COMPASS: Online Sketch-based Query Optimization for In-Memory Databases

Yesdaulet Izenov, Asoke Datta, Florin Rusu, Jun Hyung Shin
University of California Merced
{yizenov,adatta2,frusu,jshin33}@ucmerced.edu

ABSTRACT

Cost-based query optimization remains a critical task in relational databases even after decades of research and industrial development. Query optimizers rely on a large range of statistical synopses for accurate cardinality estimation. As the complexity of selections and the number of join predicates increase, two problems arise. First, statistics cannot be incrementally composed to effectively estimate the cost of the sub-plans generated in plan enumeration. Second, small errors are propagated exponentially through joins, which can lead to severely sub-optimal plans. In this paper, we introduce COMPASS, a novel query optimization paradigm for in-memory databases based on a single type of statistics—Fast-AGMS sketches. In COMPASS, query optimization and execution are intertwined. Selection predicates and sketch updates are pushed-down and evaluated online during query optimization. This allows Fast-AGMS sketches to be computed only over the relevant tuples—which enhances cardinality estimation accuracy. Plan enumeration is performed over the query join graph by incrementally composing attribute-level sketches—not by building a separate sketch for every sub-plan. We prototype COMPASS in MapD – an open-source parallel database – and perform extensive experiments over the complete JOB benchmark. The results prove that COMPASS generates better execution plans – both in terms of cardinality and runtime – compared to four other database systems. Overall, COMPASS achieves a speedup ranging from 1.35X to 11.28X in cumulative query execution time over the considered competitors.

CCS CONCEPTS

• **Information systems** → **Query optimization**; **Query planning**; **Main memory engines**.

KEYWORDS

join cardinality estimation; sketches; permutation distance

ACM Reference Format:
Yesdaulet Izenov, Asoke Datta, Florin Rusu, Jun Hyung Shin. 2021. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452840

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3452840

1 INTRODUCTION

Consider query 6a from the JOB benchmark [25]:

```
SELECT MIN(k.keyword), MIN(n.name), MIN(t.title)
FROM cast_info AS ci, keyword AS k,
     movie_keyword AS mk, name AS n, title AS t
WHERE
  ▹ selection predicates
    k.keyword = 'marvel-cinematic-universe' AND
    n.name LIKE '%Downey%Robert%' AND
    t.production_year > 2010 AND
  ▹ join predicates
    k.id = mk.keyword_id AND
    t.id = mk.movie_id AND
    t.id = ci.movie_id AND
    ci.movie_id = mk.movie_id AND
    n.id = ci.person_id
```

The query has 3 selection predicates – point, subset, and range – and joins 5 tables with 5 join predicates—there is a triangle subquery between tables *t*, *mk*, and *ci*. The corresponding join graph is depicted in Figure 1. For each join, the graph contains a named edge $e1$–$e5$ that connects the tables involved in the join predicate. For example, edge $e1$ represents the join predicate *k.id = mk.keyword_id*. Figure 1 also includes the execution plans together with their cost – the total cardinality of the intermediate results – for COMPASS and the four other databases considered in the paper. Although all the plans are left-deep trees, their cost ranges from 1,249 to 215 millions tuples. This is entirely due to the statistics used for cardinality estimation.

MapD [53] does not use any statistics, thus its cost is orders of magnitude higher. The plan is determined by sorting the tables in decreasing order of their size—number of tuples. MonetDB [54] has a rule-based optimizer with minimum support for statistics [13], which generates a better plan. The reason why both of these systems have primitive optimizers is because they are relatively "young" and are targeted at modern architectures. They try to compensate bad plans with highly-optimized execution engines that make use of extensive in-memory processing supported by massive multithread parallelism and vectorized instructions. However, this approach is clearly limited. PostgreSQL [55] and the industrial-grade DBMS A are "mature" databases with advanced query optimizers. In order to find the much better plan, they use a large variety of statistics. Histograms, most frequent values, and number of distincts are used to estimate the selectivity of the point predicate on attribute *k.keyword* and of the range predicate on *t.production_year*. The subset LIKE predicate on *n.name* is estimated with table-level samples. Estimating join cardinality requires correlated statistics on the join attributes. While such statistics exist, e.g., correlated

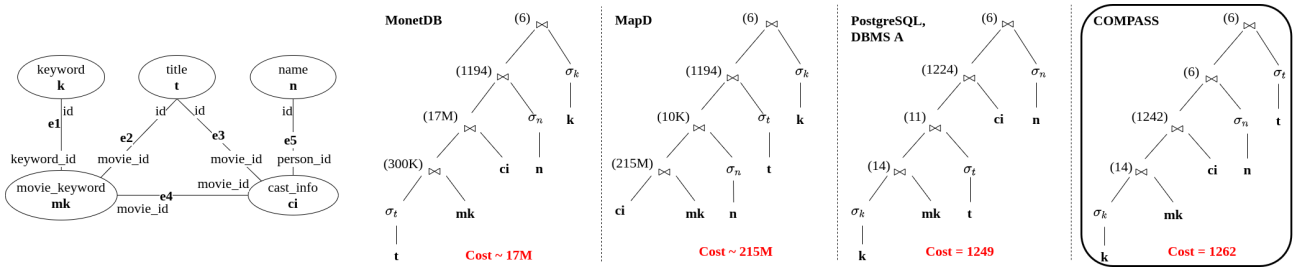


Figure 1: Join graph and corresponding execution plans for query JOB 6a. The numbers represent cardinality.

samples [17, 24, 44], they require the existence of indexes on every join attribute combination, which severely limits their applicability in the case of multi-way joins. As a result, even advanced optimizers rely on crude formulas that assume uniformity, inclusion, and independence—which are likely to produce highly sub-optimal execution plans [23]. Since implementing and maintaining these many statistics requires considerable effort, it is completely understandable that only mature systems implement them.

Problem. We investigate how to design a lightweight – yet effective – query optimizer for modern in-memory databases. We have two design principles. First, we aim to capitalize on the highly-parallel execution engine in the query optimization process. Since query execution is already fast, it is challenging to minimize the overhead incurred by the additional optimization. Second, the type and number of synopses included in the optimizer has to be minimal. Our goal is to employ a single type of synopsis built for single-attributes only. The challenge is to design a composable – and consistent – synopsis that provides incremental cardinality estimates for the sub-plans generated in plan enumeration.

COMPASS query optimizer. We introduce the online sketch-based COMPASS query optimizer. Fast-AGMS sketches [5] are the only statistics present in COMPASS. These sketches are a type of correlated synopses for join cardinality estimation [36, 37] that use small space, can be computed efficiently in a single scan over the data, are linearly composable, and – more importantly – have statistically high accuracy. These properties allow for Fast-AGMS sketches to be computed online in COMPASS by leveraging the optimized parallel execution engine in modern databases. This is realized by decomposing query processing into two stages performed before and after the optimization. In the first stage, selection predicates are pushed-down and Fast-AGMS sketches are built concurrently only over the relevant tuples. Sketches are built for each two-way join independently—not for every combination of tables. In the query optimization stage, plan enumeration is performed over the join graph by incrementally composing the corresponding two-way join sketches in order to estimate the cardinality of multi-way joins. The optimal join ordering is finally passed to the execution engine to finalize the query. As shown in Figure 1, COMPASS identifies a plan as good as PostgreSQL and DBMS A, while relying exclusively on sketches. In addition to the novel query optimization paradigm, we make the following technical contributions:

- We present a systematic approach of using sketches for join cardinality estimation in a query optimizer. We do this for two types of sketches—AGMS [1] and Fast-AGMS [5].
- We introduce two novel strategies to extend Fast-AGMS sketches to multi-way join cardinality estimation. The first strategy – sketch partitioning – is a theoretically sound estimator for a given multi-way join. Since it does not support composition, sketch partitioning is not scalable for join order enumeration. The second strategy – sketch merging – addresses scalability by incrementally creating multi-way sketches from two-way sketches. Although this is done heuristically for a certain multi-way join taken separately, all the multi-way joins with a given size are equally impacted. This property guarantees consistency in plan enumeration.
- We prototype COMPASS in MapD and perform extensive experiments over the complete 113 queries in JOB benchmark. The results prove the reduced overhead COMPASS incurs – below 500 milliseconds – while generating similar or better execution plans compared to the four databases systems included in Figure 1. COMPASS outperforms the other databases both in terms of the number of queries it obtains the best result on, as well as on the cumulative workload runtime. This is the case when the optimal plans are performed in PostgreSQL, DBMS A, and MapD.

