# **Simplicity Done Right for Join Ordering**

Axel Hertzschuch, Claudio Hartmann, Dirk Habich, Wolfgang Lehner Database Systems Group Technische Universität Dresden Dresden, Germany firstname.lastname@tu-dresden.de

### ABSTRACT

In this paper, we propose a simple, yet fast and effective approach to determine good join orders for arbitrary selectproject-join queries. Our scheme comprises three building blocks: (i) a simple upper bound for arbitrary multi-joins, (ii) appropriate join enumeration according to the upper bound, and (iii) sampling as query execution to provide fast and near-exact estimates for complex conjunctive filters. As we are going to show, using the Join-Order-Benchmark (JOB), our simple approach provides better join orderings with significantly less optimization overhead, resulting in a substantially faster response time for all 113 JOB queries compared to state-of-the-art and recent approaches.

### 1. INTRODUCTION

Although query optimization has been a core research topic for decades, it is still far from being solved [7]. This holds especially true for one of the most studied challenges in query optimization: *finding a good join order* [2, 7]. To tackle this issue, reliable cardinality estimates for arbitrary joins are required [2, 7]. This includes joins over intermediate join results and pre-filtered base tables. However, the question whether it is even possible to achieve such estimates without join execution is yet to be answered [15].

Traditional estimation techniques frequently rely on basic heuristics that may assume predicate independence and a uniform distribution of attribute values [7]. However, relying on these assumptions can lead to disastrous join orderings [7]. To overcome this issue, various sophisticated techniques for the join cardinality estimation have been proposed in recent years. On the one hand, sampling approaches seem appealing at first glance, for example [8, 11, 21], but they do not scale well to many joins [3, 21]. On the other hand, we see an increasing popularity of modern machine learning techniques [6, 20] as they model complex data characteristics. However, these models do not yet cover all relevant filter predicate types and their training depends on executing a plethora of joins, which may take days or weeks [19]. As recently shown by Cai et al. [2], guaranteed bounds for join cardinalities lead to better and more robust join orderings for arbitrary select-project-join (SPJ) queries. However, their sketch-based approach to generate upper bounds does not scale well for many joins in terms of optimization time, as these sketches have to be populated at run time if filters are present [2]. To comprehensively tackle that shortcoming, we introduce our novel *UES* concept in this paper. The most outstanding feature of our concept is its simplicity which is achieved by three building blocks:

- **U**-Block: Assuming basic attribute statistics and accurate selectivity estimates for filters over base tables, we demonstrate a simple, yet effective Upper bound for an arbitrary number of joins.
- **E**-Block: Appropriately Enumerating joins according to our upper bound effectively prevents overly aggressive (sometimes disastrous) join orderings.
- S-Block: To guarantee accurate selectivity estimates even for complex filters in SPJ-queries required by the Ublock, we treat **S**ampling like query execution.

Using the Join-Order-Benchmark (JOB) [7], we demonstrate that our novel UES concept outperforms built-in stateof-the-art optimizers of Postgres as well as of MonetDB. Moreover, we compare our concept with the sketch-based upper bound approach of Cai et al. [2]. As we are going to show, our UES concept achieves highly competitive join orderings, resulting in similar (or slightly better) execution times for all 113 JOB queries while requiring significantly less planning time due to the simplicity of our concept compared to the sketch-based technique.<sup>†</sup>

## 2. UE-BLOCKS: JOIN ORDERING

This section describes our two fundamental building blocks U and E for a *scalable* join ordering. For that, we assume access to the common statistic of most frequent values (top-k statistics) and precise filter selectivity estimates. While histograms may suffice for basic filters, Section 3 details a more sophisticated approach for complex filters (S-Block).

### 2.1 U-Block: Simple Upper Bound for Joins

The first building block U includes a simple upper bound for an arbitrary number of joins. To describe our upper bound, we start with a single join and discuss arbitrary joins afterwards. Given the (*estimated*) cardinality of two (*prefiltered*) tables, we calculate the smallest number of distinct

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2021. *11th Annual Conference on Innovative Data Systems Research (CIDR '21)* January 10-13, 2021, Chaminade, USA.

<sup>&</sup>lt;sup>†</sup>All materials to reproduce and further analyze the results reported in this work are available at: https://github. com/axhertz/SimplicityDoneRight



Figure 1: Illustration of our upper bound (U-block).

