Parameter Curation for Benchmark Queries

Andrey Gubichev[1] and Peter Boncz[2]

[1] TU Munich gubichev@in.tum.de
[2] CWI P.Boncz@cwi.nl

Abstract. In this paper we consider the problem of generating parameters for benchmark queries so these have stable behavior despite being executed on datasets (real-world or synthetic) with skewed data distributions and value correlations. We show that uniform random sampling of the substitution parameters is not well suited for such benchmarks, since it results in unpredictable runtime behavior of queries. We present our approach of *Parameter Curation* with the goal of selecting parameter bindings that have consistently low-variance intermediate query result sizes throughout the query plan. Our solution is illustrated with IMDB data and the recently proposed LDBC Social Network Benchmark (SNB). [3]

1 Introduction

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 1.1: 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

[3] Partially supported by EU project LDBC (FP7-317548), see http://ldbc.eu

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, like handling multiple interesting orders or sparse foreign key joins (in the LDBC project, we call such technical challenges " choke points" [3]).

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 [6, 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 [4]). 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 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 paper 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.

Outline. The rest of the paper is organized as follows. In Section 2 we demonstrate in examples that the straightforward approach of generating parameter bindings uniformly at random fails to deliver predictable and stable results. Section 3 formalizes the problem of *curating parameters* that would yield runtime distribution satisfying properties **P1** - **P3**. In Section 4 we present our implementation of Parameter, used in the LDBC Social Network Benchmark (SNB). Section 5 describes the set of experiments we conducted on SNB and IMDB queries. Section 6 summarizes and concludes the paper.

2 Examples

We use the recently proposed LDBC Social Network Benchmark [1] and a query on IMDB dataset (Query 1.1). For the LDBC Benchmark, 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).

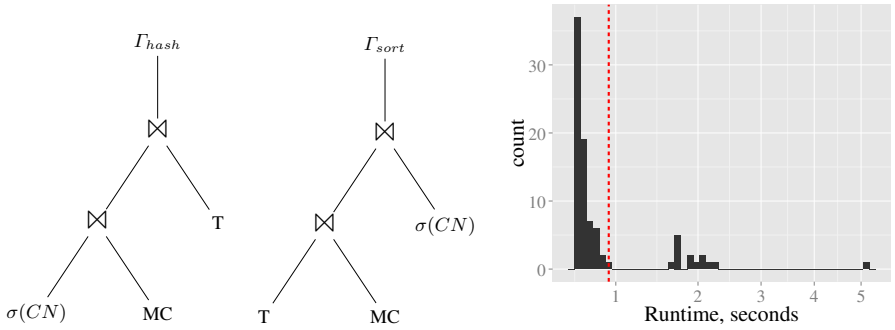
In the following examples (**E1-E4**), we illustrate our statement that uniform selection of parameters leads to unpredictable behavior of queries, which makes interpretation of benchmark results difficult.

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 Listing 1.1, 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 1.1 (as found by the PostgreSQL database) are depicted in Figure 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).

As another example, we consider LDBC 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.



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

Fig. 1: IMDB Query 1.1 plans and runtime distribution for different parameters

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 1.1. Figure 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 benchmark 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).

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-H benchmark record results are improved, this often only

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

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 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 [5]:

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

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 $\sum_{\forall T_{lq} \in L_Q} \textit{Variance}_{\forall p \in S}\, C_{out}(T_{lq}(p))$ 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 1 hold within the set of parameters S and effects mentioned in Section 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).

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). 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.

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 benchmark we were generating group-by queries based on the JSON representation of the query plan.

