Join Cardinality Estimation with OmniSketches

David Justen
TU Berlin & BIFOLD
david.justen@tu-berlin.de

Matthias Boehm
TU Berlin & BIFOLD
matthias.boehm@tu-berlin.de

***Abstract*—Join ordering is a key factor in query performance, yet traditional cost-based optimizers often produce sub-optimal plans due to inaccurate cardinality estimates in multi-predicate, multi-join queries. Existing alternatives such as learning-based optimizers and adaptive query processing improve accuracy but can suffer from high training costs, poor generalization, or integration challenges. We present an extension of OmniSketch—a probabilistic data structure combining count-min sketches and K-minwise hashing—to enable multi-join cardinality estimation without assuming uniformity and independence. Our approach introduces the OmniSketch join estimator, ensures sketch interoperability across tables, and provides an algorithm to process alpha-acyclic join graphs. Our experiments on SSB-skew and JOB-light show that OmniSketch-enhanced cost-based optimization can improve estimation accuracy and plan quality compared to DuckDB. For SSB-skew, we show intermediate result decreases up to 1,077x and execution time decreases up to 3.19x. For JOB-light, OmniSketch join cardinality estimation shows occasional individual improvements but largely suffers from a loss of witnesses due to unfavorable join graph shapes and large numbers of unique values in foreign key columns.**

I. Introduction

Query optimization is a critical problem in database systems that has been investigated for many decades [1]–[3]. For relational analytics workloads, join ordering contributes heavily to the overall performance as query plans with sub-optimal join orders can be penalized with execution time increases of multiple orders of magnitude [4].

Traditional Query Optimization: Database systems traditionally employ cost-based query optimization [3]. In this process, the optimizer enumerates sub-plans of the query, assigns a cost to each sub-plan by estimating its cardinality, and finally combines them to find the query plan with the lowest cost. One of the key goals of join order enumeration is to minimize the number of intermediate results. During cardinality estimation, optimizers often rely on database statistics and sketches such as histograms. However, for multi-predicate and multi-join queries, these techniques resort to assuming data uniformity and independence, leading to consistent under-estimations and sub-optimal plan generation [5].

Learning-based Optimization: To address these limitations, two major research directions have emerged. Learning-based optimizers [6]–[9] have been introduced to improve query plan quality. These approaches train machine learning models on the database and sample queries and use the model to predict query plans for incoming queries. While learning-based optimizers have shown improvements over traditional optimizers, some of them require extensive initial training phases, and do not generalize well for unknown queries [10].

Adaptive Query Processing: As an alternative line of research, numerous adaptive query processing (AQP) techniques have been introduced over the last two decades [11]–[14]. Instead of relying on cardinality estimates during query optimization, AQP approaches gather statistics such as selectivities during query execution and continuously adapt the query plan based on the collected telemetry. Although many of these techniques have shown promising results, adaptive join reordering approaches are often difficult to integrate into existing systems or make substantial applicability concessions [14].

OmniSketch Cardinality Estimation: While learning-based optimization and AQP are active areas of research, new sketches that may improve traditional query optimization are also emerging. The OmniSketch [15] combines count-min sketches [16] with K-minwise hashing [17] and allows for multi-attribute cardinality estimates with probabilistic error guarantees. In this work, we extend the OmniSketch to enable cardinality estimation for multi-join queries and assess the extended OmniSketch in the context of cost-based query optimization to examine its trade-offs.

Contributions: Our primary contribution is the extension of OmniSketches for multi-join cardinality estimation. The code is open-source as a C++ library on GitHub[1]. Our detailed contributions are the following:

- We introduce an OmniSketch design adaption that ensures the multi-table interoperability via primary and foreign key constraints (Section III-A).
- We define a strategy for join cardinality estimation that reformulates joins to sampled set membership predicates and a second approach that makes use of secondary sketches (Section III-B).
- We contribute an algorithm to derive OmniSketch operation plans from alpha-acyclic join graphs for multi-join cardinality estimates (Section III-C).
- With the JOB-light and SSB-skew benchmarks, we systematically study the benefits, shortcomings, and trade-offs of cost-based query optimization with OmniSketches as opposed to DuckDB (Section IV).

