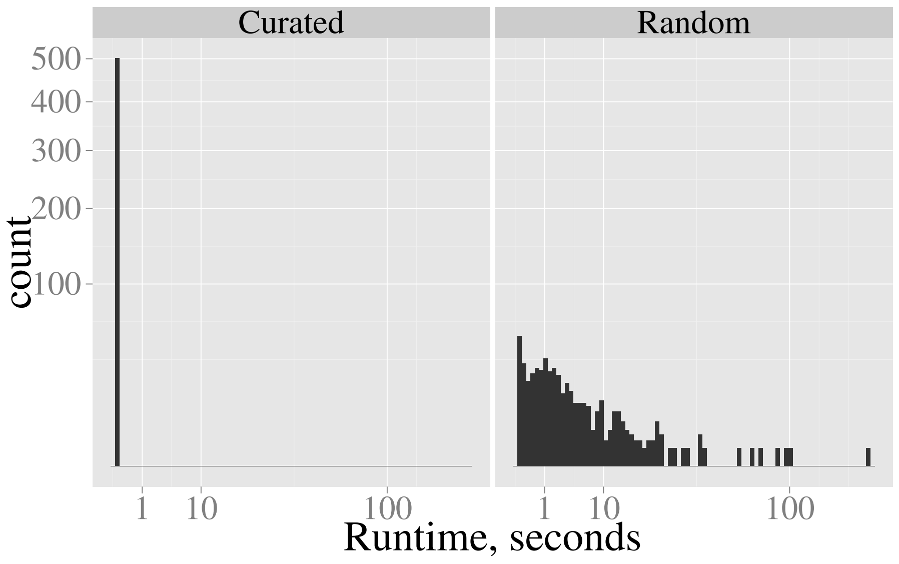


Figure 3.5: LDBC Query 4 Runtime Distribution: Curated vs Random parameters

guarantees that the location of friends is correlated with the location of a user. Naturally, when the country *X* is the user's country of residence, the amount of intermediate results is much higher than for any other country. Moreover, if *X* is a non-populous country, the reasonable plan would be to start from finding all the people that work at organizations in *X* and then figure out which of them are friends of friends of the user *P*.

As described in Section 3.4.3, our algorithm provides three sets of parameters for the three intended query plans that arise in the following situations: (i) P resides in the country X, (ii) country X is different than the residence country of P, (iii) X is a non-populous country that is not a residence country for P. As a specific example, we consider a set of Chinese users with countries (i) China, (ii) Canada, (iii) Zimbabwe. The corresponding average runtimes and standard deviations are depicted in Figure 3.6. We see that the three groups indeed have distinct runtime behavior, and the runtime within the group is very similar. For comparison, we also provide the runtime distribution for a randomly chosen country parameter, which is far from the normal distribution.

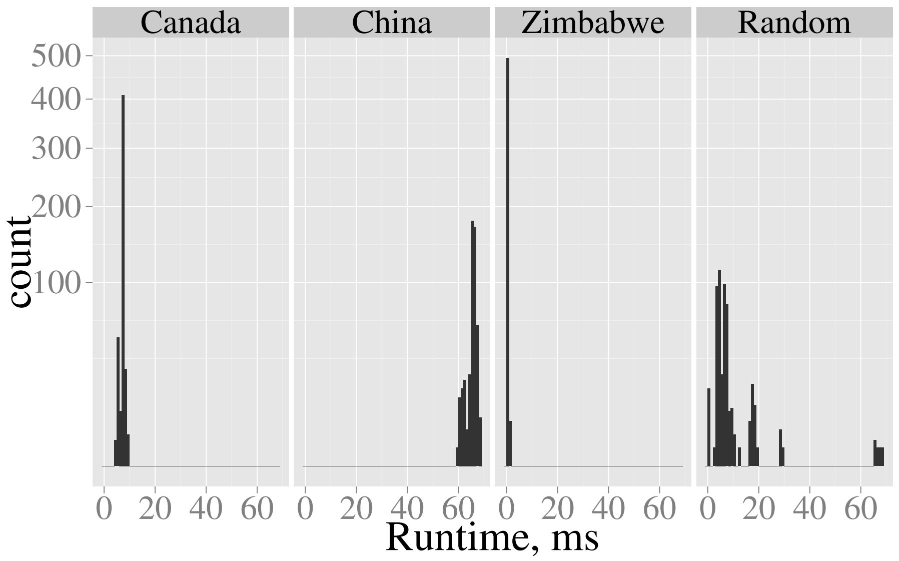


Figure 3.6: LDBC Query 11 with four different groups of parameters (for countries China, Canada, Zimbabwe, Random)