2 PRELIMINARIES

Cost-based query optimization. The query optimization problem [4, 23, 25, 48] consists in finding the best execution plan – which typically corresponds to the one with the fastest execution time – for a given query. The search space is defined over all the valid plans – combinations of relational algebra operators – which can answer the query correctly. The number of potential plans is exponentially factorial in the number of tables. Thus, inspecting all of them is not practical for a large number of tables. *Plan enumeration* is the procedure that defines the plans in the search space. Since the execution time of a plan cannot be determined without running it – which defeats the purpose – alternative cost functions are defined. The most common *cost function* is the total size – or cardinality – of the intermediate results produced by all the operators in the plan. This function captures the correlation between the amount of accessed data and execution time. Computing the cardinality of a relational algebra operator is itself a difficult problem and requires knowledge about the processed data. This knowledge is captured by incomplete statistics—or *synopses*. Different classes of statistics [6] are useful for different relational operators. For example, attribute histograms and number of distinct values are optimal for selection predicates, while correlated samples are better for join predicates. With statistics, the cardinality can only be estimated. While accurate for

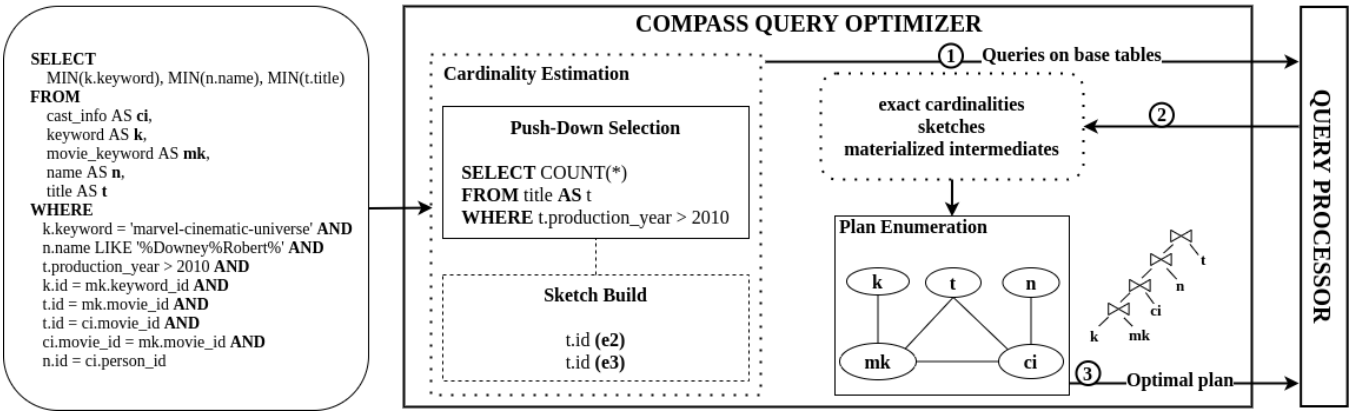


Figure 2: COMPASS workflow: online sketch-based query optimization for in-memory databases.

simple predicates over a small number of attributes, *cardinality estimation* becomes harder for correlated predicates and multi-way joins. This is not necessarily a problem if all the plans are equally impacted. However, estimation errors vary widely across sub-plans, which can lead to highly suboptimal plans. The COMPASS query optimizer includes solutions both for effective plan enumeration, as well as incremental cardinality estimation.

Sketches. Sketch synopses [6] summarize the tuples of a relation as a set of random values. This is accomplished by projecting the domain of the relation on a significantly smaller domain using random functions or seeds. In the case of joins, correlation between attributes is maintained by using the same random function. While sketches compute only approximate results with probabilistic guarantees, they satisfy several major requirements of a query optimizer for in-memory databases—single-pass computation, small space, fast update and query time, and linearity:

- A sketch is built by streaming over the input data and considers each tuple at most once.
- A basic sketch is composed of a single counter and one or more random seeds—a few bytes. In order to improve accuracy, a standard method is to use multiple independent basic sketch instances. The number of instances is derived from the desired accuracy and confidence levels. In practice, very good accuracy can be achieved with sketches having size in kilobytes.
- The update of a sketch with a new tuple consists in generating one or more random values and adding them to the sketch counter. The answer to a query involves simple arithmetic operations on the sketch. In the case of multiple sketches, both the update and query are applied to all the instances. Overall, update and query time are linearly proportional with the sketch size.
- A sketch can be computed by partitioning the input relation into multiple parts, building a sketch for every part, and then merging the partial sketches. This mergeable property makes sketches amenable for parallel query processing [5, 6] and optimization.

While previous work addresses how to apply sketches to certain cardinality estimation problems, we are not aware of any work that integrates sketches effectively with plan enumeration.

3 THE COMPASS APPROACH

In this section, we provide a high-level description of the COMPASS query optimization paradigm.

Workflow. The workflow performed by the COMPASS query optimizer is depicted in Figure 2. It consists of a two-step process that requires interaction with the query processor. First, the optimizer extracts the selection predicates and join attributes for every table. A sketch is built for every join attribute while performing the selection query on the base table, and only over the tuples that satisfy the predicate. Figure 2 shows the procedure for table *title*, which has a range predicate and two join conditions—although both join predicates involve the same attribute *t.id*, two independent sketches have to be built. COMPASS leverages the high-parallelism of in-memory databases and the mergeable property of sketches to execute this process with minimal overhead. Two additional optimizations can be applied to further reduce the overhead. Sketches for join attributes from tables without selection predicates can be built offline and plugged-in directly. Sketches can be built only over a sample [35], which, however, incurs a decrease in accuracy. In the second step of the workflow, plan enumeration is performed by estimating the cardinality of all the sub-plans using the sketches built in the first step. This is possible only because the attribute-level sketches we design are incrementally composable. Otherwise, separate sketches have to be built for every enumerated sub-plan. In our example, there are two sketches on attribute *t.id*, one for join *e2* and one for join *e3* in the join graph (Figure 1). The sketch for *e2* is included in all the sub-plans that contain this join attribute—similar for *e3*. In a sub-plan that includes both *e2* and *e3*, these two sketches are first merged and then used in estimation as before. This process is performed incrementally during plan enumeration. Finally, the optimal plan is submitted for execution together with any materialized intermediates.

Partitioned query execution. As shown in Figure 2, COMPASS intertwines query optimization and evaluation by partitioning execution into push-down selection (step 1) and join computation (step 3). Query optimization, i.e., join ordering plan enumeration, is performed in-between these two stages. Since plan enumeration and join computation are standard, we focus on push-down selection, where online sketch building is performed. Push-down selection

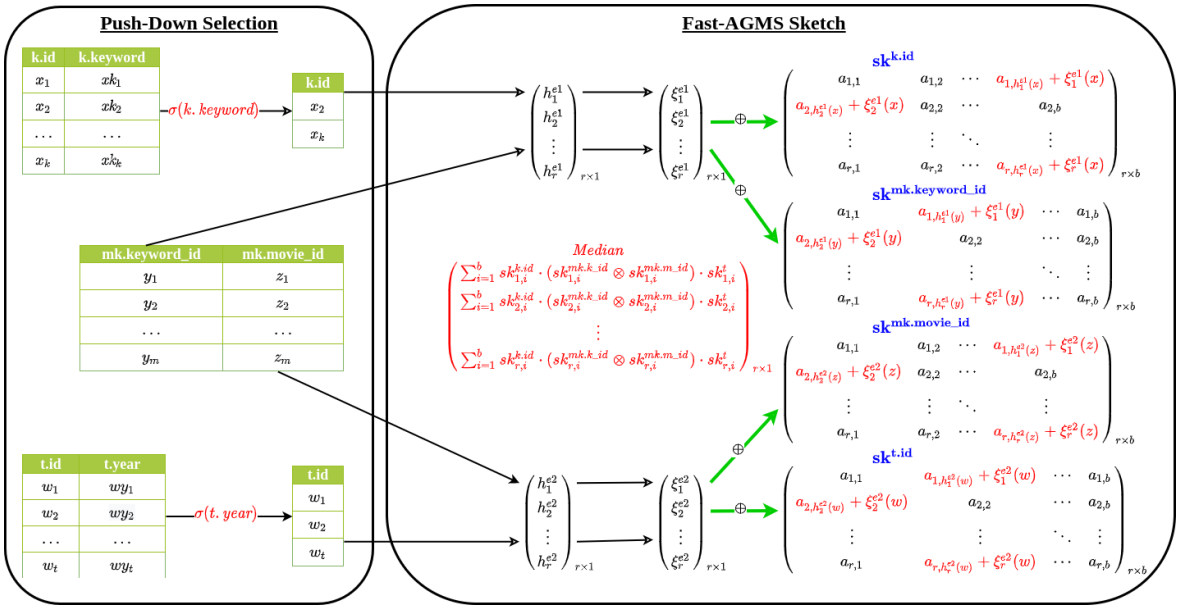


Figure 3: Cardinality estimation for query JOB 6a with (Fast-)AGMS sketches.

computes the exact selectivity cardinalities for all the base tables that have selections. This is similar to the ESC approach introduced in [38]. However, in addition to predicate evaluation, COMPASS also builds sketches for every join attribute in the table by piggybacking on the same traversal—sketch building is performed during the selection. Notice that this works both for sequential and index scans. It is important to emphasize that only the tuples that satisfy the predicate are included in the sketch, which increases their accuracy significantly. Moreover, the sketch update overhead is kept to the minimum necessary. While the exact cardinalities and sketches are always materialized due to their reduced size and role in optimization, the decision to materialize the selection output – the intermediate result – depends on its size. COMPASS follows the same approach as in [38]. If the intermediate size is smaller than a threshold, it is materialized. Otherwise, it is not, since the space reduction does not compensate for the access time reduction. Notice that, even when intermediates are not materialized, sketches still contain only the relevant tuples for join cardinality estimation.