II. Background

The OmniSketch [15] addresses a critical limitation in streaming synopses: traditional sketches such as Bloom fil-

[1]C++ library available at https://github.com/d-justen/OmniSketchCpp.

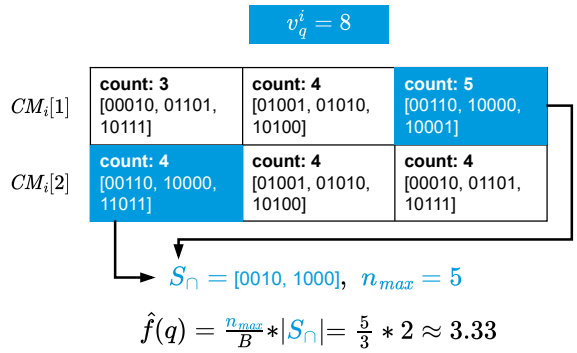

Fig. 1. OmniSketch Overview. An Omnisketch CM_i with depth $d = 2$, width $w = 3$, and min-hash sample size $B = 3$. To estimate the count-aggregate for a predicate value $v_q^i = 8$, we determine the cell index k for each row $CM_i[j], 1 \leq j \leq d$ by hashing $k = h_j(v_q^i) \mod w$, and compute $S_\cap$ by intersecting the contained record id hashes. The estimate $\hat{f}(q)$ is determined by scaling the intersection cardinality with the maximum record count $n_{\max}$ divided by the min-hash sample size B.

ters [18], count-min sketches [16], or HyperLogLog [19] are designed for single-attribute aggregates and do not support multi-attribute predicates. OmniSketch is a novel sketch designed to provide count-aggregates over complex, high-velocity streams with point and range predicates on arbitrary attributes. The sketch provides probabilistic error bounds and a tunable space–accuracy trade-off. Figure 1 gives an overview of the sketch structure and the cardinality estimation process.

Sketch Structure: OmniSketches internally consist of a count-min sketch with d rows, w cells per row and one hash function h_j per row. An OmniSketch CM_i is built for each searchable attribute $a_i \in A$ of a single table. The sketches have a counter $cnt_i[j,k]$ in each cell $CM_i[j,k]$ (analogous to count-min sketches) but also store a sample of record ids $S_i[j,k]$ using K-minwise hashing [17] with a sample size of B. To insert a record R with attribute values r_i and a record id r_0, for each $j \in \{1, \cdots, d\}$ we determine a cell $CM_i[j,k]$ by hashing the attribute with $k = h_j(r_i) \mod w$. We increment its record count $cnt_i[j,k]$, compute a record id hash $g(r_0)$ and add it to the sample $S_i[j,k]$ if $cnt_i[j,k] < B$. Otherwise, if $g(r_0) < max(S_i[j,k])$, we replace $max(S_i[j,k])$ with $g(r_0)$.

Cardinality Estimation: To estimate the cardinality under a given predicate value v_q^i, we hash the value to find d OmniSketch cells, their record counts and record id samples. We set n_{max} to the maximum record count in these matches and compute the sample intersection $S_\cap = \bigcap_{1 \leq j \leq d} S_i[j, h_j(v_q^i)]$. By scaling up the size of the intersection, we compute the cardinality estimate: $\hat{f}(q) = n_{max}/B * |S_\cap|$. Multi-attribute cardinality estimation follows the same logic. We probe each $v_q^i \in V_q$ into CM_i, compute n_{max} from all matching cells, and compute the intersection of $|V_q| * d$ samples.