Scale	Parameter Extraction Time	% of Total Generation Time	Data Size, Gb
10K	17 s	7 %	1
50K	125 s	11 %	5.5
1Mln	4329 s	12 %	227

Table 3.2: Time to extract parameters in the LDBC datasets of different scales

3.5.3 Parameter Curation time

Finally, we report the runtime of the parameter curation procedure for the LDBC Benchmark. Note that we have incorporated the data analysis stage in our case is implemented as part of data generation, e.g. we keep the number of posts per person generated, number of replies to the user's posts etc. This is done with a negligible runtime overhead. In Table 3.2 we report the runtime of the greedy parameter extraction procedure for the LDBC dataset of different scales (as number of persons in the generated social network). We additionally show the size of the generated data; this is essentially an indicator of the amount of data that the extraction procedure needs to deal with. We see that Parameter Curation takes approximately 7% to 12% of the total data generation time, which looks like a reasonable overhead.

4 Conclusion

This deliverable presented a choke-point-based design of the Business Intelligence workload for the LDBC Social Network Benchmark. We discussed the technical challenges that a DBMS needs to address in order to efficiently run complex queries, and sketched a set of 21 queries that contain these challenges.

In the second part of the deliverable we motivated and introduced *Parameter Curation*: a data mining-like process that follows data generation in a database benchmarking process. Parameter Curation finds substitution parameters for query templates that produces query invocations with very small variation in the size of the intermediate query results, and consequently, similar running times and query plans. This technique is needed when designing *understandable* benchmark query workloads for datasets with skewed and correlated data, such as found in real-world datasets. Parameter Curation was developed and is in fact used as part of the LDBC Social Network Benchmark (SNB) [1], whose data generator produces a social network with a highly skewed power-law distributions and small diameter network structure, that has as additional characteristic that both the attribute values and the network structure are highly correlated. Our results show that Parameter Curation in these skewed and correlated datasets transforms chaotic performance behavior for the same query template with randomly chosen substitution parameters into highly stable behavior for curated parameters. Parameter Curation retains the possibility for benchmark designers to test the ability of query optimizers to identify different query plans in case of skew and correlation, by grouping parameters with the same behavior into a limited number of classes which among them have very different behavior; hence creating multiple *variants* of the same query template.

[1] See http://github.com/ldbc and http://ldbcouncil.org

A Set of SQL queries

Common view definitions

```
CREATE VIEW country AS SELECT city.pl_placeid AS ctry_city,
   ctry.pl_name AS ctry_name
   FROM place city, place ctry
   WHERE city.pl_containerplaceid = ctry.pl_placeid
   AND ctry.pl_type = 'country';
```

Query 1: top 100 popular topics in each country, age and gender group

```
CREATE procedure p_age_group (IN bday DATE) returns INT
{
  return floor (datediff ('year', bday, CAST ('2014-1-1' AS DATE)) / 5);
}


SELECT top 100 ctry_name, MONTH (ps_creationdate) AS mm, p_gender,
        p_age_group (p_birthday) AS age, t_name, COUNT (*) AS cnt
FROM person, post, post_tag, tag, country
WHERE ps_creatorid = p_personid
        AND p_placeid = ctry_city AND
        pst_postid = ps_postid AND t_tagid = pst_tagid
        AND ps_creationdate BETWEEN CAST ('2012-1-1' AS DATE)
                        AND dateadd ('year', 1, CAST ('2012-1-1'AS DATE))
GROUP BY ctry_name, mm, p_gender, age, t_name
HAVING cnt > 1000
ORDER BY ctry_name, mm, t_name, age, p_gender;
```

Query 2: find new tags that appeared during last month

```
SELECT top 100 m1.t_name, cnt1, cnt2, cnt2 - cnt1 AS diff
FROM
        (SELECT t_name, COUNT (*) AS cnt1
                FROM post, post_tag, tag
                WHERE t_tagid = pst_tagid AND pst_postid = ps_postid
                AND ps_creationdate BETWEEN CAST ('2012-6-1' AS DATE)
                        AND dateadd ('month', 1, CAST ('2012-6-1' AS DATE))
                GROUP BY t_name) m1,

        (SELECT t_name, COUNT (*) AS cnt2
                FROM post, post_tag, tag
                WHERE t_tagid = pst_tagid AND pst_postid = ps_postid
                AND ps_creationdate BETWEEN dateadd ('month', 1,
                                CAST ('2012-6-1' AS DATE))
                        AND dateadd ('month', 2, CAST ('2012-6-1' AS DATE))
                GROUP BY t_name) m2
WHERE m1.t_name = m2.t_name
ORDER BY diff DESC;
```