values each table can contain and assume as many joining values as possible. Note, the naïve worst-case assumption expects both tables to share the same single distinct value which leads to a join size equal to the cartesian product. However, we use top-k statistics to narrow down the upper bound. Figure 1 illustrates the core concept, using an example. The left-hand side depicts the worst-case - used to derive the upper bound – constrained by the table statistics, while the right-hand side depicts the actual join. Using table statistics, we denote MF(R.x), MF(S.y), the maximum frequency a value can occur in attribute x, y of table R, S. According to the example, each value of the joining attributes can occur at most five times and two times, respectively. Dividing the table cardinality by the maximum value frequency gives the minimum number of distinct values. Thus, in the worst-case,  $\sigma(R.x)$  contains two and  $\sigma(S.y)$  five distinct values. As both join partners need to have the same value, we take  $\min(2,5) = 2$  and multiply it with the respective frequencies. Although the number of distinct values may differ in reality, the following inequality (upper bound) holds true in any case:

$$\begin{aligned} |\sigma(R) \bowtie \sigma(S)| &\leq \operatorname{upper}(|\sigma(R) \bowtie \sigma(S)|) \\ &= \min\left(\frac{|\sigma(R)|}{\operatorname{MF}(R.x)}, \frac{|\sigma(S)|}{\operatorname{MF}(S.y)}\right) * \underbrace{\operatorname{MF}(R.x) * \operatorname{MF}(S.y)}_{=: \operatorname{MF}(R \bowtie S)} \end{aligned}$$

The right-hand side of Figure 1 gives an intuition of why the inequality holds: While value a occurs with the respective maximum frequency in both tables, value b appears contrary to the assumption on the left-hand side—only two times. Thus, the former bucket of  $\sigma(R.x)$ , containing only value b is split into three buckets (with values b, c, and d). Since the maximum value frequency in  $\sigma(S.y)$  can be at most MF(S.y) = 2, each bucket in  $\sigma(R.x)$  is at most twice the size after the join. Hence, having three smaller buckets of size 2 (value b), size 2 (value c), and size 1 (value d) in  $\sigma(R.x)$ that can expand by a factor of at most two, we can never exceed the upper bound given by a single bucket of size 5 (value b) that doubles its size after the join (cf. Figure 1).

Joining multiple tables. In case of multiple joins, we start by deriving the upper bound for the first two (potentially pre-filtered) tables according to the stated inequality, giving an estimate of  $|\sigma(R) \bowtie \sigma(S)|$ . The upper bound after joining an additional table  $\sigma(T)$  over attribute z, is given by:

$$upper(|(\sigma(R) \bowtie \sigma(S)) \bowtie \sigma(T)|) = \\min(\frac{upper(|\sigma(R) \bowtie \sigma(S)|)}{MF(S.y) * MF(R.x)}, \frac{|\sigma(T)|}{MF(T.z)}) * MF(R \bowtie S \bowtie T),$$

where  $MF(R \bowtie S \bowtie T) = MF(R.x) * MF(S.y) * MF(T.z)$  is the maximum possible value frequency after joining R, S, T. Note that similar to the sketch-based approach of [2], the join order may determine the tightness of our upper bound.

### 2.2 E-Block: Enumeration Scheme

The second building block E for our scalable join ordering is a simple enumeration scheme using our upper bound. We detail our scheme in Figure 2 on an example query of the Join-Order-Benchmark, whereby we transform the implicit where clauses (Figure 2a) into an explicit join order (Figure 2b). The key idea of our enumeration scheme is (i) to push down the non-expanding operators, i.e. filters and primary-foreign-key joins (pk-fk joins) and (ii) to enumerate the (potentially expanding) n:m joins according to our upper bound and to the following greedy heuristic<sup>\*</sup>:

For all n:m join candidates (red part in Figure 2a), **ci**, **mi**, **mi\_idx**, the upper bounds for the pk-fk joins (green part in Figure 2a) with **t**, **it1**, **it2**, **n** are derived. Note, we consider pk-fk joins "special filters" as they may shrink (but never expand) n:m candidate sizes before applying the n:m join. However, we distinguish two cases: Pk-fk joins are either applied directly or within a subquery. While a subquery may reduce the size of an n:m join candidate before applying the n:m join, it employs an *additional pipeline-breaker*, and thus result-tuples of preceding joins may need to wait for the result of the subquery. Therefore, we only employ subqueries when our upper bound guarantees that the preceding pk-fk join reduces the size of the respective n:m join candidate.

According to the steps outlined in Figure 2b, we proceed as follows: (1) We start with the n:m join candidate **mi\_idx**, as it is—according to our upper bound—the smallest candidate after applying all non-expanding operations. (2) We compute the upper bound for the join with the (pre-filtered) candidates **ci** and **mi**. Despite of having no guarantee that the pk-fk join **n.id = ci.person\_id** reduces the size of **ci**, the n:m join **ci.movie\_id = mi\_idx.movie\_id** provides a smaller upper bound compared to **mi.movie\_id = mi\_idx.movie\_id** and is applied next. Note, the pk-fk join **t.id = ci.movie\_id** is not present in the explicit join order since **t.id = mi\_idx.movie\_id** has already been applied. (3) Before applying the last n:m join, **mi** is filtered by a subquery that is—according to our upper bound—guaranteed to shrink the table.