III. Join Extension

In order to extend the OmniSketch structure for multi-table cardinality estimation, we create OmniSketches for all searchable attributes in each table and assume the availability of a primary key column in each of them, so that the min-hash samples are built on each table's primary keys. This design allows us to retrieve primary key hashes from single-table queries. As it is not possibly by default to probe these hashes into an OmniSketch on a foreign key column, we unify the OmniSketch hashing strategy to allow for interoperability. With the extended OmniSketch, we devise and discuss a strategy to estimate one-to-many joins by probing primary key hashes. We also introduce an alternative strategy for better accuracy and lower latency with secondary OmniSketches. Finally, we define OmniSketch inter-table operations and introduce a join graph traversal algorithm that produces operation sequences to estimate multi-join queries.

A. Sketch Interoperability

In order to make OmniSketches interoperable so that an OmniSketch on a foreign key column can be probed with a primary key hash, we unify the hash functions used in all OmniSketches. To that end, we base all hash functions on a single 64-bit hash function $g'(r_i)$. The min-hash samples contained in the OmniSketch cells use only that hash function. To determine the OmniSketch cells in CM_i for a given value r_i, we compute $g'(r_i)$ and split the result into two 32-bit hashes g'_1, g'_2 [20]. With these partial hashes, we construct a hash function for each row in CM_i as $h_j(r_i) = (g'_1 + j g'_2)$ [21]. This adaption results in two different probing methods. For attribute values we compute the hash $g'(v_q^i)$ and split it to find the cell indices in each row, and for a hashed primary key we skip the hashing and split the hash directly.

B. Join Cardinality Estimation

We introduce two strategies for single join estimation: a universal strategy that estimates joins as sampled set membership predicates and an alternative strategy that builds secondary sketches for additional accuracy and lower latency.

PK Sample Joins: For the PK sample join strategy, we treat joins like set membership predicates (e.g., `a IN (1, 2, 3)`, in which each join key $v_q^i \in V_q^i$ is part of the set to be probed. We can estimate the cardinality of such a query by probing each key individually into an OmniSketch and summating the estimates. In the OmniSketch join estimation case, we do not know the full set of primary keys. Consider a join query such as `SELECT count(*) FROM R, T WHERE R.sid = T.id And T.a = 3` with a primary key constraint on `T.id`. We probe the OmniSketch on `T.a`, producing a cardinality estimate for $'$`T.a = 3`$'$ and a min-hash sample $S_\cap$ on `T.id`. We compute the sampling probability $p = |S_\cap| / \hat{f}('$`T.a = 3`$')$ and probe the samples into the OmniSketch on `R.sid`. Given an n_{max}^i and $S_\cap^i$ for each sample probe, we estimate the cardinality and scale it up with:

$$\hat{f}(q) = \frac{1}{p} \sum_{1 \leq i \leq |S_\cap|} \frac{n_{max}^i}{B} |S_\cap^i|$$

For multi-join support, we compute the union of all samples $S_\cup^i$, and store them with their respective n_{max}^i in a map. That map is used as an intermediate result that can be intersected

with other min-hash samples on the same primary key. By associating each sample with its n^i_{max}, we can compute a single n-way intersection with other min-hash sample unions instead of intersecting each probe result individually with other predicate or join estimation results. This strategy introduces the ability estimates to estimate join cardinalities but also comes with substantial shortcomings. Since the primary key hashes of `T.id` are uncorrelated with the minimal hashes of `R.id`, they act as a random sample. Thus, the upscaling with the sampling probability introduces an assumption that the sample is representative for all qualifying primary keys. However, this may not be the case if the data distribution in the foreign key column is non-uniform. In the worst case, the foreign key column could contain a heavy hitter that is also a qualifying primary key, leading to severe under-estimation. Another drawback of this method is its high compute cost as each individual probe requires a multi-way intersection. Finally, for join graphs in which a fact table joins multiple dimension tables, the estimation relies on the intersection of multiple random primary key samples. As the probability for a match to be in all random samples can become very low (i.e., the product of individual sampling probabilities), these join graphs are likely to run out of witnesses.