Query 3: find most relevant forums on a given topic in a given country

```
SELECT top 20 f_forumid, f_title, f_creationdate,
                f_moderatorid, COUNT (*) AS cnt
FROM forum, post, post_tag,
                tag_tagclass, tagclass, person, country
WHERE tc_name = 'Musical_Artist'
         AND ttc_tagclassid = tc_tagclassid
         AND ttc_tagid = pst_tagid AND pst_postid = ps_postid
         AND ps_forumid = f_forumid AND f_moderatorid = p_personid
         AND p_placeid = ctry_city AND ctry_name = 'India'
GROUP BY f_forumid, f_title, f_creationdate, f_moderatorid
ORDER BY  cnt DESC;
```

Query 4: top posters in 100 top forums in China

```
SELECT top 100 p_personid, p_firstname,
                p_lastname, p_creationdate, COUNT (*)
FROM person, post, forum f, forum_person,
        (SELECT top 100 f_forumid, COUNT (*) AS cnt
                FROM forum, forum_person, person, country
                WHERE f_forumid = fp_forumid
                AND p_personid = fp_personid
                AND p_placeid = ctry_city AND ctry_name = 'China'
                GROUP BY f_forumid ORDER BY cnt DESC) tf
WHERE ps_creatorid = p_personid
        AND ps_forumid = f.f_forumid
        AND f.f_forumid = tf.f_forumid
        AND p_personid = ps_creatorid
GROUP BY p_personid
ORDER BY cnt DESC;
```

Query 5: top posters on a given topic

```
SELECT top 100 p_personid, n_posts, n_replies,
        n_likes, n_posts + 2 * n_replies + 10 * n_likes AS sc
FROM (
        SELECT p_personid, COUNT (*) AS n_posts,
                SUM (n_lks) AS n_likes, SUM (n_reps) AS n_replies
                FROM (
                        SELECT p_personid, ps_postid,
                        (SELECT COUNT (*) FROM likes
                                WHERE l_postid = ps_postid) AS n_lks,
                        (SELECT COUNT (*) FROM post p2
                                WHERE p2.ps_replyof = p1.ps_postid) AS n_reps
                        FROM person, post p1, post_tag, tag
                        WHERE ps_creatorid = p_personid
                        AND pst_postid = ps_postid
                        AND t_tagid = pst_tagid
                        AND t_name = 'Winston_Churchill'
                        ) psts
                GROUP BY p_personid) sums
ORDER BY sc DESC;
```

Query 6: most authoritative user posting on a given topic

```
SELECT top 100 ps_creatorid, SUM (likeauth.auth) AS sc
```

```
FROM post, tag, post_tag, likes l,
        (SELECT ps_creatorid AS liker,  COUNT (*) AS auth
        FROM post, likes
        WHERE ps_postid = l_postid
        GROUP BY ps_creatorid) likeauth
WHERE liker = l_personid AND l_postid = ps_postid
        AND pst_postid = ps_postid
        AND   pst_tagid = t_tagid AND t_name = ’Augustine_of_Hippo’
GROUP BY ps_creatorid
ORDER BY sc DESC;
```

Query 7: tags that are most frequently mentioned together with the given tag

```
SELECT top 100 t_name, COUNT (*) AS cnt
FROM post ps, post rep, post_tag, tag
WHERE rep.ps_replyof = ps.ps_postid
        AND EXISTS (SELECT 1 FROM post_tag, tag
                WHERE pst_postid = ps.ps_postid
                                AND pst_tagid = t_tagid
                                AND t_name = ’Augustine_of_Hippo’)
        AND NOT EXISTS (SELECT 1 FROM post_tag, tag
                WHERE pst_postid = rep.ps_postid
                                AND pst_tagid = t_tagid
                                AND t_name = ’Augustine_of_Hippo’)
        AND pst_postid = rep.ps_postid AND t_tagid = pst_tagid
GROUP BY t_name
ORDER BY cnt DESC;
```