### 2.3 Facing the Join-Order-Benchmark

To show the effectiveness and applicability of our two fundamental building blocks U and E for a *scalable* join ordering, we present an evaluation based on the *Join-Order-Benchmark* (JOB) [7], which comprises 113 SPJ-queries with up to 16 joins over real-world data. We run our experiments on a 64-bit Linux machine with a single-socket Intel Core i7-6700 CPU, 16GiB of main memory and SSD storage. All 113 JOB queries are evaluated after a warm-up phase on the following database instances:

<sup>\*</sup>In favor of focusing on the core concept, a more detailed discussion of the pseudo-code is postponed to Section 4.

| SELECT COUNT(*) FROM  | SELECT COUNT(*) FROM                                      |   |
|---|---|---|
| cast_info AS ci, name AS n, title AS t,                         | movie_info_idx AS mi_idx                                  |   |
| info type AS it1, info type AS it2,                             | JOIN info_type AS it2 ON                                  | choose smallest n:m join                |
| movie_info AS mi, movie_info_idx AS mi_idx                      | (it2.info = 'votes' AND it2.id = mi_idx.info_type_id)     | candidate <b>mi_idx</b> and apply       |
| WHERE   | <b>JOIN</b> title AS t <b>ON</b> (t.id = mi_idx.movie_id) | non-expanding operations                |
| ci.note IN ('(producer)', '(executive producer)') non-expanding | JOIN cast_info AS ci ON (ci.movie_id = mi_idx.movie_id    |   |
| AND it1.info = 'budget' AND it2.info = 'votes'                  | AND ci.note IN ('(producer)','(executive producer)')      | $\rightarrow$ choose next n:m join acc. |
| AND n.gender = 'm' AND n.name LIKE '%Tim%'                      | JOIN name AS n ON   | to <b>upper bound</b> and apply         |
| AND t.id = mi.movie_id AND t.id = mi_idx.movie_id               | (n.gender = 'm' AND n.name LIKE '%Tim%'                   | non-expanding operations                |
| AND t.id = ci.movie_id AND it1.id = mi.info_type_id             | AND $n.id = ci.person_id$ )                               | ) non expanding operations              |
| AND n.id = ci.person_id AND it2.id = mi_idx.info_type_id        | JOIN (SELECT movie_id FROM movie_info AS mi_sub           | 3                                       |
| AND ci.movie_id = mi.movie_id potentially expanding             | JOIN info_type AS it1 ON (it1.info = 'budget'             | ≻ filter n:m join candidate mi          |
| AND ci.movie_id = mi_idx.movie_id                               | AND it1.id = mi_sub.info_type_id))                        | by subquery with pk-fk join             |
| AND mi.movie_id = mi_idx.movie_id;                              | AS mi <b>ON</b> (mi.movie_id = ci.movie_id);              | J                                       |
|   | //  |   |

(a) Implicit Syntax – Joins yet to be ordered

(b) Explicit Join Order - Physical operators yet to be determined

Figure 2: Rewriting of JOB query 18a according to our UE approach. Non-expanding operators (pk-fk joins, filters) are highlighted green and potentially expanding operators (n:m joins) are highlighted red.

- 1. Postgres v. 12.4: The most recent release of the opensource disk-centric row store [16]. We use the standard statistics gathered by the autovacuum daemon.
- 2. Postgres v. 9.6: An instance, modified by Cai et al. [2] and publicly available at [10].
- 3. <u>MonetDB v. 11.37.11</u>: The most recent release of the open-source in-memory column store [13].

For our experiments, we transform all JOB queries into our explicit join syntax (cf. Figure 2) representing our determined join ordering. For that, we rely on Postgres' default histograms and table statistics to derive upper bounds for joins, i.e. we do not inject external knowledge (e.g. true filter selectivities). To generate the explicit query string, we use a python script that parses the implicit query, handles the statistic requests and applies our enumeration scheme. This adds a planning time overhead of around 7ms per query, which we include in all reported results. However, note that the full integration of our *UES* concept into the database system would virtually remove this overhead.

Postgres v. 12.4: In our first experiment, we use Postgres v. 12.4. to compare the implicit queries to our explicit join order queries. To bypass reordering of our explicit joins (cf. Figure 2b), we use "set join\_collapse\_limit = 1" and use the default value for the implicit join order (cf. Figure 2a). As our external enumeration scheme is agnostic to Postgres' fine tuned cost model, we initially limit our queries to hash joins (using yet another SQL hint). The implicit JOB queries are evaluated with (hash) and without (plain) restricting them to hash joins. Table 1 comprises the query response times (which includes optimization) achieved by our UE-blocks in comparison to the default Postgres. We distinguish two index configurations for primary (pk) and foreign key (fk) indices. The effectiveness of our (rather passive) approach is most evident with regard to the longest running queries where we achieve speedups of an order of magnitude. Despite of forcing the same physical join operator, we demonstrate a considerably smaller cumulative and maximum query response time compared to hash, thus confirming more efficient join orderings. In particular, we achieve faster query response times for 62% of the workload. That is, Postgres' aggressive query optimization actually finds a better join ordering in many cases. However, in

|        |                 | Post  | gres v. | 12.4  | Mone  | tDB  |
|--------|-----------------|-------|---------|-------|-------|------|
|        | index           | plain | hash    | UE    | plain | UE   |
| $\sum$ | pk              | 464.4 | 472.5   | 258.7 | 90.5  | 29.8 |
| max    |                 | 57.4  | 78.9    | 5.1   | 9.9   | 0.9  |
| $\sum$ | pk+fk           | 315.9 | 387.7   | 258.1 | 153.3 | 34.2 |
| max    | $P_{V} \perp W$ | 47.0  | 79.9    | 6.2   | 45.5  | 2.1  |