Secondary Sketches: For faster join estimation with higher accuracy, we can build *secondary* OmniSketches on dimension tables that map attribute values of the dimension tables directly to their corresponding primary keys of a fact table. For that, we first build the OmniSketches for the fact table. Once these sketches are built, we create the dimension sketches. Instead of inserting their primary keys into the min-hash samples, we probe each primary key into the referencing foreign key column sketches, insert the resulting $S_\cap$ sample into the dimension side samples and increment their record counts by the cardinality estimate. Note that the resulting secondary OmniSketch suffers some information loss (as opposed to a sketch that could be built by probing a hash table on the foreign key column). However, our experiments show that this effect is negligible for cardinality estimation accuracy, and it can be further reduced by increasing the sample size for the primary sketch. Building secondary sketches trades a higher upfront sketch buildup time with a lower estimation latency as all joins and predicates on a fact table and its dimension tables only require a single multi-way intersection. Moreover, this approach resolves the previous method's problem of assuming representativeness as all OmniSketches contain min-hash samples on the same primary key column.

C. Join Graph Traversal

While multi-join cardinality estimation with secondary sketches works analogous to single-table table estimation, using these sketches is not always possible. If the database is not organized in a star-schema, we resort to the default join sampling technique. Estimating arbitrary alpha-acyclic join graphs requires computing a legal sequence of OmniSketch probe operations due to the directional constraint that probing `T.id` primary keys into a `R.sid` OmniSketch yields a min-hash sample of `R.id`. Therefore, all additional filters on `T` must be applied to `T.id` before probing into `R.sid`. An illustrative example for this problem can be found in Figure 2.

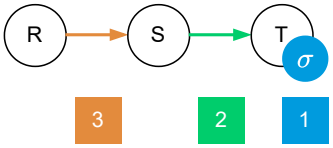

Fig. 2. Probe Sequence Example. Given a join graph with directed edges of foreign key columns pointing to primary key columns and a predicate on `T`. The only legal sequence of OmniSketch probes is (1) the predicate on `T`, (2) probing the resulting primary key hashes into `S`, and (3) probing the resulting `S.id` hashes into `R`.

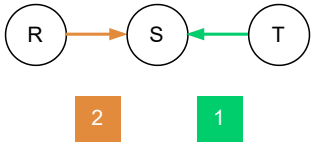

Fig. 3. Expansion Example. Given a join graph, where foreign keys of `R` and foreign keys of `T` join with primary keys of `S`. In that case, neither sequence (1), (2) or (2), (1) is legal for cardinality estimation with regular OmniSketch probes. With a primary key expansion, we can estimate any join first, and then perform a join probe on the remaining relation.

Primary Key Expansions: In certain situations, we must loosen the sequence constraint. Consider the directed join graph from Figure 3, in which foreign key sides point to the primary key side. As `S.id` must be probed into `R.sid` as well as `T.sid`, we either produce a sample on `R.id` that cannot be used to probe `T.sid` or a sample on `T.id` that cannot probe the OmniSketch on `R.sid`. To resolve this stalemate, we introduce a second kind of OmniSketch join estimator: the primary key expansion. For this operation, we perform the PK sample join and intersect the result with any other predicate or join probe results on the foreign key side. However, instead of returning the resulting hashes, we filter the input sample based on whether it has led to matches or not. In our example, we would expand the `S.id` hashes on `R.sid`, resulting in a filtered min-hash sample on `S.id` with a cardinality estimate for $\texttt{R} \bowtie \texttt{S}$. After that, we include `T` in the estimate with a PK sample join of the filtered sample into `T.sid`. Note that also a secondary sketch, if available, can be used either for the expand step or the probe step.

Traversal Algorithm: Algorithm 1 gives an overview of the graph traversal strategy for multi-join cardinality estimation, inspired by the GYO ear removal algorithm [22]. Instead of hypergraphs, it operates on a directed join graph $G \leftarrow (V, E)$ with relations V and joins E, connecting a primary key side $e.pk$ and a foreign key side $e.fk$. We also expect to know the predicates on each $v \in G.V$ and evaluate them with an $EstimatePredicate$ method. The strategy of the algorithm is to find graph "ears" that are only connected to one other relation via their primary key column (line 3-7). We remove those ears by applying them as set membership predicates to the foreign key side (line 9). If a relation has multiple joins on its primary key column (line 12), we use the primary key expansion on any foreign key side that does not have other edges (line 13-17) and

Algorithm 1 Join Graph Traversal Algorithm Overview. The algorithm iteratively merges join graph ears and expands primary keys whenever necessary.