Query 8: anticorrelation between two tags’ occurance

```
SELECT top 200 f_forumid, SUM (competing) AS comp, SUM (ours) AS ours2
FROM (SELECT f_forumid,
  (SELECT COUNT (*)
         FROM post
         WHERE ps_forumid = f_forumid
          AND EXISTS (SELECT 1 FROM post_tag WHERE pst_postid = ps_postid
      AND pst_tagid IN (SELECT ttc_tagid FROM tagclass, tag_tagclass
        WHERE ttc_tagclassid = tc_tagclassid
          AND tc_name = ’MusicalArtist’))) AS competing,
  (SELECT COUNT (*)
    FROM post
         WHERE ps_forumid = f_forumid
                AND EXISTS (SELECT 1 FROM post_tag WHERE pst_postid = ps_postid
          AND pst_tagid IN (SELECT ttc_tagid FROM tagclass, tag_tagclass
                WHERE ttc_tagclassid = tc_tagclassid
                 AND tc_name = ’Writer’))) AS ours
FROM forum
WHERE (SELECT COUNT (*) FROM forum_person WHERE fp_forumid = f_forumid) > 200)
mindshare
WHERE competing > 0 AND ours > 0
GROUP BY f_forumid  ORDER BY comp - ours2 DESC;
```

Query 9: find the most central user for the given tag

```
CREATE VIEW person_tag_rel AS
```

```
SELECT tr_personid, tr_tag, SUM (sc) AS tr_score
FROM (
  SELECT pt_personid AS tr_personid, t_name AS tr_tag, 100 AS sc
  FROM person_tag, tag
  WHERE  t_tagid = pt_tagid
  UNION ALL SELECT ps_creatorid, t_name, 1 AS sc
  FROM post, post_tag, tag
  WHERE pst_postid = ps_postid AND pst_tagid = t_tagid) ff
GROUP BY tr_personid, tr_tag
;


SELECT top 100 p1.p_personid, SUM (tr1.tr_score + tr2.tr_score) AS sc
FROM person p1, knows, person p2,
  person_tag_rel tr1, person_tag_rel tr2
WHERE tr1.tr_tag = 'Franz_Kafka'
  AND tr2.tr_tag = 'Franz_Kafka'
  AND tr1.tr_personid = p1.p_personid
  AND tr2.tr_personid = p2.p_personid
  AND p1.p_personid = k_person1id AND p2.p_personid = k_person2id
GROUP BY p1.p_personid
ORDER BY sc DESC;
```

Query 10: find people that successfully introduce a new topic

```
SELECT p_personid, t_name, COUNT (*) AS cnt
FROM person, post org, post rep, post_tag rtag, tag, country
WHERE rep.ps_replyof = org.ps_postid
  AND rtag.pst_tagid = rep.ps_postid
  AND NOT EXISTS (SELECT 1
          FROM post_tag orgtag
          WHERE orgtag.pst_postid = org.ps_postid
            AND rtag.pst_tagid = orgtag.pst_tagid)
  AND EXISTS (SELECT 1 FROM likes
          WHERE l_postid = rep.ps_postid)
  AND rep.ps_creatorid = p_personid
  AND p_placeid = ctry_city AND ctry_name = 'Mexico'
  AND rtag.pst_tagid = t_tagid
GROUP BY p_personid, t_name
ORDER BY cnt DESC;
```

Query 11: most liked content of the network in the given period

```
SELECT top 100 ps_postid, p_firstname, p_lastname, ps_creationdate, COUNT (*)
FROM post, person, likes
WHERE ps_postid IN
  (SELECT l_postid, COUNT (*)
        FROM likes
        GROUP BY l_postid
        HAVING COUNT (*) > 100)
  AND p_personid = ps_creatorid
  AND l_postid = ps_postid
  AND ps_creationdate > CAST ('2010-3-1' AS DATE)
GROUP BY ps_postid, p_firstname, p_lastname, ps_creationdate
```

```
ORDER BY ps_creationdate DESC;
```