Table 1: Query response time in [s], using different index configurations. Sum  $(\sum)$  comprises the cumulative and max the maximum individual response time.

these cases our response times are at most 1.5 seconds behind those of Postgres, while Postgres' join orderings require up to 74 seconds more time to execute.

MonetDB v. 11.37.11: In our second experiment, we use MonetDB to compare the implicit queries with our explicit join order queries. Again, our UE approach considerably reduces query response times in all considered scenarios as depicted in Table 1. We achieve faster query response times for 80% of the workload. In this experiment, our response times are at most 0.3 seconds behind those of MonetDB, while MonetDB's plans require up to 9.1 seconds more time to run the query. Thus, both experiments confirm that our UE approach produces more efficient join orderings than query optimizers in state-of-the-art database systems.

Postgres v. 9.6: The most recent work on SPJ-query optimization proposes sketches to estimate upper bounds [2]. This approach has been integrated in a Postgres v. 9.6 instance which is publicly available [10]. Thus, we also compare our UE approach with this sketch-based technique and show the results in Figure 3, using only primary-key indices. From a query execution plan perspective, both approaches produce similar plans over all different number of joins resulting in similar query execution times. In some cases, we still achieve slightly better plans. For example, analyzing the slowest sketch-based query (16b) reveals: While the sketch-based plan directly applies the n:m join candidate ci on **mk**, our UE approach forces all pk-fk joins on **mk** prior to the n:m joins, reducing the respective intermediate results. However, both approaches greatly differ in the time spent to determine the join order. If filters are present, the sketches must be populated at runtime [2]. In short: sketches are compact representations of (potentially prefiltered) tables,



Figure 3: Comparison of average planning and execution time grouped by the number of joins, using a Postgres (v. 9.6) instance, modified by Cai et al. [2]. The black band marks the standard deviation of the execution time.

partitioned according to some hash function. To derive join cardinality estimates, Cai et al. [2] combine (join) the eligible hash partitions of each join partner. Although, these estimates might be sophisticated, this scheme entails two major drawbacks: First, populating the sketches requires a full scan on each join attribute. Second, if query graphs become more complex, the number of possible join paths and therefore the number of sketch combinations grows rapidly. In particular, as Figure 3 reveals, the planning time of the sketch-based approach greatly exceeds the execution time with an increasing number of joins. In contrast, our much simpler UE approach entails virtually no planning time for the join ordering, and thus, is barely visible in Figure 3.

**Reoptimization.** As our scheme is agnostic to the cost model, it heavily relies on hash joins and therefore does not benefit from fk-indexes and potentially more efficient indexbased joins. As an alternative, one could cache our plans and reiterate the choice of physical operators when the query is issued a second time and intermediate result sizes are known. To indicate the potential of more fine grained physical operator selections, we run all JOB queries on Postgres v. 12.4 with and without the restriction to hash joins (while keeping the join order). Running the best of each query reduces the cumulative response time from 258s to 168s. Note that individual decisions of physical join operators cannot be controlled with SQL hints and thus require the full integration of our scheme into the respective optimizer.

## 3. S-BLOCK: SELECTIVITY ESTIMATES

As shown above, our simple, but effective building blocks U and E provide better join orderings, resulting in faster query execution times for all 113 JOB queries with significantly less overhead for the optimization compared to state-of-the-art and recent approaches. However, that only applies because Postgres' default histograms are sufficient to estimate filter selectivities in case of the Join-Order-Benchmark. Without accurate filter selectivity estimates, such as for complex filter predicates [5], our upper bound may not hold and can result in a sub-optimal join ordering.

As an illustration, we run a small experiment by artificially adding multiple *like* expressions of the form mi.info like '%e%' as conjunctive filter predicate to JOB query 19d. This leads to a vast underestimate of the filter selectivity based on histograms, which invalidates our upper bound and drives our enumerator towards a sub-optimal plan. While the former join order is executed in 5s in Postgres, this single mis-estimate causes a join order that takes 25s.