Input: Join Graph $G \leftarrow (V, E)$
Output: Cardinality Estimate

1: **while** $|G.V| > 1$ **do**
2: **for all** $v_i \in G.V$ **do**
3: **if** $\{\exists e \in G.E | v_i = e.fk\}$ **then**
4: **continue**
5: **end if**
6: $E^{v_i} \leftarrow \{e \in G.E | v_i = e.pk\}$
7: **if** $|E^{v_i}| = 1$ **then**
8: $S_{\cap}^{v_i} \leftarrow EstimatePredicates(v_i)$
9: $AddSetPredicate(e_1^{v_i}.fk, S_{\cap}^{v_i})$
10: $G.V \leftarrow G.V \setminus v_i$
11: **else**
12: $E^{v_i} \leftarrow \{e \in E_{v_i} | e.fk \textit{ has exactly one edge}\}$
13: **if** $E^{v_i} = \emptyset$ **then**
14: **continue**
15: **end if**
16: $S_{\cap}^{v_i} \leftarrow EstimatePredicates(v_i)$
17: $AddExpansion(v_i, Expand(e_1^{v_i}.fk, S_{\cap}^{v_i}))$
18: $G.V \leftarrow G.V \setminus e_1^{v_i}.fk$
19: **end if**
20: $G.E \leftarrow G.E \setminus e_1^{v_i}$
21: **end for**
22: **end while**
23: **return** $EstimatePredicates(v_1)$

remove the foreign key side (line 18). The depicted algorithm only provides an overview and does not include processing steps for join graphs containing rings. However, these rings can be processed as well by applying a join predicate to the foreign key side and remove the common edge, if the node is reachable through other edges of the primary key side.

Running Out of Witnesses: If we run out of witnesses during the join graph traversal, we fallback to heuristics to compute a cardinality estimate. For simple predicates, set membership predicates, and sample joins, we set the cardinality estimate for each probe with an empty result to n_{max}/B, which is the minimal cardinality estimate the OmniSketch would have been able to give for a single matching hash. If we run out of witnesses while intersecting multiple join or predicate results, we multiply the associated selectivities. With these heuristics, we pick up the common assumptions of data uniformity and independence but are less likely to do so for plans with large cardinalities as these have a lower probability of running out of witnesses.

IV. Experiments

Our experimental evaluation studies the performance and accuracy of join cardinality estimation with OmniSketches. To this end, we integrate our join graph traversal algorithm with DPsize and enumerate join plans for the SSB-skew [14] benchmark on scale factor 100 and the JOB-light benchmark [8].

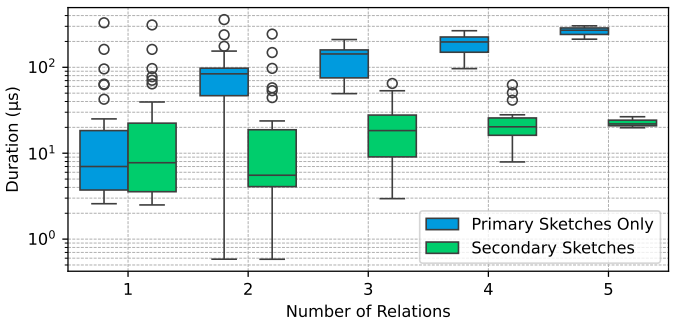

Fig. 4. Cardinality Estimation Latency by Number of Relations in Sub-plan on SSB-skew.