Plan enumeration. The join attribute-level sketches computed during push-down selection can be composed to estimate the cardinality of any valid join order – excluding cross products – generated during plan enumeration. In most cases, cross products are ignored by join enumeration algorithms anyway [24]. As shown in Section 4, sketch composition consists of two stages. First, the sketches of all the relevant join attributes in a table are merged together. An attribute is relevant for a partial join order if its join is part of the order. Second, the sketches across tables are combined to estimate the cardinality of the join order. Since the overall composition consists only of arithmetic operations, sketches can be integrated into any enumeration algorithm—exhaustive, bushy, or left-deep.

Sketches vs. other synopses. The decision to use only sketches in COMPASS may seem questionable given that sketches are designed for specific stream processing tasks, while databases support generic batch-oriented execution. To put it differently, there is a specific sketch for every streaming query, while synopses are for the entire database. To achieve generality, COMPASS has to build a set of sketches for every query—except base tables without predicates. However, this is done concurrently with push-down selection and is highly-parallel, resulting in low overhead (Section 6). As a result, sketches do not require any maintenance under modification operations since they are built on the current data. This is not possible for any of the other database synopses. The benefit of having query-specific synopses is also exploited in [24], where index-based join sampling – a variation of ROX chain sampling [17] – is introduced. Index-based join sampling is performed during the plan enumeration of every query under the corresponding selection predicates. Since the sample size – both minimum and maximum – is carefully controlled, index-based join sampling has improved memory usage and accuracy because it avoids empty results. Compared to sketches, though, this sampling strategy has two serious shortcomings. First, it requires the existence of an index and complete frequency distribution on every join attribute. Sketches require nothing beyond the data. Second, the estimation of every join cardinality requires separate sampling from each of the involved tables. Since this process is time-consuming, plan enumeration is performed bottom-up – or breadth-first – in a limited time budget. Sketches can be composed incrementally in any order without the need to access data. Moreover, they capture correlations by design.

4 SKETCH CARDINALITY ESTIMATION

In this section, we present how the class of AGMS sketches are applied for estimating the cardinality of complex queries involving

selection predicates and multi-way joins. We organize the presentation around the original AGMS sketches [1], which have known solutions to these problems. However, AGMS sketches are too inefficient to be accurate and cannot be integrated in query plan enumeration. This leads us to the Fast-AGMS sketches [5], which are asymptotically more efficient and have been shown to be statistically more accurate [36, 37]. However, Fast-AGMS sketches are limited to estimating two-way join cardinality. Our main contributions are to extend Fast-AGMS sketches to multi-way joins and to effectively integrate them in query plan enumeration.

4.1 AGMS and Fast-AGMS Sketches

The basic **AGMS sketch** [1] of an attribute consists of a single random value *sk* that summarizes the values of all the tuples in the relation. For example, all the values of attribute *id* from table *keyword* can be summarized by a sketch $sk(k.id)$ computed as $sk(k.id) = \sum_{t \in k} \xi(t.id)$, where ξ is a family of $\{+1, -1\}$ random variables that are 4-wise independent. Essentially, a random value of either $+1$ or -1 is associated to each point in the domain of attribute *k.id*. Then, the corresponding random value is added to the sketch $sk(k.id)$ – initialized to 0 – for each tuple t in table *keyword*. Since all the tuples are combined in the same sketch $sk(k.id)$, the sketch value can be far away from the frequency of each single attribute value. However, the 4-wise independence property of ξ guarantees that, for any group of at most 4 different attribute values, the product of their corresponding ξ values is 0 on expectation. This, in turn, allows for each individual attribute value frequency to be unbiasedly estimated by multiplying the sketch with the corresponding ξ random value. For example, the frequency of $k.id = 5$ is estimated by the product $sk(k.id) \cdot \xi(5)$.

The accuracy of this estimator is poor since a table with any number of tuples is summarized as a single number. The standard technique to improve accuracy is to build multiple independent basic sketch estimators. This is achieved by using independent families of random variables ξ. It is theoretically proven that, in order to obtain an estimator with relative error at most ϵ with confidence δ, $O\left(1/\epsilon^2 \log(1/\delta)\right)$ basic sketches are necessary. As shown in Figure 3, they are grouped into a matrix of r rows and b columns. Then, the final AGMS estimator is obtained by averaging the b instances in each row and taking the median over the resulting r averages. Thus, an AGMS sketch has $\Omega(r \cdot b)$ update and query time, and its space usage is also $\Omega(r \cdot b)$.

While **Fast-AGMS sketches** preserve the $(r \times b)$ matrix structure of AGMS sketches, they define a complete row of b counters as a basic sketch element (Figure 3). Only one of these counters is updated for every tuple, thus, a factor b reduction in update time is obtained. The updated counter is chosen by a random hash function h associated with the row. The purpose of h is to spread tuples with different values as evenly as possible—tuples with the same key still end up in the same bucket. On average, a factor b less tuples collide on the same counter, which preserves the frequency of each of them better. Since a full row is a sketch element, a single ξ family of random variables is associated with every row. Thus, a Fast-AGMS sketch with r rows requires only r independent hash and ξ random functions. The value of a counter j is $sk(k.id)_j = \sum_{t \in k, h(t.id)=j} \xi(t.id)$. The Fast-AGMS sketch estimates the frequency of $k.id = 5$ by the product $sk(k.id)_{h(5)} \cdot \xi(5)$, which, although has the same variance as the AGMS estimator [5], has much better statistical accuracy [36].

4.2 Two-Way Join Cardinality Estimation

Consider the join $e1$ between tables *movie_keyword* and *keyword* with predicate *mk.keyword_id=k.id* (Figure 1). The cardinality of this join operator can be estimated with two (Fast-)AGMS sketches $sk(k.id)$ and $sk(mk.keyword\_id)$ built on the join attributes. As depicted in Figure 3, the requirement is that these sketches share the same random functions ξ (and h, for Fast-AGMS)—ξ^{e1} and h^{e1} are associated with edge $e1$. The hash function h lands identical keys to the same bucket, while ξ guarantees that join keys with the same value are assigned the same $\{+1, -1\}$ random value—they are correlated. Since the difference between the AGMS and Fast-AGMS estimator is minor, we give only the former:

$$Est(|e1|) = \sum_{j=1}^{b} sk(k.id)_j \cdot sk(mk.keyword\_id)_j$$

The basic Fast-AGMS unbiased estimator for the cardinality of $|e1|$ sums up the products of the corresponding sketch buckets. Summation is necessary because h partitions the tuples. Due to the 2-universal and 4-wise independence properties of h^{e1} and ξ^{e1}, respectively, this estimator is unbiased. The final estimate is obtained by taking the median of the r independent basic sketches. Thus, the complexity of sketch-based join cardinality estimation is building the sketches. This requires a scan over the tuples in each of the two tables. Fast-AGMS sketches update $\Omega(r)$ counters for each tuple, while the estimate is computed in $\Omega(r \cdot b)$ time. This assumes that the random number generators have small seeds and produce their values fast—aspects that require careful implementation.

4.3 Multi-Way Join Cardinality Estimation

We show how to extend AGMS sketches to multi-way join cardinality estimation. For this, we add the join $e2$ between *movie_keyword* and *title* to $e1$ and aim to estimate the cardinality of this 3-table query. Following the approach for two-way joins, a sketch is built for edge $e2$ on attributes *mk.movie_id* and *t.id*, respectively. These sketches share their own family ξ^{e2} of random variables. Since two attributes from *mk* participate in join operators with other tables, we have to preserve their tuple connection. This is achieved by creating a single composed sketch $sk(mk.k\_id, mk.m\_id)$ instead of separate sketches for each attribute [9]. The value of $sk(mk.k\_id, mk.m\_id)$ is computed as:

$$sk(mk.k\_id,mk.m\_id) = \sum_{t \in mk} \xi^{e1}(t.k\_id) \cdot \xi^{e2}(t.m\_id)$$

where the product of the two random variables is added to the sketch. The cardinality estimator is defined as the product of the three sketches in this case:

$$Est(|e1 \cup e2|) = sk(k.id) \cdot sk(mk.k\_id,mk.m\_id) \cdot sk(t.id) = \sum_{x \in k} \sum_{y \in mk} \sum_{z \in t} \xi^{e1}(x.id) \cdot \xi^{e1}(y.k\_id) \cdot \xi^{e2}(y.m\_id) \cdot \xi^{e2}(z.id)$$

As long as the families ξ^{e1} and ξ^{e2} are independent, this estimator is unbiased. However, its variance can be exponentially worse than

