Incremental Query Optimizer Statistics in Amazon Redshift

Pascal Pfeil
Amazon Web Services
Munich, Germany
pfeip@amazon.de

Zhe Heng Eng
Amazon Web Services
East Palo Alto, USA
zhee@amazon.com

Magnus Müller
Amazon Web Services
East Palo Alto, USA
magnusmu@amazon.com

George Erickson
Amazon Web Services
Boston, USA
geoee@amazon.com

Roger Kim
Amazon Web Services
East Palo Alto, USA
rogerkm@amazon.com

Mohammed Al-Kateb
Amazon Web Services
Los Angeles, USA
malkateb@amazon.com

Majid Saeedan
Amazon Web Services
East Palo Alto, USA
majidsas@amazon.com

Dominik Horn
Amazon Web Services
Munich, Germany
domhorn@amazon.de

Orestis Polychroniou
Amazon Web Services
New York, USA
orestis@amazon.com

Mengchu Cai
Amazon Web Services
East Palo Alto, USA
mengchu@amazon.com

Tim Kraska*
MIT
Cambridge, USA
kraska@mit.edu

ABSTRACT

Accurate optimizer statistics are fundamental to query and ML-prediction performance in modern database systems, yet maintaining them poses a significant challenge for large-scale data warehouses. Traditional statistics collection relies on full table scans, which become prohibitively expensive as tables grow to billions of rows and beyond. This creates a critical tension: statistics must be kept current to ensure high-quality query plans and accurate ML predictions, but the cost of collecting them grows with data volume. Amazon Redshift, a fully managed, petabyte-scale cloud data warehouse, exemplifies this challenge as customers continuously ingest data from sources such as application logs, IoT telemetry, clickstream data, or use zero-ETL to ingest transactional and operational data. Incremental statistics collection addresses this challenge by updating statistics based solely on modified data, avoiding full table scans while maintaining accuracy. In this paper, we present the design and implementation of incremental statistics collection and maintenance based on data sketches in Amazon Redshift. This technique reduced the fleet-wide weekly compute time spent on collecting statistics for large tables by 40%, while producing statistics that are as accurate as, or even more accurate than before. Our experiences offer practical insights for database practitioners seeking to adopt incremental statistics in production systems at scale, particularly those employing PostgreSQL-style optimizer statistics.

PVLDB Reference Format:
Pascal Pfeil, Zhe Heng Eng, Magnus Müller, George Erickson, Roger Kim, Mohammed Al-Kateb, Majid Saeedan, Dominik Horn, Orestis Polychroniou, Mengchu Cai, and Tim Kraska. Incremental Query Optimizer Statistics in Amazon Redshift. PVLDB, 19(12): 3941 - 3954, 2026.
doi:10.14778/3827998.3828007

*Work performed while at Amazon Web Services.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828007

1 INTRODUCTION

A common pattern in enterprise data warehousing workloads involves periodic data ingestion followed by immediate querying of newly loaded data. However, this creates a challenge for cardinality estimation as new data can be out-of-distribution of the existing statistics, potentially leading to suboptimal query plans and also propagating as noisy inputs to downstream ML predictors, degrading performance on all fronts. Timestamp columns or ID columns, which monotonically increase over time, are especially susceptible to this problem, as newly ingested data often falls outside the range captured by existing statistics. While most commercial systems implement techniques to automatically update statistics, e.g., Amazon Redshift's built-in automatic `ANALYZE`, many existing solutions do so by recomputing them from scratch using a sample or even a full table scan.

Analysis of Amazon Redshift fleet logs revealed a critical insight: the 90th percentile of `ANALYZE` time for large tables (10B+ rows) exceeds 10 hours, amounting to months of cumulative compute time per day across the entire Redshift fleet. Reducing the `ANALYZE` time directly improves performance through fresher optimizer statistics and decreased operational costs.