We compare the resulting plans with the default plans from DuckDB [23] v1.2.2. For the SSB-skew dataset, we set the min-hash sample size $B \leftarrow 128$, depth $d \leftarrow 3$, fact table width $w_f \leftarrow 256$, and dimension table width $w_d \leftarrow 32$. We build OmniSketches on each attribute used in predicates and joins, resulting in a total estimated space consumption of about 4.15 MiB for all primary sketches and 4.93 MiB with secondary sketches included. For the JOB-light, we use depth $d \leftarrow 3$, width $w \leftarrow 256$ and $B \leftarrow 256$ across all tables on all attributes used in the benchmark, resulting in a total estimated size of 12.1 MiB. We do not build any secondary sketches for JOB-light, as all joins in the benchmark are performed on the primary key of the `title` table. All benchmarks are executed on a Macbook Pro M3 Max with 36 GiB of main memory.

Estimation Latency: Figure 4 shows the latencies for query sub-plan cardinality estimation on SSB-skew, grouped by the number of relations per sub-plan. We compare the durations for the case, in which we only use primary OmniSketches with the PK sample join strategy with the case with secondary sketches enabled. While the primary-only case shows substantially longer estimation times for 2+ relations, the secondary sketch case only mildly deteriorates in performance for each additional join. The longest measured end-to-end estimation time for a whole query was 1.87 ms in the primary-only case and 0.59 ms with secondary sketches. We omit the individual sub-plan estimation times for JOB-light as they are similar to the primary-only case of SSB-skew. However, the longest end-to-end estimation time for JOB-light was 7.02 ms. We conclude that estimation times are neglibible if secondary sketches can be used. For queries with larger numbers of joins, the estimation overhead may become notable if query processing is cheap and only primary sketches are available.

Estimation Error: We measure the Q-Error to determine the quality of the OmniSketch cardinality estimation and compare it with DuckDB's default cardinality estimator. We define the Q-Error as $CardEst/ActualCard$ to differentiate between under-estimation (Q-Error < 1) and over-estimation (Q-Error > 1). Figure 5 depicts the Q-Errors for all sub-plans enumerated in the SSB-skew benchmark, grouped by the number of relations in the sub-plan. The experiment using only primary OmniSketches vastly over-estimates sub-plans with more than three relations, partly because of hash

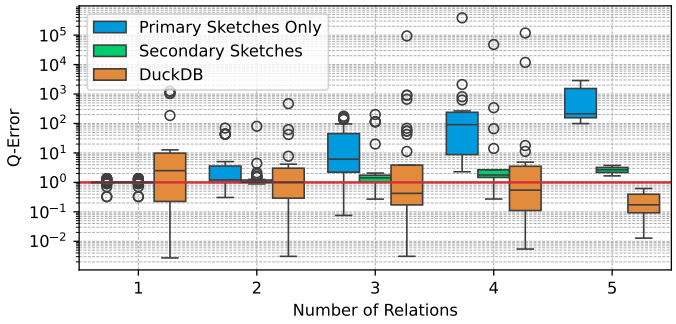

Fig. 5. Q-Error by Number of Relations in Sub-plan on SSB-skew.

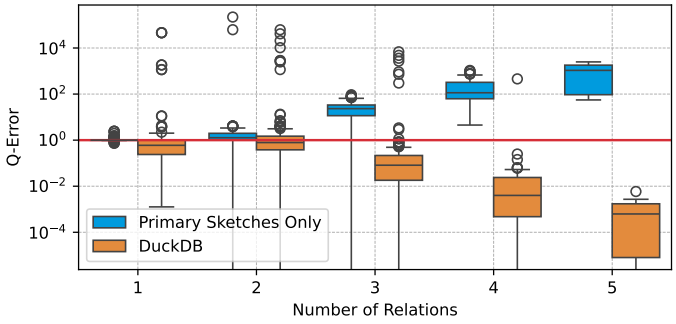

Fig. 6. Q-Error by Number of Relations in Sub-plan on JOB-light.