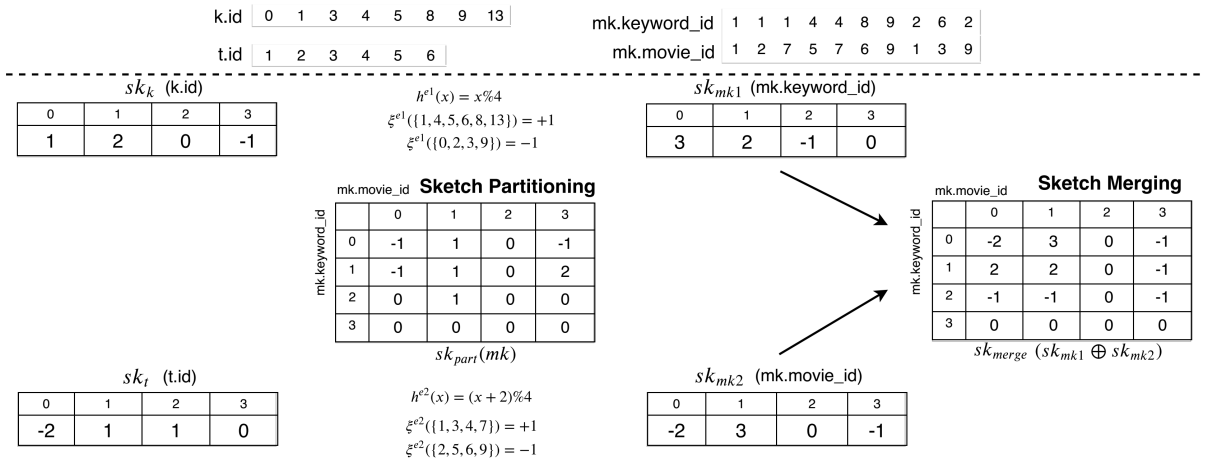


Figure 4: Fast-AGMS sketches for multi-way join cardinality estimation on query JOB 6a.

that of the two-way join estimator. This strategy can be generalized to complex queries involving any number of tables and joins by using independent random families ξ for every join predicate.

As far as we know, there is no work that extends Fast-AGMS sketches to multi-way join estimation. The main problem is posed by the requirement to combine the independent hash functions h^{e1} and h^{e2} for the two join attributes. In the case of the ξ random variables, the composition is realized through their multiplication. The hash functions allocate the attributes to different buckets, which means that the bucket indices have to be composed. It is unknown how to do this in the basic sketch vector such that the relationship between the tuple attributes is not lost.

4.4 Plan Enumeration

In order to apply AGMS sketches to join order enumeration, a separate sketch has to be employed for every subset of joins—which is an exponential number. For example, 7 separate sketches have to be built for table mk because it has 3 join predicates. There are two reasons for this. AGMS sketches cannot be incrementally composed, e.g., $sk(mk.k\_id, mk.m\_id)$ cannot be computed from $sk(mk.k\_id)$ and $sk(mk.m\_id)$. The inverse also does not work. The sub-sketch for a subset of attributes cannot be extracted from a larger sketch, e.g., $sk(mk.k\_id)$ or $sk(mk.m\_id)$ cannot be derived from $sk(mk.k\_id, mk.m\_id)$. These operations are not possible because of the order of multiplication and addition with the random families ξ. In practice, all the sketches for a table can be built in a single scan. However, since the update time per AGMS sketch is linear in the sketch size, updating an exponential number of sketches becomes dominant. Moreover, the space requirement for all the sketches is also a problem. In summary, the application of AGMS sketches to join order enumeration is not scalable, while Fast-AGMS sketches work only for two-way joins.

5 FAST-AGMS SKETCH JOIN ORDERING

We present two strategies to extend Fast-AGMS sketches to multi-way join cardinality estimation. The first strategy – sketch partitioning – is a theoretically sound estimator for a given multi-way join. Its limitation is that it cannot be composed/decomposed, thus, it is not scalable for plan enumeration. The second strategy – sketch merging – addresses the scalability issue by incrementally creating multi-way sketches from two-way sketches. Although this is done heuristically for a certain multi-way join taken separately, all the multi-way joins with a given size are equally impacted. We show empirically that this property is a good surrogate for accuracy—which is much harder to consistently achieve in join enumeration.

5.1 Sketch Partitioning

The idea of sketch partitioning is to reorganize the b buckets of the elementary sketch into a $(b_1 \times b_2)$ 2-D matrix—as done in [3] for frequency sketches. h^{e1} hashes a tuple $mk(k\_id, m\_id)$ to one of the b_1 rows, while h^{e2} hashes to one of the b_2 columns. Then, only the counter at indices $\left[h^{e1}(k\_id), h^{e2}(m\_id)\right]$ is updated with the product $\xi^{e1}(k\_id) \cdot \xi^{e2}(m\_id)$. This process is depicted in Figure 4. For example, tuple (6,3) in mk adds 1 to the counter [2,1]. h^{e1} guarantees that all the tuples with $k\_id = 6$ are hashed to row 2, while h^{e2} sends tuples with $m\_id = 3$ to column 1. Conflicts happen only when the output of both hash functions is identical. Given the quadratic number of buckets compared to the sketch for a single attribute – while the number of tuples is the same – conflicts are less frequent. The cardinality estimate for the 3-table join $k \bowtie mk \bowtie t$ is obtained by summing up all the entries in the matrix resulted after the scalar multiplication between $sk(k.id)$ and every row in $sk_{part}(mk)$, followed by the scalar multiplication between the transpose of $sk(t.id)$ and every column in $sk_{part}(mk)$. This can be written as:

$$Est(|e1 \cup e2|) = \sum_{0 \le i < b_1} \sum_{0 \le j < b_2} sk_k[i] \cdot sk_{part}(mk)[i,j] \cdot sk_t[j]$$

It can be shown theoretically that this estimator is unbiased following the same proof as for AGMS sketches in [9]. Moreover, given the larger size of sketch $sk_{part}(mk)$, its accuracy is expected to be better. This procedure can be generalized to any number of join attributes by partitioning – or replicating – b into the corresponding number of dimensions. For example, a table with 3 joins has a 3-D tensor as

its sketch, with one dimension for every join attribute. Thus, there is a polynomial factor increase in the size of the sketch and the estimate computation. This has to be carefully accounted for in the overall memory budget since the likelihood of conflicts varies with the dimensionality of the sketch tensor. The constraint to have the same number of buckets for a join predicate, e.g., $sk(k.id)$ has as many buckets as the number of rows in $sk_{part}(mk)$, makes memory allocation among sketches more complicated than for the 1-D AGMS sketch vectors.

Partitioned Fast-AGMS sketches are not scalable for join order enumeration. This is because separate sketches are required for every join. For example, in Figure 4, the 2-D sketch $sk_{part}(mk)$ is used for the 3-way join $k \bowtie mk \bowtie t$, while the 1-D sketches sk_{mk1} and sk_{mk2} are used for the 2-way joins $k \bowtie mk$ and $t \bowtie mk$, respectively. Building and storing these many sketches is impractical in query optimization. One alternative is to build only the sketches for up to k-way joins and use other methods to estimate higher-order join cardinality. This strategy is applied for run-time join samples in [24]. The drawback is that other statistics are required for the higher-order joins and the interaction between these statistics and sketches has to be carefully controlled.

Our goal is to exclusively use sketches. Intuitively, we want to be able to either generate the 2-D sketch from the 1-D sketches or extract the 1-D sketches from the 2-D sketch. Unfortunately, none of these have a clear solution for Fast-AGMS sketches. The composition of sk_{mk1} and sk_{mk2} requires to determine how to combine all the pairs of buckets in the 1-D sketches in order to compute the quadratic number of entries in the 2-D sketch. Since the identity of tuples is lost when they are inserted in the 1-D sketch, it is not possible to recreate the tuple and determine its corresponding 2-D bucket. Moreover, due to conflicts in the ξ random functions, we do not even know how many tuples belong to a 1-D bucket. For example, bucket 1 in sk_{mk1} is 2 even though 4 tuples are hashed to it. The extraction of a 1-D sketch from the 2-D sketch also does not work because of the ξ variables. Specifically, the update by the product $\xi^{e1} \cdot \xi^{e2}$ makes it impossible to retrieve the value of a 1-D bucket by summing up the corresponding 2-D buckets. For example, the value of bucket 0 in sk_{mk1} is not the sum of the buckets in row 0 of sketch $sk_{part}(mk)$. This is true only for hash-based sketches [3].