While data distribution in large tables typically remains stable during append operations, timestamp columns were hypothesized as a particular challenge. We validated this assumption through log analysis for the Redshift fleet, which revealed a 25-fold increase in Q-Error [40] for predicates involving temporal column filters when comparing estimates from stale to fresh statistics. Temporal columns are of type `DATE`, `TIME` or `TIMESTAMP` and constitute 10% of all columns in Amazon Redshift [53]. 45% of scans have a filter containing a predicate on a temporal column. We consider statistics stale when inserted or deleted rows comprise more than 10% of the table's total rows.

While incremental maintenance of statistics has been studied extensively [10, 20, 22, 23, 25, 30, 37, 45, 54], implementing a solution for our case presents unique challenges:

(1) **Amazon Redshift is a large-scale production system.** Tens of thousands of customers cumulatively process exabytes of data every day. Regressing performance because of a new statistics collection mechanism is unacceptable. Additionally, we must

minimize changes to the user-facing catalog interface, as some customers depend on existing statistics.

(2) **Data structures for incremental statistics (sketches) exhibit insufficient insertion throughput in practice.** While novel alternatives exist, they did not meet performance expectations. This necessitated significant engineering effort and novel implementation strategies to achieve the required insertion speeds.

Amazon Redshift's new incremental query optimizer statistics feature relies on data sketches, substantially reducing the cumulative cost of maintaining statistics for large tables. Statistics are refreshed when the percentage of newly inserted rows exceeds a threshold. Our benchmarks demonstrate that incremental ANALYZE outperforms the existing ANALYZE in cumulative execution time after only 3 refresh operations (corresponding to 30% table growth) when using identical refresh thresholds. More importantly, incremental ANALYZE maintains 10× fresher statistics by reducing the refresh threshold from 10% to 1% of newly inserted rows, while still achieving lower cumulative cost after just 34 refresh operations (34% table growth). Real-world deployment data from thousands of production clusters reveals a 40% reduction in weekly time spent collecting statistics for large tables, with this advantage increasing over time as tables continue to grow. We make the following contributions:

- **Sketches-to-statistics conversion.** An algorithm that derives Redshift's single-column statistics from sketches.
- **Sketch selection.** A combination of sketches to generate Redshift's PostgreSQL-style statistics.
- **Incremental ANALYZE framework.** Hierarchical sketch aggregation, MVCC-based delta scans, sampling, and refresh policies.
- **Production deployment.** Rollout to thousands of Redshift clusters, and an analysis of its effect on the fleet.

In the remainder of this paper, we provide background (Section 2), give an overview of incremental query optimizer statistics in Amazon Redshift (Section 3), dive deep into data sketches (Section 4), describe incremental ANALYZE, the framework that builds and maintains incremental statistics (Section 5), evaluate both statistics collection and statistics quality (Section 6), discuss related work (Section 7), and conclude (Section 8).

2 BACKGROUND

This section provides the necessary background on terms and concepts that are relevant to the remainder of the paper, such as query optimizer statistics, Amazon Redshift and data sketches.

2.1 Query Optimizer Statistics in Databases

Database Management Systems rely on query optimizer statistics to generate efficient execution plans. Stale or inaccurate statistics can lead to suboptimal plans that degrade performance by orders of magnitude, making the maintenance of fresh statistics critical for optimal database performance. Beyond plan quality, modern cloud data warehouses also embed ML-based components that consume optimizer-produced estimates as input features. Inaccurate statistics therefore not only affect the query performance but also propagate errors into these learned models, compounding their impact on system performance.

These statistics typically capture both table-level properties such as row counts and column-level distribution properties such as distinct values, null fractions, and histograms, which the optimizer uses to estimate the cost and selectivity of alternative execution plans.

As data is modified, added, or deleted, the underlying statistics must be updated to reflect these changes. Many database systems offer automatic statistics collection mechanisms, though database administrators often need to fine-tune these processes based on their specific workload patterns and performance requirements.