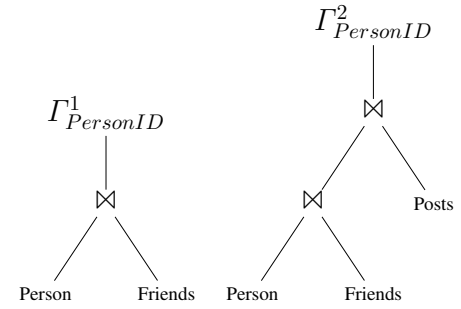
The second way of computing the Parameter-Count table is to compute the corresponding counts as part of data generation. Indeed, in case of the LDBC benchmark, for instance, all the group-by queries boil down to counting the number of generated entities: number of friends per person, number of posts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

As an example, consider a simplified version of LDBC Query 2, given in Listing 1.2, 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 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 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 1.2: LDBC 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
```

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 2c such windows on the first column ($|\Gamma^1|$) are highlighted with red and green colors (they consist of parameter



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

(a) Step 1: # Friends per Person (b) Step 2: # Posts of Friends (c) Parameter-Count table

Fig. 2: Preprocessing for the query plan with a single 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 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, $[1673, 7511]$.

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 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 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

Algorithm 1: PARAMETER CURATION (SINGLE PARAMETER)

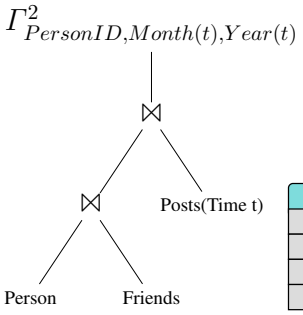
```
FINDWINDOWS
Input: PC – Parameter-Count table, i – column, start, end – offsets in the table
begin
    scan the PC table on the ith column from start to end rows
    W ← generate Windows of size K
    merge overlapping windows with the same variance
    return w ∈ W with the smallest variance of PC[i] values
PARAMETERCURATION
Input: PC – Parameter-Count table, n – number of count columns in PC
Result: 𝕎 – window in PC table with the smallest variance of counts across all columns
begin
    i ← 1 ▷ corresponds to the column number in the table, i.e. |Γ^i|
    𝕎 ← FINDWINDOWS(PC, 1, 0, |PC|) ▷ find windows on the entire first column
    while |𝕎 > 1| and i < n do
        i ← i + 1
        𝕎_new ← list()
        for w ∈ 𝕎 do
            w' ← FINDWINDOWS(PC, i, w.start, w.end)
            𝕎_new.add(w')
        sort 𝕎_new by variance asc
        𝕎 ← all w ∈ 𝕎_new with the smallest variance
    return 𝕎
```

choose the Timestamp parameter in LDBC 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 2 is extended with the Timestamp parameter in Figure 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 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



(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 ($PCTime$)

Fig. 3: Preprocessing for the query plan with two correlated parameters

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 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 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.

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 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 1.3 and its query plan in Figure 4a.

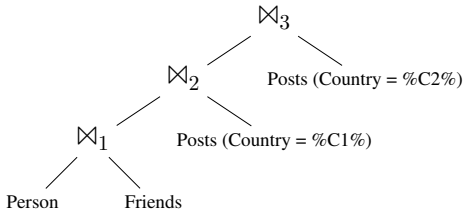
Query 1.3: LDBC Query 3

```
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 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 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 4b.

Now we can essentially split the LDBC Query 3 into three different (related) query variants (a), b) and c)), based on the combination of the two *%Country* parameters: a)



(a) Query Plan

Countries

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

(b) Parameter-Count table

Fig. 4: Case of multiple correlated parameters

%C1 and *%C2* from the *Frequent* group, b) both from *Non-Frequent* group, c) combination of the two above.

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 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.

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 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 5.