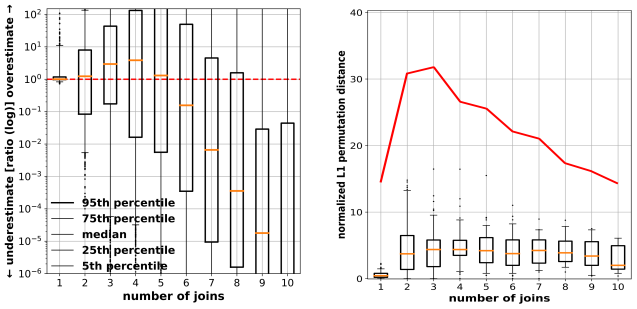
5.2 Sketch Merging

We introduce the sketch merging heuristic as a lightweight method to compose two-way join Fast-AGMS sketches in order to estimate the cardinality of multi-way joins. The procedure works as follows. We build sketches for every two-way join predicate independently, as shown in Figure 4. The number of sketches corresponding to a table is equal to the number of joins it participates in. For example, tables k and t have one sketch, while mk has two sketches. We estimate any join combination generated during plan enumeration using only these sketches. The two-way joins $k \bowtie mk$ and $t \bowtie mk$ are estimated optimally with the sketch pairs (sk_k, sk_{mk1}) and (sk_t, sk_{mk2}), respectively. For the 3-way join $k \bowtie mk \bowtie t$, we create a merged sketch $sk_{merge}(mk) = sk_{mk1} \oplus sk_{mk2}$ from the two 2-way join sketches on demand during plan enumeration. This merged sketch approximates the partitioned sketch $sk_{part}(mk)$ computed with the same random functions, without accessing the tuples. A bucket $[i, j]$ in sk_{merge} is set to the value having the minimum absolute magnitude among the corresponding $[i]$ and $[j]$ buckets in the two basic sketches:

$$sk_{merge}[i,j] = \begin{cases} sk_{mk1}[i], & if\ |sk_{mk1}[i]| \le |sk_{mk2}[j]| \\ sk_{mk2}[j], & if\ |sk_{mk1}[i]| > |sk_{mk2}[j]| \end{cases}$$

For the example in Figure 4, bucket $[0, 0]$ is set to -2 because $|3| > |-2|$, while bucket $[0, 2]$ to 0 because $|0| < |3|$. The reason for this merge procedure is multifolded. The interaction between the random functions ξ is considered – albeit not through a direct multiplication – by preserving the sign of the value in the basic sketch. The absolute magnitude corresponds to the maximum number of tuples with a given join key that are hashed to the bucket—assuming no conflicts. These tuples are partitioned across the buckets of the other join key. The minimum is chosen because this is the maximum number of tuples that can have identical values for both join keys when considered together. However, this is an overestimate because the exact tuple pairing is lost. This can be seen when comparing the magnitude of the values in the two 2-D sketches in Figure 4. In fact, sketch merging is likely to always overestimate join cardinality. The only caveat is the interaction between the ξ functions.

Sketch merging can be generalized to any number of joins by applying the procedure iteratively. Moreover, (n+1)-D sketches can be derived incrementally from n-D sketches in a single step—without the need to always start from the basic sketches. This property can be exploited to speed up the computation and reduce memory usage in bottom-up plan enumeration since only the highest-dimensional sketches have to be maintained. An even more important property of sketch merging is that it is consistent in how it handles the multi-way joins with the same number of predicates. Specifically, all these joins rely on the same basic sketches and the same assumptions for merging. Thus, it is likely that these estimates exhibit similar accuracy behavior—same type of errors for equal join size.



(a) estimate/true (b) L1-distance

Figure 5: Accuracy ratio (a) and L1-distance between the estimated sketch permutation and the correct join order (b).

In order to verify this claim, we depict the accuracy of sketch merging for the JOB queries in Figure 5. We use two measures to quantify accuracy. The first is the ratio between the sketch estimate and the true cardinality for all the enumerated sub-plans having at most ten joins (Figure 5a). We observe that the median ratio is within a factor of 10 for up to six joins, which is better than any previous practical results [24]. For a larger number of joins, sketch

merging generates underestimates systematically. In previous results [25], this behavior occurs starting from 3-way joins.

4-way join	$mk \bowtie ci \bowtie n \bowtie k$	$mk \bowtie ci \bowtie n \bowtie t$	$mk \bowtie ci \bowtie k \bowtie t$
True cardinality	6	1194	1224
Sketch merging	198	*18M*	*6.5M*

Table 1: 4-way join L1 permutation distance for JOB 6a.

The second measure is the normalized L1 distance [51] between the permutation generated by sketch merging and the correct join order. Given n sub-plans of the same size, the correct order C is obtained by sorting them in increasing order of their cardinality. The permutation corresponding to sketch merging $\mathcal{S}$ is obtained by sorting the sub-plans based on the sketch estimates. The L1 distance is defined as $\sum_{i=1}^{n} |S_i - C_i|$, the sum of the differences between the position in the permutation and the correct order. For example, the L1 distance for the 4-way joins in query JOB 6a (Table 1) is $0 + 1 + 1 = 2$. The normalized L1 distance – we divide the distance by the number of sub-plans in the query – is depicted in Figure 5b. The closer the distance is to zero, the more similar is the permutation to the correct order. For reference, we plot the line corresponding to the maximum L1 distance. The join orders generated by sketch merging have an L1 distance that is significantly below the maximum. In particular, for 2-way join sub-plans, the distance is almost zero, while for sub-plans with more joins, the distance is constantly below 10. This confirms that sketch merging selects orders that are close to optimal most of the time.

6 EMPIRICAL EVALUATION

We perform an extensive experimental study over the complete JOB benchmark [25] in order to evaluate the performance of COMPASS and compare it against four other database query optimizers (Figure 1). Our main goal is to determine whether COMPASS is a complete optimizer, rather than limit ourselves only to the Fast-AGMS sketch accuracy—which is depicted in Figure 5. This requires an effective integration of cardinality estimation in plan enumeration. To this end, our evaluation investigates the following questions:

- What is the quality of the query execution plans generated by COMPASS? We measure plan quality as the total cardinality of the intermediate results since this is independent from specific execution engine optimizations. Moreover, logical optimizers use cardinality information as the main criterion to rank plans.
- What is the execution time – or runtime – for the COMPASS plans? Since this is highly dependent on the underlying query processing engine, we execute the plans in MapD, PostgreSQL, and DBMS A. This allows us to identify the correlation – if there is one – between plan quality and execution time.
- What is the overall JOB workload runtime? While individual queries allow for localized analysis, the workload execution time measures the reliability of COMPASS. However, due to the high variance in JOB query complexity, this measure alone is not an absolute indicator of the quality of an optimizer.
- What is the optimization overhead incurred by sketch merging in plan enumeration? While significantly improving upon sketch partitioning, it is not clear if online sketch merging during push-down selection is small enough to be practical.

6.1 Experimental Setup

Implementation. We implement COMPASS in MapD (version 3.6.1) [53]. The source code is publicly available [46]. MapD has a highly-parallel GPU-accelerated query execution engine. Relational operators are compiled into CUDA kernels that are executed concurrently across the SIMD GPU architecture. In order to reduce data movement, MapD compiles multiple relational operators into a single CUDA kernel. For joins, this corresponds to a worst-case optimal join algorithm [32]. The MapD query optimizer, however, is not as sophisticated as its execution engine. It relies on the Calcite SQL compiler [52] to get a lexicographic – in the order in which the query is written – query execution plan. The join order is computed based on a primitive heuristic that sorts the tables in decreasing order of their cardinality. Moreover, selection predicates are not considered in the optimization. COMPASS brings a principled cost-based optimization procedure to the MapD query optimizer.

The COMPASS implementation consists of two modules—a scan operator that integrates Fast-AGMS sketch construction with push-down selection and a lightweight join order enumeration algorithm. For sketch construction, we adapt a publicly available two-way join Fast-AGMS sketch implementation [50] to the MapD CUDA kernel API. This requires parallelizing both the update and the estimation functions. The scan operator filters only the relevant tuples to be passed to the sketch update. Since separate sketch instances are created for every GPU block, this requires an additional merge stage—currently performed on the CPU.

The COMPASS plan enumeration algorithm uses the join graph in Figure 1 to guarantee that only valid join orders are considered and cross products are ignored. Plan enumeration becomes a graph traversal problem. The algorithm performs a greedy depth-first search (DFS) traversal that adds tables incrementally to the current partial join order. At each step, the table that has the lowest cardinality estimate when joined with the already selected tables is added to the plan. The estimates are computed using the (multi)-way join Fast-AGMS sketches. A table is a candidate only if it joins with a table that is already part of the plan. While the incremental step is clear, the initialization is not. Rather than choosing a single starting table, COMPASS performs join enumeration from all the tables in the query. This enlarges the space of candidate plans—all of which are left-deep trees. However, a stopping criterion that compares the current estimate with the best estimate so far, prunes sub-optimal plans early. In summary, COMPASS considers a number of plans that is at most equal with the number of tables in the query.