collisions but also because the fallback heuristics tend to over-estimate sub-plans with negatively correlated predicates. With secondary sketches enabled, the amount of over-estimation can be drastically reduced. The DuckDB cardinality estimator suffers from over- and under-estimation but tends towards under-estimation for three or more relations. Figure 6 shows the Q-Errors for sub-plans from the JOB-light benchmark. Note that we only compare primary OmniSketches in this experiment as all relations in the JOB-light benchmark join on the primary key of the `title` table. Effectively, these join graphs are processed with $n-1$ primary key expansions and one PK sample join for each sub-plan with n relations. In this benchmark, the OmniSketch join estimation strategy systematically over-estimates sub-plans with three or more relations. As our approach frequently runs out of witnesses during primary key expand operations for these sub-plans, it resorts to the heuristics, which tend to over-estimate individual joins and propagate these over-estimations with an increasing number of relations. Nevertheless, the majority of absolute Q-Errors, especially for sub-plans with four or more relations is smaller than DuckDB's absolute Q-Errors.

Plan Quality: Finally, we examine the quality of query plans emitted from DPSize join enumeration with OmniSketches. For that, we run SSB-skew (using secondary sketches) and JOB-light on DuckDB with eight threads. We execute the plans generated from DuckDB's optimizer and our plans in DuckDB and measure end-to-end execution times and the cumulated join cardinalities (C_{out}). For the OmniSketch plan execution times, we also add the individual cardinality estimation latencies. Table I shows a summary of the experiment

TABLE I
END-TO-END EXECUTION TIME AND INTERMEDIATE RESULT IMPROVEMENTS AND REGRESSIONS FOR SSB-SKEW AND JOB-LIGHT OF OUR APPROACH (OMNI) COMPARED TO DUCKDB.

	SSB-skew		JOB-light	
	Omni	**DuckDB**	**Omni**	**DuckDB**
∑ **Execution time** [s]	4.61	7.57	12.46	10.65
Max. improvement	**3.19x**		**2.14x**	
Max. regression	**0.98x**		**0.25x**	
∑ **Intermediates**	367 M	2,620 M	14,932 M	14,467 M
Max. improvement	**1077x**		**3.97x**	
Max. regression	**0.86x**		**0.92x**	

results. For SSB-skew, the total execution time decreases from 9.16 seconds to 6.02 seconds, with a maximum execution time improvement of 3.19x. The OmniSketch approach reduces the total number of intermediates from 2.6 billion to 367 million with a maximum decrease of 1,077x. However, for JOB-light, the experiment shows an increase in total execution time from 10.65 seconds to 12.46 seconds, even though the number of total intermediates is only slightly larger and most queries show a slight decrease of intermediate results. One of the reasons for this deterioration is a single query that has a much larger execution time than all other queries of the benchmark. This query produces 4 % more intermediates with OmniSketches but deteriorates in performance from 4.75 seconds to 5.92 seconds. Another factor comes from a large portion of small queries that have identical query plans and low execution times, where our approach suffers from the overhead of cardinality estimation. These results show that OmniSketch join estimation can be useful with large performance and intermediate result improvements for star-schema workloads, even if the intermediate result decreases do not fully translate to execution time decreases. For other schema shapes, the applicability of OmniSketch may be limited due to frequent loss of witnesses.

V. CONCLUSION

We introduced a new concept for join cardinality estimation with OmniSketches. OmniSketches can be used to estimate the cardinality of multi-join, multi-predicate queries and are especially useful if the data is organized in a star-schema so that we can build secondary sketches on dimension tables. For such a case, our experiments with the SSB-skew benchmark show intermediate result decreases of up to 1,077x and performance improvements of up to 3.19x. In other cases, OmniSketch join estimation may suffer from long cardinality estimation latencies and running out of witnesses. Interesting directions of future work include the applicability for galaxy schemas, cardinality estimation for additional operators, and deriving error bounds for multi-join OmniSketch estimates.

ACKNOWLEDGMENT

We gratefully acknowledge funding from the German Federal Ministry of Research, Technology and Space under the grant BIFOLD25B.