Thus, to guarantee accurate selectivity estimates for complex filter expressions in SPJ-queries, we propose *treating sampling like query execution* as our third building block *S*. We emphasize that idea to exploit the potential of common database objects such as indices, zone-maps and histograms. While recent work of Birler et al. [1] makes sampling feasible on disk-based database systems, we introduce two sampling approaches tailored to modern in-memory column-stores:

**Focused Sampling:** a new approach to accelerate sampling by exploiting the column-store format.

## **Conditional Sampling:** employs any index structure to increase estimation accuracy.

Inspired by previous work, e.g. [11, 14], our sampling experiments are based on the forest data set [9]. In line with the related work, we execute 10,000 random range queries with r predicates according to the following pattern:

- SELECT \* FROM forest\_data\_set WHERE
- attribute  $A_1$  BETWEEN value  $v_1^{\perp}$  AND value  $v_1^{\top}$  AND . . .
- attribute  $A_r$  BETWEEN value  $v_r^{\perp}$  AND value  $v_r^{\top}$

After randomly choosing subsets of attributes, range predicates are generated from two uniformly and randomly selected values from the attribute's domain. To avoid any interference with the q-error-metric [12] throughout the evaluation, we only generate queries with non-empty result sets.

### 3.1 Focused Sampling

Although sampling might be reasonably fast for in-memory systems due to efficient random access, ad-hoc sampling entails a considerably stronger overhead in comparison to histograms. In the following, we describe an *online* sampling approach that provides fast estimates from *fresh* data.

Instead of materializing tuples—referenced by sampled tuple identifiers (TIDs)—and evaluating the filter on the materialized sample afterwards, we directly evaluate the filter predicates over the respective base columns of the tuples. The advantages are two-fold: First, there is no need to copy or update tuples separate from the base table. Second, we potentially skip page accesses for conjunctive filters: Given a conjunctive filter of four predicates, we skip the evaluation of three predicates if the respective attribute of the random tuple does not qualify the first predicate. This is especially appealing for in-memory column stores where different attributes of one tuple are stored across different pages. Thus, the number of random page accesses can be effectively reduced during the sampling process. The example given in Figure 4 summarizes our core idea and demonstrates that

| TID  | А | В | С | D |  |
|------|---|---|---|---|--|
| 5118 | 1 | 2 | 7 | 4 | Query:   |
| 252  | 5 | 2 | 3 | 7 | $\mathbf{A} = 1 \wedge \mathbf{B} = 2 \wedge \mathbf{C} = 3 \wedge \mathbf{D} = 4$ |
| 3934 | 1 | 4 | 3 | 1 |  |
| 16   | 1 | 2 | 3 | 4 | Random accesses: 12/24   |
| 129  | 4 | 5 | 2 | 4 |  |
| 7586 | 2 | 1 | 2 | 5 | Estimated selectivity: 1/6   |

Figure 4: Pages of base table referenced by random tuple identifiers (TIDs). Grey parts are not accessed.

we skip half of the random accesses due to non-qualifying values (depicted in red font) of the first attributes. Similar to traditional sampling, we divide the number of qualifying tuples by the total number of sample tuples, giving a selectivity estimate of 1/6 for the conjunctive filter. To bypass the overhead of generating random numbers (TIDs), a vector of n random numbers is generated only once. Accordingly, each query evaluated over a sample of size k uses the same first k TIDs of the vector. Note, the random vector requires just a fraction of the memory consumed by an equally sized materialized sample with all attributes (1/55 in case of forest data) and only needs to be updated if the base table cardinality and therefore the sample space changes.

**Evaluation:** For different sample sizes from 1,000 to 11,000 tuples and filters with r conjunctive predicates,  $r \in \{3, 5, 7\}$ , we measure the cumulative estimation time for 10,000 random range queries according to the following approaches:

- 1. <u>traditional</u>: Generate random TIDs and copy attribute values of r columns, referenced by the TIDs. Evaluate r filter predicates over the materialized sample.
- 2. <u>trad. fixed TIDs</u>: Same as (1), but instead, random TIDs are generated only once for each query.
- 3. <u>focused w/o enumeration</u>: Filter predicates are evaluated over table tuples, referenced by (fixed) random TIDs, while skipping unnecessary accesses (cf. Fig. 4).
- 4. <u>focused with enumeration:</u> Same as (3), but the filter predicates are sorted in ascending order according to their single selectivity, e.g. using histograms, to skip random accesses as early as possible.

In this evaluation, we use a custom storage engine implementation that imitates a modern in-memory column store. Again, we run all experiments on a 64-bit Linux machine with an Intel i7-6700 CPU and share the implementation [4]. As can be seen in Figure 5, reusing random TIDs drastically reduces the required estimation time. Besides circumventing the generation overhead, fixed random TIDs increase the probability of accessing cached values. Further, evaluating the filter predicates directly over the base table and skipping accesses by *focusing* on references that still may contribute qualifying tuples, consistently demonstrates fast estimates. Prioritizing selective predicates achieves a speedup of 65% on our focused approach. Irrespective of the number of predicates, starting with the most selective predicates results in an average estimation latency of  $7\mu$ s using  $10^3$  random TIDs and around  $100\mu$ s for  $10^4$  random TIDs. The gap between the reported methods continues to widen as filters become more complex -a likely scenario in real world applications [18]. Transferred to our UE building blocks with the assumption of a maximum sample size of 10<sup>4</sup> tuples, our focused sampling would entail an additional planning time overhead of < 1s for all 113 JOB queries in total while the sketch-based approach of Cai et al. [2] takes > 1000s.