Database systems & hardware. In addition to MapD, the other three systems we use are PostgreSQL (version 11.5), MonetDB (version 11.33.11), and the commercial DBMS A. PostgreSQL and DBMS A are used as common ground in all the experiments because of their extensibility. Both of them allow us to inject and execute the join orders computed by the other databases—the `CROSS JOIN` statement in PostgreSQL and the hints in DBMS A, respectively. We configure PostgreSQL with 16GB memory per operator, 32GB OS buffer cache size, and we force the optimizer to use dynamic programming in plan enumeration for queries with no more than 18 join predicates. These settings follow prior art [25]. We use an optimized docker image publicly available for DBMS A, while for MonetDB we keep the default configuration. All the systems run

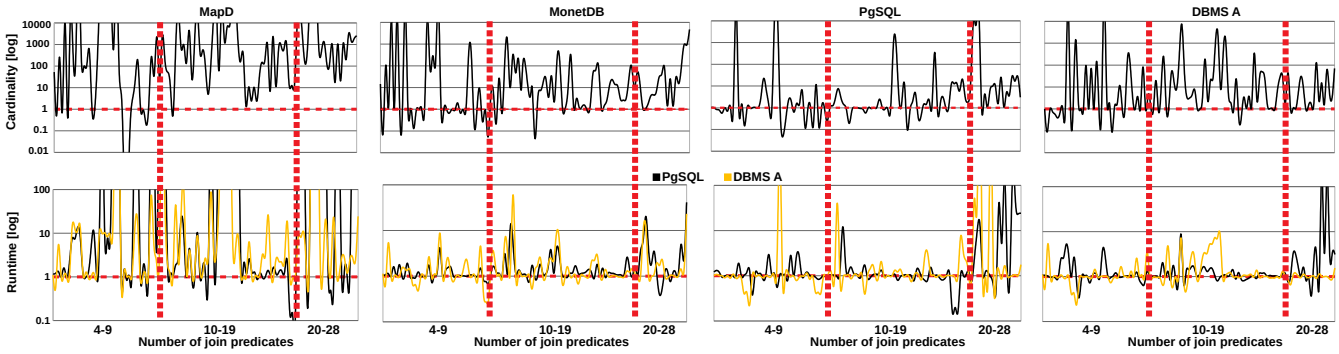


Figure 6: Cardinality and runtime – in PostgreSQL and DBMS A – as a normalized ratio to COMPASS.

on a Ubuntu 16.04 LTS machine with 56 CPU cores (Intel Xeon E5-2660), 256GB RAM, and an NVIDIA Tesla K80 GPU.

Dataset and query workload. We perform the experiments on the IMDB dataset [45], which has been used extensively to evaluate query optimizers [23] and has become a de-facto standard. The JOB benchmark [49] defines 113 queries – grouped into 33 families – over the IMDB dataset. These queries vary significantly in their complexity, with the simplest one having 4 joins and the most complex one having 28 joins. This variability manifests itself in execution times that are highly-different. To compensate for this, we split the queries into three groups and examine each group separately. These groups are based on the number of joins in the query: group1 contains queries with 4-9 joins; group2, 10-19 joins; and group3, queries with 20-28 joins, respectively.

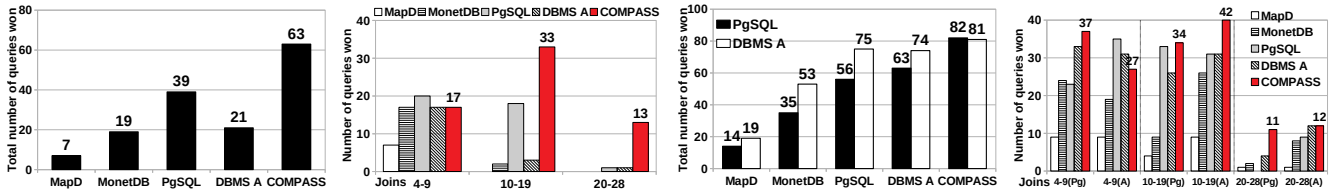
Methodology. To quantitatively assess the quality of a join order plan, we use two metrics—intermediate result cardinality and query execution time. The total cardinality of the intermediate results quantifies how many tuples are produced by all the joins in the plan. The lower this number is, the better the plan. This is the primary metric used in logical query optimization to estimate the cost of a plan. However, the actual execution time depends on specific query processing optimizations. Thus, the execution time is not entirely correlated with the cardinality.

In order to fairly evaluate the join orders produced by every database, we use both PostgreSQL and DBMS A as common ground. First, we run the queries in each database and collect their join plans. Then, we inject these plans into PostgreSQL and DBMS A, respectively, and measure their runtime. Moreover, we execute all the subqueries in the plans to compute the intermediate cardinality. Notice that every system generates its plan independently based on its own algorithm and statistics. PostgreSQL and DBMS A serve as common execution engines for all the plans. We argue that this procedure allows for a holistic comparison of the query optimizers—independent of the execution engine.

6.2 Results

Query-level analysis. In this experiment, we compare COMPASS against every other system for each query in the JOB benchmark. We measure both the intermediate result cardinality, as well as the execution time—taken as the median value over 9 runs. The execution plans are obtained by performing the query in each system. These are subsequently injected in PostgreSQL and DBMS A, and executed on the same execution engine. The cardinalities are generated by executing all the subqueries in the plan in the corresponding order—which is done in PostgreSQL. This information is extracted from the individual plans. Figure 6 depicts the results – cardinality on the upper part of the figure, runtime on the lower part – normalized to COMPASS. All the values are divided by the COMPASS results—represented as a horizontal dotted line at position 1 on the y-axis. A point below this line means that the other system has a better result, otherwise, COMPASS performs better. The results are grouped by the number of joins in the JOB queries (x-axis) and separated by two dotted vertical lines.

MapD consistently produces execution plans that have cardinality two orders of magnitude or larger than COMPASS. With a few exceptions, all MapD plans are worse. There is one such query – the discontinuity going to zero in the figure – that indeed has cardinality zero and MapD correctly detects it. However, this is only a matter of chance because the first join in the plan – between the largest tables in the query – does not produce any results. The reason for this poor plan quality is the lack of statistics in the MapD query optimizer. Decisions are taken solely based on the full table cardinality—the number of tuples before any selection predicate. Therefore, the resulting plans are highly sub-optimal. While runtime follows cardinality – with many results 100X slower than COMPASS – the correlation between the two is not complete. There are several queries for which the MapD cardinality is considerably worse, while the execution time is similar or better than COMPASS. This is the case for some of the complex queries with 20 or more joins executed in PostgreSQL. In this situation, MapD chooses a large well-connected table early in the plan. This allows it to check many join predicates at the beginning and prune a large number of tuples. On the other hand, COMPASS – and the other systems – start from small tables on the periphery of the join graph and make their way to the highly-connected tables in the center. This strategy produces many staged intermediate results that increase the runtime. While the runtime trend across PgSQL and DBMS A is similar, we observe that queries with 20 or more joins are handled better by DBMS A, while queries with less than 20 joins are faster in PgSQL. This is an indication that DBMS A is better optimized for complex queries.



(a) Cardinality comparison (b) Cardinality vs join number (c) Runtime comparison (d) Runtime vs join number

Figure 7: Distribution of winning queries in terms of cardinality and runtime in PostgreSQL and DBMS A.

The trend of the cardinality results in **MonetDB** follows the one in MapD. While the majority of the results are worse than COMPASS, the ratio is smaller than for MapD. This improvement is due to the more advanced rule-based MonetDB query optimizer with limited statistics support. However, compared to the full sketch-based COMPASS, the MonetDB cardinalities are considerably worse—many times an order of magnitude or more. Interestingly enough, though, the corresponding query runtimes fare much better than predicted by the cardinality. With few exceptions, they are always within a factor of 10 – more often less – off of COMPASS. Moreover, they are independent of query complexity and do not exhibit spikes. Overall, the MonetDB runtimes are the most consistent with COMPASS across both PgSQL and DBMS A. This is because the MonetDB query optimizer finds plans that are executed similarly to COMPASS—albeit they have higher cardinality.

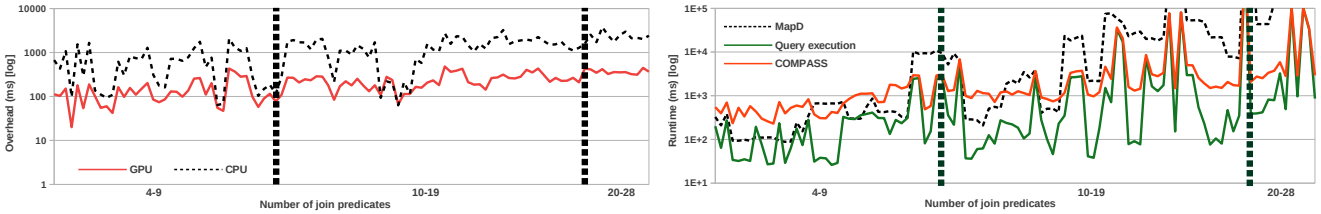
The cardinality results for **PostgreSQL** – PgSQL in the figure – are the closest to COMPASS among all the systems. This is entirely due to the advanced statistics the PostgreSQL optimizer employs. While mildly better than COMPASS for several queries, PostgreSQL still exhibits spikes that go beyond a factor of 1000X. The reason is the failure to detect correlations between join attributes. Since the plans are optimized for the PostgreSQL execution engine in this case, we expect the runtimes to be optimal. This is indeed the case for queries with less than 20 joins. However, for 20 or more joins, the PostgreSQL runtime is considerably worse compared to COMPASS. This is where the PostgreSQL optimizer drops dynamic programming in plan search. With a few exceptions where there are dramatic spikes that go beyond 100X, the PgSQL plans executed in DBMS A perform as well as or better than in PostgreSQL itself. This is especially true for the complex queries having 20 or more joins. Overall, COMPASS generates more stable plans than PostgreSQL. Although not specifically optimized for it, PostgreSQL executes them as fast – or faster – than its own plans.