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

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

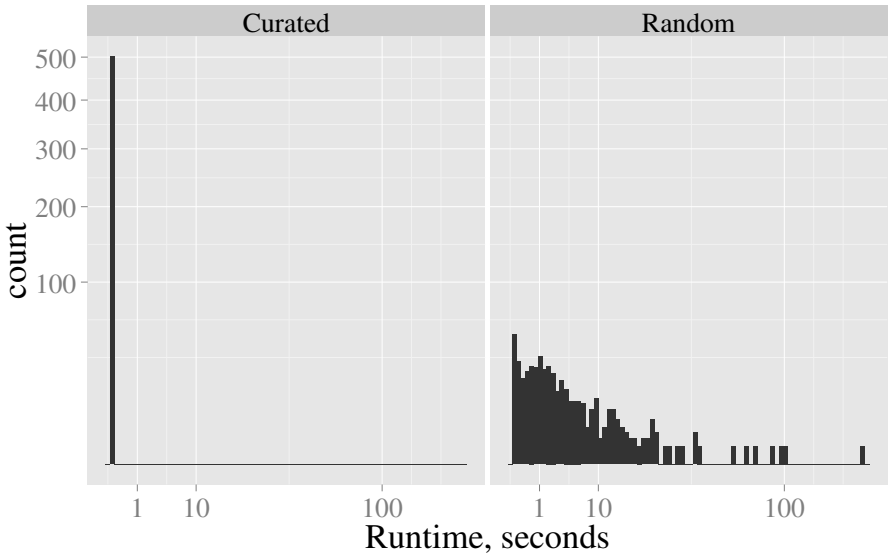


Fig. 5: LDBC Query 4 Runtime Distribution: Curated vs Random parameters

data generator 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 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 6. We see that the three groups indeed have distinct runtime behavior, and the runtime within

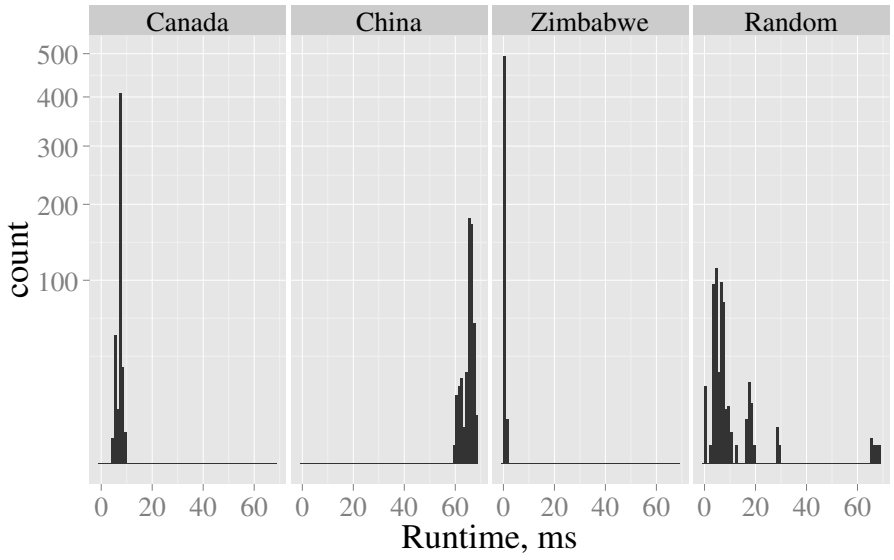


Fig. 6: LDBC Query 11 with four different groups of parameters (for countries China, Canada, Zimbabwe, Random)

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.

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 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.

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

Table 2: Time to extract parameters in the LDBC datasets of different scales

6 Conclusions

In this paper 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) [4], 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. Similar techniques can be used for transactional benchmarks on graph-shaped data (e.g. BG [2]). 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. Our approach to focus the problem on a single *intended* query plan for each template variant reduces the high complexity of generic parameter curation. We experimentally showed that group-by based *data analysis* followed by *greedy parameter extraction* that implements Parameter Curation in the case of LDBC SNB is practically computable and can form the final part of the database generator process.

References

1. LDBC Benchmark. http://ldbc.eu:8090/display/TUC/Interactive+Workload
2. Barahmand, S., Ghandeharizadeh, S.: BG: A benchmark to evaluate interactive social networking actions. In: CIDR (2013)
3. Boncz, P., Neumann, T., Erling, O.: TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In: TPCTC (2013)
4. Minh Duc, P., Boncz, P.A., Erling, O.: S3g2: A Scalable Structure-Correlated Social Graph Generator. In: TPCTC (2012)
5. Moerkotte, G.: Building Query Compilers. http://pi3.informatik.uni-mannheim.de/ moer/-querycompiler.pdf
6. Poess, M., Stephens, Jr., J.M.: Generating thousand benchmark queries in seconds. pp. 1045–1053. VLDB '04
7. Stephens, J.M., Poess, M.: Mudd: a multi-dimensional data generator. SIGSOFT Softw. Eng. Notes 29(1), 104–109 (Jan 2004)

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