Query 12: top thread initiators

```
SELECT top 100 p_personid, p_firstname, p_lastname,
  COUNT (*) AS cnt, COUNT (DISTINCT org.ps_postid) AS n_threads
FROM  person,  post org,
  (SELECT transitive t_in (1) t_out (2) r.ps_replyof, r.ps_postid
        FROM post r
        WHERE r.ps_creationdate BETWEEN  CAST ('2011-10-1' AS DATE)
          AND  dateadd ('month', 3, CAST ('2011-10-1' AS DATE) )) reps
WHERE reps.ps_replyof = org.ps_postid AND org.ps_replyof is NULL
  AND org.ps_creatorid = p_personid
  AND org.ps_creationdate BETWEEN  CAST ('2011-10-1' AS DATE)
    AND  dateadd ('month', 3, CAST ('2011-10-1' AS DATE) )
GROUP BY p_personid
ORDER BY cnt DESC, p_personid;
```

Query 13: top 100 people with the number of friends higher than average in China

```
SELECT top 100 p_personid, COUNT (*) AS cnt
FROM person, knows, country
WHERE ctry_name = 'China'
  AND p_placeid = ctry_city AND k_person1id = p_personid
GROUP BY  p_personid
HAVING cnt = floor ((
        SELECT AVG (fcnt)
        FROM (SELECT p_personid, COUNT (*) AS fcnt
                FROM person, knows, country
      WHERE p_placeid = ctry_city AND ctry_name = 'China'
      AND k_person1id = p_personid
      GROUP BY p_personid) ctavg))
ORDER BY p_personid;
```

Query 14: find people that are connected to each other and talk about one topic

```
SELECT top 100 kn.k_person2id, t_name, COUNT (*)  AS cnt
FROM (SELECT transitive t_distinct t_in (1) t_out (2) k_person1id, k_person2id
          FROM knows
     WHERE  k_person2id IN (
          SELECT p_personid
          FROM person, country
        WHERE p_placeid = ctry_city AND ctry_name = 'China')) kn,
post, post_tag, tag, tag_tagclass, tagclass
WHERE ps_postid = pst_postid
  AND t_tagid = pst_tagid
  AND ttc_tagid = pst_tagid
  AND  ttc_tagclassid = tc_tagclassid
  AND tc_name = 'MusicalArtist'
  AND ps_creatorid = kn.k_person2id AND kn.k_person1id = 1030
GROUP BY t_name, kn.k_person2id
ORDER BY cnt DESC, t_name;
```

Query 15: triangle counting

```
SELECT COUNT (*)
```

```
FROM knows k1, knows k2, knows k3
WHERE k1.k_person1id = k3.k_person2id
  AND k1.k_person2id = k2.k_person1id
  AND  k2.k_person2id = k3.k_person1id
  AND k1.k_person1id < k1.k_person2id AND k2.k_person1id < k2.k_person2id
  AND k1.k_person1id IN (
          SELECT p_personid
          FROM person, place city, place ctry
          WHERE p_placeid = city.pl_placeid
             AND city.pl_containerplaceid = ctry.pl_placeid
             AND ctry.pl_name = 'India' AND ctry.pl_type = 'country')
  AND k2.k_person1id IN (
          SELECT p_personid
          FROM person, place city, place ctry
          WHERE p_placeid = city.pl_placeid
              AND city.pl_containerplaceid = ctry.pl_placeid
              AND ctry.pl_name = 'India'  AND ctry.pl_type = 'country')
  AND k3.k_person1id IN (
          SELECT p_personid
          FROM person, place city, place ctry
          WHERE p_placeid = city.pl_placeid
              AND city.pl_containerplaceid = ctry.pl_placeid
              AND ctry.pl_name = 'India'  AND ctry.pl_type = 'country')
;
```

Query 16: distribution of number of posts

```
SELECT cnt, COUNT (*) AS n
FROM (SELECT p_personid, COUNT (ps_postid) AS cnt
          FROM person LEFT  JOIN post ON ps_creatorid = p_personid
          AND ps_creationdate > CAST ('2012-1-1' AS DATE)
GROUP BY p_personid) post_cnt
GROUP BY cnt
ORDER BY cnt DESC, n;
```

Query 17: communication between strangers

```
SELECT top 10 p_personid, COUNT (*) AS cnt,
  SUM (CASE WHEN  EXISTS (SELECT 1 FROM knows
            WHERE k_person1id = teen.p_personid
            AND k_person2id = org.ps_creatorid)
       THEN 0 ELSE 1 END) AS strangercnt
FROM person teen, post org, post rep
WHERE p_birthday > CAST ('2000-1-1' AS DATE)
  AND rep.ps_creatorid = teen.p_personid
  AND org.ps_postid = rep.ps_replyof
GROUP BY teen.p_personid
ORDER BY cnt DESC;
```