### **3.2 Conditional Sampling**

To improve estimation accuracy, we integrate common index (or index-like) structures. Thus, our *conditional sampling* approach can be seen as a conceptual extension of index-based join sampling [8] to filter selectivity estimation.

Let  $q := A = 1 \land B = 2 \land C = 3 \land D = 4$  be a conjunctive filter over the attributes A, B, C, D. Assuming an available index



Figure 5: Cumulative sampling time for 10,000 queries.

for attribute A, we proceed as follows: Using the index, we sample TIDs of tuples that qualify the respective predicate A = 1, uniformly at random. Similar to our *focused sampling* approach, we access and count the qualifying tuples for the subexpression  $B=2 \wedge C=3 \wedge D=4$ . This gives the conditional probability, that is the fraction of tuples qualifying  $B=2 \wedge C=3 \wedge D=4$  under the condition A=1. In other words, we apply sampling to estimate the conditional probability p(B=2, C=3, D=4|A=1). Since we are interested in the joint probability, we apply Bayes' rule:

$$p(A=1,...,D=4) = p(B=2,C=3,D=4|A=1)p(A=1),$$

where p(A=1) is given by a traditional histogram or by the index itself. Figure 6 summarizes our core concept. Here, we revisit our example from Figure 4 and apply our conditional sampling approach. Due to the predicate selectivity p(A=1) = 0.5, we only need to consider half of the random TIDs. Since we already know that the residual TIDs qualify A = 1, there is no need to access the respective attribute. If we use the same number of random accesses as before, this approach virtually increases the sample size by a factor of  $p(A = 1)^{-1}$  and therefore improves estimation quality. Moreover, as we potentially skip a highly selective predicate, we may circumvent the worst-case for sampling where no sample tuple qualifies the conjunctive filter.

**Evaluation:** We consider a fixed budget of sampled tuples and analyze the effect of our *conditional sampling* approach on estimation accuracy. Besides the traditional approach of directly sampling for the joint probability, we analyze:

- 1. <u>conditional random pred.</u>: Using an index, we sample <u>conditional TIDs (cf. Fig. 6)</u> for a randomly selected predicate of the conjunctive filter.
- 2. <u>conditional most selective</u>: An index is used to sample conditional TIDs for the most selective predicate.

To measure estimation accuracy, we use the multiplicative error of the actual and estimated selectivity, known as



Figure 6: Conditional Sampling.



Figure 7: Estimation accuracy for 10,000 queries.

q-error metric [12]. In the difficult case of an empty sample (i.e. no sample tuple satisfies the filter expression), instead of returning 0, we use the geometric mean of the probabilistic lower and upper bound for qualifying table tuples as described in [11]. Note, the conditional sample is still a random sample. Therefore, the probabilistic bounds also apply to the conditional selectivity p(B=2, C=3, D=4|A=1).

Figure 7 comprises the q-error on a logarithmic scale for conjunctive queries with r = 5 predicates and three different sample sizes. The whiskers of each box span the 5% to 95% quantile. That is, we focus on the general performance rather than the outliers. The relative performance of the outliers is dictated by the handling of empty samples which occur far less frequently when using conditional sampling. In particular, providing a conditional sample for the most selective predicate yields the smallest maximum q-error in all considered scenarios and reduces the relative number of empty samples from 10% to < 1% when using 10k sample tuples. While using an index for the targeted attribute of a randomly selected predicate significantly improves over traditional sampling, we see a tremendous accuracy boost when conditional TIDs are sampled for the most selective predicate. In fact, conditional sampling with 1k tuples achieves a median q-error of 1.15 and outperforms traditional sampling using 10k tuples. Interestingly, if the selectivity of the predicate over the indexed attribute falls below the sample fraction, the conditional sample explores all qualifying tuples and thus gives an exact estimate.

**Discussion.** If no full index is provided, we may use a zonemap [17] on the most selective predicate and sample TIDs from valid zones. Then, we proceed analogously to the full index case, except that we may access tuples where the predicate does not qualify. In line with our focused approach, we reduce the number of random accesses by intersecting all valid zones for each predicate. If a random TID lies within a zone that is invalid for at least one predicate, we count the tuple as non-qualifying without actually accessing it.