The commercial **DBMS A** produces plans that have consistently higher cardinality than COMPASS across all the JOB queries. This clearly shows that the employed statistics do a poor job at estimating the join cardinality. However, when executed in PostgreSQL, these plans have unexpectedly good runtimes—except for queries with more than 20 joins. This is likely due to the more complex cost function that considers other parameters beyond cardinality in determining the optimal plan. Interestingly enough, when executing its own plans, DBMS A does not fare better than PostgreSQL, except for the complex queries with more than 20 joins. In fact, DBMS A has worse runtime for queries with 10 to 20 joins. The runtimes of DBMS A and COMPASS are close to each other and always within a factor of 10X. This confirms that the COMPASS plans are also optimal for DBMS A.

Aggregated workload statistics. We aggregate the query-level results (Figure 6) in order to obtain an overall view of the relative performance of the compared systems. These aggregated results are depicted in Figure 7. They give the total number of queries for which a database performs the best, as well as the distribution as a function of the number of joins in the query. In the case of cardinality, a database is counted if it achieves the minimum cardinality among all the databases. For runtime, a database is counted if it comes within 10% of the fastest runtime—computed as the median of 9 runs. This bound compensates for variations in the environment.

Based on Figure 7a, COMPASS achieves the plan with the minimum cardinality for 63 out of the 113 JOB queries. This represents approximately 56% of the workload. PostgreSQL (PgSQL) comes in second place with 39 queries. The other three databases obtain the best cardinality in less than 20% of the queries each, with MapD winning only 7 queries. The careful reader notices that the sum of the winning queries is larger than 113. This is because there are queries for which two or more systems achieve the same best cardinality—case in which we count each of them. The distribution of the winning queries in terms of the number of joins is depicted in Figure 7b. While for the simpler queries with less than 10 joins all the systems perform similarly, COMPASS clearly dominates the others when the complexity increases. PostgreSQL is the only other database that performs sufficiently well, however, only for queries with a moderate number of joins. These results prove the benefit of using statistics in query optimization, especially for complicated queries. While the PostgreSQL statistics perform well for simple to moderate queries, COMPASS sketches are less sensitive to the number of joins in the query—they provide more consistent estimates. Moreover, COMPASS is not heavily impacted by the greedy join enumeration algorithm. When PostgreSQL switches from dynamic programming – more than 18 joins – it fails to find any best plan.

The aggregated runtime results in PgSQL and DBMS A are depicted in Figure 7c and 7d. They follow closely the corresponding cardinality results—with one exception. The runtime for the commercial DBMS A is much better than its cardinalities anticipate—DBMS A has the best runtime for 63 and 74 queries, while its cardinality is best only for 21 queries. The reasons are outlined when the individual query results are discussed. Additionally, DBMS A benefits from the bound on runtime since it often comes within the fastest system. Overall, COMPASS achieves the fastest runtime for 82 (PgSQL) and 81 (DBMS A) out of the 113 JOB queries – 72% of the workload – which is more than any other database.



(a) Sketch building overhead **(b) Query runtime**

Figure 8: Performance of the COMPASS optimizer implementation in MapD.

This proves the superiority of the identified plans and confirms the correlation between cardinality and runtime. The correlation manifests more clearly for queries with a larger number of joins because of the higher runtime, which makes ties more unlikely. Moreover, the correlation is stronger for PgSQL than for DBMS A since the number of winning queries is higher in DBMS A for all systems except COMPASS. A careful reader observes that the runtime results are higher than the cardinality results for all the systems—and larger than 113 when summed up. This is because it is more common to have close-enough runtimes than it is to have the same cardinality—multiple counting is more frequent. Based on these results, we conclude that COMPASS is the optimizer with the most consistent and resilient plans on the JOB benchmark.

Database	Runtime (minutes)		Ratio to COMPASS	
	PgSQL	DBMS A	PgSQL	DBMS A
MapD	>300	>300	**>23**	**>13**
MonetDB	27.52	35.71	**2.19**	**1.65**
PgSQL	103.00	244.31	**8.20**	**11.28**
DBMS A	70.72	29.22	**5.63**	**1.35**
COMPASS	**12.56**	**21.66**	1.00	1.00

Table 2: JOB benchmark runtime in PgSQL and DBMS A.

Total workload runtime. The runtimes for the complete JOB workload execution in PostgreSQL and DBMS A using the plans generated by each database are included in Table 2. Given the high variance among queries, these numbers have to be taken with a grain of salt since they may be dominated by a few complex queries with a large number of joins. Nonetheless, we follow prior art [3, 39] and include them together with the aggregated workload statistics. As expected, COMPASS has the overall fastest runtime. Somewhat unexpectedly, MonetDB comes in second for the PgSQL execution with a runtime that is almost twice as large as that of COMPASS. The reason is because MonetDB does not fail dramatically for any of the JOB queries. While it performs consistently slower, it never derails on heavily sub-optimal plans. The runtime for PgSQL and DBMS A in PgSQL is dominated by the long-running queries with 20 or more joins, which pull the total time to more than 8X and 5X that of COMPASS. These outliers are sufficient to skew the overall runtime. In the case of MapD, there are 30 queries that do not finish execution even after a timeout of 20 minutes per query. Thus, the very large runtime. When the workload is executed in DBMS A, all systems except DBMS A incur an increase in runtime. The increase is most significant for COMPASS as it stands at 50% more than in PgSQL. On the other hand, DBMS A has a reduction of more than 50% of its PgSQL runtime. Nonetheless, COMPASS still has the overall fastest runtime, which is 35% faster than DBMS A.

COMPASS overhead. We measure the optimization overhead of building Fast-AGMS sketches, as well as that of sketch-based plan enumeration, for the COMPASS MapD implementation. Sketch building can be performed either on GPU or CPU, while merging and plan enumeration are performed on CPU. The results are depicted in Figure 8a. As expected, sketch building is more efficient on GPU than on CPU due to the higher degree of parallelism. In both cases, the optimization overhead increases with the number of joins in the query. For GPU, the overhead is in the order of hundreds of milliseconds (ms), with a maximum of around 500 ms for certain complex queries. For CPU, the overhead is always below 5 seconds, which is relatively small for queries that take minutes to run. Given that this is only a prototype, we believe that the sketch overhead can be further reduced with more optimized code.

Database	Queries won	Runtime (minutes)	Ratio to COMPASS
MapD	42	47.64	**7.67**
COMPASS	**74**	**6.21**	1.00

Table 3: JOB benchmark execution in MapD.

Runtime in MapD. We evaluate the impact COMPASS has on the MapD database. For this, we replace the default MapD query optimizer with COMPASS and execute the JOB benchmark in both scenarios. We measure the end-to-end query runtime, as well as only the query execution time without optimization—these are the same in MapD. Figure 8b depicts the results for every query. We observe that MapD outperforms COMPASS for the simple and some moderate queries. This may be surprising given the primitive MapD query optimizer. However, its execution engine is quite different from PostgreSQL. It is highly-optimized for parallel in-memory processing. This alleviates the need for careful optimization on simple queries. For more complicated queries, though, sketch-based optimization pays off as COMPASS finds considerably better plans. In fact, MapD fails for 8 queries and times out after 30 minutes for 8 other queries. COMPASS finishes all the queries and is faster than MapD for 74 of them, which represents 65% of the workload. The total runtime for the 97 queries MapD successfully runs is included in Table 3. COMPASS has a runtime of 6.21 minutes to MapD's 47.64—which is a net speedup of 7.67X. This proves both that sketches can be effectively computed at runtime, as well as

their benefit to generate better query plans, which result in faster execution. The last point is clear when we compare only the execution time, without optimization overhead. There are less than ten COMPASS plans that have execution time larger than MapD.

6.3 Summary

Based on the presented results, we can answer the questions raised at the beginning of the experimental section:

- COMPASS generates query plans with the lowest cardinality among all the considered systems for 56% of the queries in the workload. This percentage increases to 65% for complicated queries with 10 or more joins.
- The better plans identified by COMPASS translate into faster query runtimes in PostgreSQL, DBMS A, and MapD. Out of the 113 JOB queries, COMPASS achieves the fastest runtime for more than 80 in PostgreSQL and DBMS A, and 74 in MapD. This confirms the correlation between cardinality and runtime. DBMS A is the only database that does not satisfy this correlation, which can be problematic for a user.
- COMPASS and MonetDB are the only databases that perform all the JOB queries without serious hiccups both in PostgreSQL and DBMS A. The other systems have several queries for which the runtime "explodes". This results in significantly higher workload runtime. On the PostgreSQL engine, COMPASS outperforms MonetDB by a factor of 2.19X, while on DBMS A by 1.65X. DBMS A optimizes queries specifically for its engine, resulting in a significant reduction in runtime compared to PostgreSQL. However, COMPASS is faster by a factor of 1.35X. Moreover, COMPASS is at least 7.67X faster than MapD.
- The overhead incurred by the COMPASS optimizer in MapD is less than 500 milliseconds on GPU and less than 5 seconds on CPU. While this may be too large for simple queries, it results in faster execution for more than 91% of the queries. We plan to optimize our implementation in the future.