REFERENCES

[1] B. Babcock and S. Chaudhuri, "Towards a robust query optimizer: A principled and practical approach," in *SIGMOD*, 2005, pp. 119–130. [Online]. Available: https://doi.org/10.1145/1066157.1066172

[2] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *PVLDB*, 1990, pp. 314–325. [Online]. Available: http://www.vldb.org/conf/1990/P314.PDF

[3] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979, pp. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[4] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the join order benchmark," *VLDB J.*, vol. 27, no. 5, pp. 643–668, 2018. [Online]. Available: https://doi.org/10.1007/s00778-017-0480-7

[5] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *PVLDB*, vol. 9, no. 3, pp. 204–215, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p204-leis.pdf

[6] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *PVLDB*, vol. 12, no. 11, pp. 1705–1718, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf

[7] Z. Yang, W. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica, "Balsa: Learning a query optimizer without expert demonstrations," in *SIGMOD*, 2022, pp. 931–944. [Online]. Available: https://doi.org/10.1145/3514221.3517885

[8] 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. [Online]. Available: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[9] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep unsupervised cardinality estimation," *PVLDB*, vol. 13, no. 3, pp. 279–292, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol13/p279-yang.pdf

[10] Y. Zhang, Y. Chronis, J. M. Patel, and T. Rekatsinas, "Simple adaptive query processing vs. learned query optimizers: Observations and analysis," *PVLDB*, vol. 16, no. 11, pp. 2962–2975, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p2962-zhang.pdf

[11] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD*, 2000, pp. 261–272. [Online]. Available: https://doi.org/10.1145/342009.335420

[12] S. Babu and P. Bizarro, "Adaptive query processing in the looking glass," in *CIDR*, 2005, pp. 238–249. [Online]. Available: http://cidrdb.org/cidr2005/papers/P20.pdf

[13] A. Deshpande, Z. G. Ives, and V. Raman, "Adaptive query processing," *Found. Trends Databases*, vol. 1, no. 1, pp. 1–140, 2007. [Online]. Available: https://doi.org/10.1561/1900000001

[14] D. Justen, D. Ritter, C. Fraser, A. Lamb, N. Tran, A. Lee, T. Bodner, M. Y. Haddad, S. Zeuch, V. Markl, and M. Boehm, "POLAR: adaptive and non-invasive join order selection via plans of least resistance," *PVLDB*, vol. 17, no. 6, pp. 1350–1363, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1350-justen.pdf

[15] W. R. Punter, O. Papapetrou, and M. N. Garofalakis, "Omnisketch: Efficient multi-dimensional high-velocity stream analytics with arbitrary predicates," *PVLDB*, vol. 17, no. 3, pp. 319–331, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol17/p319-punter.pdf

[16] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005. [Online]. Available: https://doi.org/10.1016/j.jalgor.2003.12.001

[17] R. Pagh, M. Stöckel, and D. P. Woodruff, "Is min-wise hashing optimal for summarizing set intersection?" in *PODS*, 2014, pp. 109–120. [Online]. Available: https://doi.org/10.1145/2594538.2594554

[18] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *CACM*, vol. 13, no. 7, pp. 422–426, 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[19] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," *Discrete Mathematics & Theoretical Computer Science*, 03 2012.

[20] S. Krassovsky, "Modern bloom filters: 22x faster!" 2024, accessed: 2025-08-25. [Online]. Available: https://save-buffer.github.io/bloom_filter.html

[21] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *Algorithms - ESA*, ser. Lecture Notes in Computer Science, vol. 4168, 2006, pp. 456–467. [Online]. Available: https://doi.org/10.1007/11841036_42

[22] C. T. Yu and M. Z. Ozsoyoglu, "An algorithm for tree-query membership of a distributed query," in *COMPSAC*, 1979, pp. 306–312. [Online]. Available: https://doi.org/10.1109/CMPSAC.1979.762509

[23] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *SIGMOD*, 2019, pp. 1981–1984. [Online]. Available: https://doi.org/10.1145/3299869.3320212