### 4. UES: THE ALGORITHM

This section provides a concluding walk-through of our building blocks with regard to Algorithm 1. Starting with the problem definition: The algorithm takes as input a set of relations and outputs a tree that dictates the join order. Here, we expect that there is at least one join-path that connects all tables without using cross-products. If crossproducts are necessary, we apply them as late as possible in the join tree. **S-Block:** As we always push-down filter operators, our scheme relies on precise estimates for the number of qualifying tuples from the base table. To achieve these cardinality estimates fast and independent of the filter type,

#### Algorithm 1: UES

**Input:** a set of relations to be joined R**Output:** join tree T

- T = empty join tree
- $|\cdot|^{\sigma} \sim \text{cardinality estimate for filtered base table}$
- $n_m_tab = tables of R$  that are part of any n:m-join
- 4  $pk_{fk_{tab}} = R \setminus n_m_{tab} //$  set minus
- $_{5}$  MF = dictionary that maps (table, attribute) pairs to maximum frequency moments
- 6 initialize MF according to top-k statistics
- 7 upper = dictionary mapping n:m candidate to bound // enumerate n:m joins

| 9         | for $R_i \in n\_m\_tab$ do   |
|-----------|--|
| 10        | $\mathtt{upper}(R_i) =$  |
|           | $\min\{\min( R_i ^{\sigma}, \ \mathtt{MF}(R_i, fk\_attr) *  pk\_rel ^{\sigma})$  |
|           | $\mid S_{j} \in \texttt{joinPartners}(R_{i}, pk\_fk\_tab)$   |
|           | $\land (fk\_attr, pk\_attr) \in \texttt{joinAttr}(R_i, S_j) \}$  |
| 11        | if $T$ is empty then   |
| 12        | $T = \underset{R_i \in n\_m\_tab}{\operatorname{argmin}} \operatorname{upper}(R_i)$  |
| 13        | continue   |
| 14        | $best\_upper = \infty$   |
| 15        | for $R_i \in \texttt{joinPartners}(T, n\_m\_tab)$ do   |
|           | <pre>// get minimal possible upper bound if</pre>  |
|           | multiple join attributes are present   |
| 16        | $cur\_upper =$   |
|           | $\min \Bigl\{ \min \Bigl( \tfrac{\mathtt{upper}(T)}{\mathtt{MF}(T,a_1)}, \tfrac{\mathtt{upper}(R_i)}{\mathtt{MF}(R_i,a_2)} \Bigr) * \mathtt{MF}(T,a_1) * $ |
|           | $	extsf{MF}(R_i,a_2) \mid (a_1,a_2) \in 	extsf{joinAttr}(T,R_i) \}$  |
|           | <pre>// greedy selection of next n:m join</pre>  |
| 17        | if $cur\_upper < best\_upper$ then   |
| 18        | $best\_upper = cur\_upper$   |
| 19        | $next\_n\_m = R_i$   |
|           | <pre>// update partial join tree</pre>   |
| <b>20</b> | if upper(next_n_m) < $ next_n_m ^{\sigma}$ then  |
|           | /* apply pk-fk joins first */  |
| <b>21</b> | $T = (T \bowtie (((next_n \bowtie \bowtie S_i^1) \bowtie S_i^2)S_i^n)),$   |
|           | $S_i^j \in \texttt{joinPartners}(next\_n\_m, pk\_fk\_tab)$   |
|           | such that: $ S_i^j ^{\sigma} \leq  S_i^{j+1} ^{\sigma} \wedge S_i^j \neq S_i^k, \forall j \neq k$  |
| 22        | else   |
|           | <pre>/* apply n:m join first */</pre>  |
| 23        | $T = (\dots(((T \bowtie next\_n\_m) \bowtie S_i^1) \bowtie S_i^2) \dots S_i^n),$   |
|           | $S_i^j \in \texttt{joinPartners}(next\_n\_m, pk\_fk\_tab)$   |
|           | such that: $ S_i^j ^{\sigma} \leq  S_i^{j+1} ^{\sigma} \wedge S_i^j \neq S_i^k, \forall j \neq k$  |
|           | // register new bound and update statistics  |
| 24        | $upper(T) = best\_upper$   |
| <b>25</b> | for $(a_1, a_2) \in \text{joinAttr}(T, next_n_m)$ do   |
| 26        | $MF(T, a_1) = MF(T, a_1) * MF(next\_n\_m, a_2)$  |
| 27        | $pk\_fk\_tab = pk\_fk\_tab \setminus pk\_fk\_jp$   |
| 28        | $n\_m\_tab = n\_m\_tab \setminus next\_n\_m$   |
| 29        | return T   |
| -         |  |

we provide our custom sampling approach (cf. Section 3) in Line 2. Besides pushing down the *regular* filter operators, we always apply pk-fk joins prior to n:m joins. Remember that pk-fk joins cannot expand the n:m candidates, and thus are treated as *special* filters. **U-Block:** In Line 9-10, we assess the maximum size of the n:m candidates after applying each non-expanding operator. Therefore, we combine the bound formula for pk-fk joins with the sample-based cardinality estimates of pre-filtered base tables. Although a combination of pk-fk joins may reduce the join cardinality beyond a single pk-fk join, there is always one pk-fk join that drives the guaranteed upper bound towards its minimum. Since pk-fk joins are a special case, we can transform the bound formula and minimize the following in Line 10:

 $upper(\sigma(R_i) \bowtie \sigma(S_j))$ 

$$= \min\left(\frac{|\sigma(R_i)|}{\mathrm{MF}(R_i)}, \frac{|\sigma(S_j)|}{\mathrm{MF}(S_j)}\right) * \mathrm{MF}(R_i) * \mathrm{MF}(S_j)$$
$$= \min\left(\frac{|\sigma(R_i)|}{\mathrm{MF}(R_i)}, \frac{|\sigma(S_j)|}{1}\right) * \mathrm{MF}(R_i) * 1$$
$$= \min\left(|\sigma(R_i)|, |\sigma(S_j)| * \mathrm{MF}(R_i)\right)$$

where  $S_i$  is a pk-fk join partner of  $R_i$ . **E-Block:** We initialize the join tree with the n:m candidate that provides the smallest estimated cardinality after applying all non-expanding operators in Line 11-12. In the consecutive iterations, we order the n:m joins according to the greedy policy whereby Line 16 determines the minimal upper bound of the next intermediate result by considering all pairs of join attributes. After selecting the next n:m join, we decide the position of the respective pk-fk joins in Line 20-23. If we are guaranteed to reduce the size of the n:m candidate, we apply the pk-fk joins first, thus adding a new branch to the tree. Otherwise we apply the pk-fk joins after the n:m join which gives a linear sub-tree. Lastly, in Line 24-28, we register the upper bound of the intermediate join tree, update the frequency statistics and remove already considered tables from the remaining join candidates.

## 5. CONCLUSION

This paper presents a simple, yet effective concept to determine good join orders for arbitrary SPJ-queries. Most importantly, our concept is founded on common table statistics and accurate selectivity estimates. Based on that, we derive a simple upper bound for join cardinalities and enumerate the join candidates accordingly. To guarantee accurate selectivity estimates, we use a customized sampling approach that exploits specific access patterns and index structures. Using the popular Join-Order-Benchmark, we demonstrate faster query response times compared to the state-of-the-art built-in optimizers of Postgres and MonetDB. In comparison to the most recent sketch-based upper bound approach [2], we achieve highly competitive execution times while cutting away the unacceptable planning time overhead. Although traditional histograms seem sufficient to estimate decent filter selectivities for the JOB queries, we observe a strong accuracy degradation when adding more complex filter predicates to the JOB queries. Therefore, we envision the integration of our combined UES-concept into modern in-memory column-oriented DBMS where complex filter expressions are no exception in real-world OLAP scenarios.

### Acknowledgments

This work was funded by the German Research Foundation (DFG) within the RTG 1907 (RoSI). We would like to thank Guido Moerkotte for sharing the basic implementation of the columnar storage engine used in our sampling experiments.

## 6. REFERENCES

- [1] A. Birler, B. Radke, and T. Neumann. Concurrent online sampling for all, for free. *DaMoN*, 2020.
- [2] W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *SIGMOD*, pages 18–35, 2019.
- [3] Y. Chen and K. Yi. Two-level sampling for join size estimation. In SIGMOD, pages 759–774, 2017.
- [4] A. Hertzschuch et al. https://github.com/axhertz/SimplicityDoneRight. Accessed: 2020-10-19.
- [5] F. Kastrati and G. Moerkotte. Optimization of conjunctive predicates for main memory column stores. *PVLDB*, 9(12):1125–1136, 2016.
- [6] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.
- [7] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.
- [8] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.
- [9] M. Lichman et al. UCI machine learning repository. https://archive.ics.uci.edu/ml/index.php, 2013.
- [10] W. Cai et al. https://github.com/waltercai/pqo-opensource. Accessed: 2020-08-07.
- [11] G. Moerkotte and A. Hertzschuch. alpha to omega: the g(r)eek alphabet of sampling. In *CIDR*, 2020.
- [12] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.
- [13] MonetDB Team. https://www.monetdb.org/. Accessed: 2020-08-07.
- [14] M. Müller, G. Moerkotte, and O. Kolb. Improved selectivity estimation by combining knowledge from sampling and synopses. *PVLDB*, 11, 2018.
- [15] M. Perron, Z. Shang, T. Kraska, and M. Stonebraker. How I learned to stop worrying and love re-optimization. In *ICDE*, pages 1758–1761, 2019.
- [16] Postgres Team. https://www.postgresql.org/. Accessed: 2020-07-22.
- [17] L. Sidirourgos and M. L. Kersten. Column imprints: a secondary index structure. In *SIGMOD*, pages 893–904, 2013.
- [18] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. *DBTest*, 2018.
- [19] L. Woltmann, C. Hartmann, D. Habich, and W. Lehner. Machine learning-based cardinality estimation in dbms on pre-aggregated data. *preprint* arXiv:2005.09367, 2020.
- [20] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In *aiDM*, pages 5:1–5:8, 2019.
- [21] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539, 2018.