Query 18: frequency of topics

```
SELECT  tc_name, COUNT (*) AS cnt
FROM tagclass, tag_tagclass, tag, post_tag
WHERE tc_tagclassid = ttc_tagclassid AND ttc_tagid = t_tagid
        AND pst_tagid = ttc_tagid
```

```
GROUP BY tc_name ORDER BY cnt DESC;
```

Query 19: Zombies: People who are liked by people who produce nothing

```
SELECT top 100 liked, COUNT (*) AS n_likers, SUM (likerposts) AS liker_posts
FROM (
         SELECT liked.p_personid AS liked,
           (SELECT  COUNT (*)
            FROM post lrpost
            WHERE lrpost.ps_creatorid = liker.p_personid) AS likerposts
    FROM person liked, post, likes, person liker, country
    WHERE  ps_creationdate BETWEEN
       CAST ('2012-4-1' AS DATE)
       AND dateadd ('month', 3, CAST ('2012-4-1' AS DATE))
    AND ps_creatorid = liked.p_personid
    AND liked.p_placeid = ctry_city AND ctry_name = 'India'
    AND l_postid = ps_postid AND l_personid = liker.p_personid) lks
GROUP BY liked
ORDER BY CAST (n_likers AS real) / liker_posts DESC;
```

Query 20: correspondence between two countries

```
CREATE VIEW related AS
SELECT rep.ps_creatorid AS p1,
  org.ps_creatorid AS p2, 4 AS score
  FROM post org, post rep
  WHERE rep.ps_replyof = org.ps_postid
UNION ALL
SELECT org.ps_creatorid AS p1,
  rep.ps_creatorid AS p2, 1 AS score
  FROM post org, post rep
  WHERE rep.ps_replyof = org.ps_postid
UNION ALL
SELECT k_person1id, k_person2id, 15 AS score
  FROM knows
UNION ALL
  SELECT l_personid, ps_creatorid, 10
  FROM likes, post
  WHERE l_postid = ps_postid
UNION ALL
  SELECT ps_creatorid, l_personid, 1
  FROM likes, post
  WHERE l_postid = ps_postid;

SELECT contact.p_personid, contact.p_firstname, contact.p_lastname, SUM (score)
FROM person contact, related,
     person contacted, country target, country source
WHERE contact.p_personid = p1 AND contacted.p_personid = p2
   AND contacted.p_placeid = target.ctry_city
   AND target.ctry_name = 'Yemen'
   AND contact.p_placeid = source.ctry_city
   AND source.ctry_name = 'United_States'
GROUP BY contact.p_personid, contact.p_firstname, contact.p_lastname
ORDER BY score DESC;
```

Query 21: tag timeline

```
SELECT YEAR (ps_creationdate) AS yy,
  MONTH (ps_creationdate) AS mm, continent, COUNT (*) AS n_posts,
  SUM ((SELECT COUNT (*) FROM likes WHERE l_postid = ps_postid)) AS n_likes
FROM post JOIN post_tag ON pst_postid = ps_postid
  LEFT JOIN (SELECT cont.pl_name AS continent, ctry.pl_placeid AS pl
          FROM place cont, place ctry
          WHERE  cont.pl_placeid = ctry.pl_containerplaceid) ppl
  ON pl = ps_locationid
WHERE pst_tagid IN (
  SELECT ttc_tagid
  FROM  tag_tagclass, tagclass
  WHERE tc_name = 'Politician'
  AND ttc_tagclassid = tc_tagclassid)
GROUP BY yy, mm, continent
ORDER BY yy, mm, continent
```

REFERENCES

[1] LDBC Benchmark. http://ldbc.eu:8090/display/TUC/Interactive+Workload.

[2] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, 2013.

[3] P. Minh Duc, P. A. Boncz, and O Erling. S3g2: A Scalable Structure-Correlated Social Graph Generator. In *Proceedings of TPC Technology Conference on Performance Evaluation & Benchmarking 2012*, 2012.

[4] Guido Moerkotte. Building Query Compilers. http://pi3.informatik.uni-mannheim.de/ moer/-querycompiler.pdf.

[5] Meikel Poess and John M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1045–1053. VLDB Endowment, 2004.

[6] P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Complex query decorrelation. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 450–458, Feb 1996.

[7] John M. Stephens and Meikel Poess. Mudd: a multi-dimensional data generator. *SIGSOFT Softw. Eng. Notes*, 29(1):104–109, January 2004.