The extended version of the paper [15] contains more details and additional experiments.

7 RELATED WORK

Cardinality estimation. While exhaustive surveys on query optimization [4, 48] argue that each component is important in finding the optimal plan, Leis et al. [23, 25] show experimentally that cardinality estimation is the most dominant component in query optimization. However, consistency in estimations is more important than high accuracy only for a limited number of instances. There are four mainstream cardinality estimation approaches in the literature—histograms, sampling techniques, sketches, and, more recently, machine learning models. While histograms can provide accurate selectivity estimation for a single attribute in a relation [14], it is difficult for them to capture correlations between cross-join attributes [34], thus limiting their applicability to joins. Unlike histograms, sampling techniques [24, 31] can detect arbitrary correlations for common values. However, samples are sensitive to skewed and sparse data when few tuples are selected by a query [42]. Estimating the cardinality of multi-way joins with AGMS sketches is introduced in [8, 9], while a statistical analysis of two-way join sketch-based techniques is performed by Rusu and Dobra [36, 37]. Vengerov et al. [40] present an extension to AGMS sketches that captures selection predicates, while Cai et al. [3] introduce bound sketches that provide theoretical upper bounds for cardinality estimation. Kernel density models for cardinality estimation (KDE) are introduced in [12, 18]. They are built on samples extracted either from the base tables or the join. While their accuracy is shown to be superior to any other method on JOB queries over at most five tables – the simplest in the benchmark – it is not clear how to generalize and fully integrate KDE models in plan enumeration. Specifically, the KDE implementation [47] builds a separate estimator for every query. No details are provided on how to apply the estimator to query sub-plans derived from the main query, which is the centerpiece of plan enumeration.

Query reoptimization. In order to overcome the inherent misestimations in the query optimizer, Adaptive Query Processing [7] allows the query processor to modify the optimal query plan computed by the optimizer in case of large deviations from the true cardinality values detected at runtime. The Mid-Query Re-Optimizer [16], ROX [17], and SkinnerDB [39] re-run the query optimizer at runtime in the case of large differences between estimations and the true cardinalities. These approaches use the output of the query executor and sampling techniques to re-estimate the cardinalities based on already computed intermediate join outputs and change the query plan whenever the estimated values deviate significantly. In the self-adaptable LEO optimizer [30], the query engine monitors and uses the feedback from the execution engine in order to adjust the histogram-based synopses for better performance in subsequent queries. Eddies [2] process batches of tuples by following dynamic routing policies during query execution. Unlike these systems, COMPASS performs query optimization as a single stage, while query execution is partitioned into two phases—before and after the optimization.

Machine learning for query optimization. Using machine learning techniques and deep neural networks is a recent trend in query optimization. Join order enumeration [22, 28, 29], cardinality estimation [19, 20, 26, 27, 33, 41], selectivity estimation [10, 11, 43], and index structures [21] have been active research directions. For example, Kipf et al. [19] use multi-set convolutional neural networks in order to model join and selection predicates, and capture join correlations in the data. Marcus et al. [29] use reinforcement learning in order to efficiently explore the search space and find optimal join order plans. Different from these approaches, COMPASS uses traditional randomized algorithms to estimate cardinality.

8 CONCLUSIONS AND FUTURE WORK

We introduce the online sketch-based COMPASS query optimizer, which uses exclusively Fast-AGMS sketches for cardinality estimation and plan enumeration. We show that COMPASS outperforms four other databases on all the considered metrics over the JOB benchmark. In future work, we plan to investigate alternative merging strategies for Fast-AGMS sketches in order to support multi-way joins and SIMD-optimized sketch algorithms – for CPU and GPU – with lower overhead.

Acknowledgments. This work is supported by NSF award number 2008815 and by a U.S. Department of Energy Early Career Award (DOE Career).

REFERENCES

[1] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. In *PODS 1999*, pages 10–20.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD 2000*, pages 261–272.

[3] W. Cai, M. Balazinska, and D. Suciu. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *SIGMOD 2019*, pages 18–35.

[4] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS 1998*, pages 34–43.

[5] G. Cormode and M. Garofalakis. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *VLDB 2005*, pages 13–24.

[6] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundation and Trends in Databases*, 4:1–294, 2012.

[7] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[8] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *SIGMOD 2002*, pages 61–72.

[9] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-Based Multi-query Processing over Data Streams. In *EDBT 2004*, pages 551–568.

[10] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. Narasayya, and S. Chaudhuri. Selectivity Estimation for Range Predicates Using Lightweight Models. *PVLDB*, 12(9):1044–1057, 2019.

[11] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Multi-Attribute Selectivity Estimation Using Deep Learning. *CoRR*, arXiv:1903.09999v2, 2019.

[12] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD 2015*, pages 1477–1492.

[13] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[14] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. *SIGMOD Record*, 20(2):268–277, 1991.

[15] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. Online Sketch-based Query Optimization. *CoRR*, arXiv:2102.02440, 2021.

[16] N. Kabra and D. J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *SIGMOD 1998*, pages 106–117.

[17] A. R. Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: Run-time Optimization of XQueries. In *SIGMOD 2009*, pages 615–626.

[18] M. Kiefer, M. Heimel, S. Breß, and V. Markl. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *PVLDB*, 10(13):2085–2096, 2017.

[19] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities:Estimating Correlated Joins with Deep Learning. In *CIDR 2019*.

[20] A. Kipf, D. Vorona, J. Muller, T. Kipf, B. Radke, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Estimating Cardinalities with Deep Sketches. *CoRR*, arXiv:1904.08223v1, 2019.

[21] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD 2018*, pages 489–504.

[22] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR*, arXiv:1808.03196v2, 2018.

[23] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[24] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR 2017*.

[25] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDB Journal*, 27:643–668, 2018.

[26] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *CASCON 2015*, pages 53–59.

[27] T. Malik, R. C. Burns, and N. V. Chawla. A Black-Box Approach to Query Cardinality Estimation. In *CIDR 2007*.

[28] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *VLDB Journal*, 12(11), 2019.

[29] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM 2018*.

[30] V. Markl, G. M. Lohman, and V. Raman. LEO: An Autonomic Query Optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[31] M. Muller, G. Moerkotte, and O. Kolb. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *PVLDB*, 9(11):1016–1028, 2018.

[32] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-Case Optimal Join Algorithms. In *PODS 2012*, pages 37–48.

[33] J. Ortiz, M. Balazinska, J. Gehrke, and S. Sathiya Keerthi. An Empirical Analysis of Deep Learning for Cardinality Estimation. *CoRR*, arXiv:1905.06425v2, 2019.

[34] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB 1997*, pages 486–495.

[35] F. Rusu and A. Dobra. Sketching Sampled Data Streams. In *ICDE 2009*, pages 381–392.

[36] F. Rusu and A. Dobra. Statistical Analysis of Sketch Estimators. In *SIGMOD 2007*, pages 187–198.

[37] F. Rusu and A. Dobra. Sketches for Size of Join Estimation. *TODS*, 33(15), 2008.

[38] J. H. Shin, F. Rusu, and A. Suhan. Exact Selectivity Computation for Modern In-Memory Database Query Optimization. *CoRR*, arXiv:1901.01488v1, 2019.

[39] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD 2019*, pages 1153–1170.

[40] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen. Join Size Estimation Subject to Filter Condition. *PVLDB*, 8(12):1530–1541, 2015.

[41] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality Estimation with Local Deep Learning Models. In *aiDM 2019*, pages 1–8.

[42] W. Wu. Sampling-Based Cardinality Estimation Algorithms: A Survey and An Empirical Evaluation, 2012.

[43] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Selectivity Estimation with Deep Likelihood Models. *CoRR*, arXiv:1905.04278v2, 2019.

[44] F. Yu, W. Hou, C. Luo, D. Che, and M. Zhu. CS2: A New Database Synopsis for Query Estimation. In *SIGMOD 2013*.

[45] P. Boncz. The IMDB Dataset. http://homepages.cwi.nl/~boncz/job/imdb.tgz.

[46] Y. Izenov. The COMPASS Query Optimizer. https://github.com/yizenov/compass_query_optimizer.

[47] M. Kiefer. join-kde. https://github.com/martinkiefer/join-kde.

[48] G. Lohman. Is Query Optimization a Solved Problem? https://wp.sigmod.org/?p=1075, 2014.

[49] G. Rahn. Join Order Benchmark (JOB). https://github.com/gregrahn/join-order-benchmark.

[50] F. Rusu. Sketches for Size of Join Estimation. https://faculty.ucmerced.edu/frusu/Projects/Sketches.

[51] StackExchange. Distance Between Two Permutations? https://math.stackexchange.com/questions/2492954/distance-between-two-permutations.

[52] Apache Calcite. https://calcite.apache.org.

[53] MapD. www.omnisci.com.

[54] MonetDB. www.monetdb.org.

[55] PostgreSQL. www.postgresql.org.