2.2 Amazon Redshift

Amazon Redshift [6] is Amazon's fully managed, petabyte-scale cloud data warehouse service. It operates on a distributed architecture comprising a single leader node and multiple compute nodes. The leader node functions as a centralized coordinator, managing query planning and orchestrating overall system operations. The compute nodes are responsible for executing the distributed query workload, with their data partitioned into units called slices (or partitions), each processing its assigned portion independently. Data in Amazon Redshift resides in Redshift Managed Storage (RMS) backed by Amazon S3 in a compressed columnar format in blocks of 1MB. It is append-only, meaning that deletes are implemented as marking a row as deleted and inserting a new row. To speed up scans, Redshift maintains min/max values per data block (zone maps) used to avoid scanning blocks whose [min,max] range does not overlap with the predicate range of a filter column. The Amazon Redshift Query planner features both a rule-based rewriting engine and a cost-based optimizer.

Amazon Redshift aims to automate as much as possible for its customers. It incorporates autonomics to automate critical tasks such as workload management [51], cluster right-sizing [44], physical design optimization [16], and statistics collection [4]. Several of these components, including Auto-WLM [51] and Redshift's next-generation AI scaling [44], rely on ML-based execution time and memory predictors [44, 51, 55] that featurize physical query plans using optimizer-estimated costs and cardinalities, making them concrete examples of the dependency on fresh statistics described above.

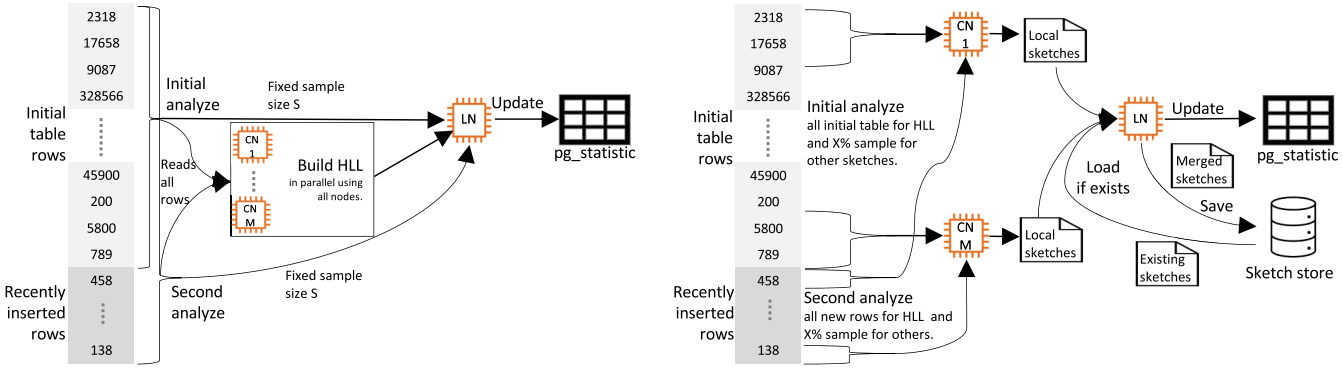
2.3 Optimizer Statistics used in Amazon Redshift

Amazon Redshift uses single-column statistics comparable to those implemented in PostgreSQL.

Basic Statistics. These fundamental statistics can trivially be collected with a single pass over the data and maintained incrementally. They include the table *Row Count*, the *Average Width* of a column's values in bytes, the *Null Fraction*, and the column *Minimum* and *Maximum*, which are also used to reduce the scan set size via zone maps.

Number of Distinct Values (NDV). The optimizer uses NDV primarily for join strategy selection and intermediate result size estimation. Exact NDV computation is resource-intensive for large datasets, so many database systems employ probabilistic counting techniques such as HyperLogLog [20] or sampling strategies [26] to estimate it efficiently.

Most Common Values (MCVs). MCVs capture the most frequently occurring values in a column, enabling the optimizer to account for data skew. When an equality predicate references a value present in the MCV list, the optimizer uses its precise frequency; otherwise, it assumes a uniform distribution over the remaining values. MCVs are



(a) ANALYZE (full table or sample for large tables)

(b) Incremental ANALYZE (sketch-based)

Figure 1: Overview of ANALYZE operations in Amazon Redshift. CN are compute nodes, LN is the leader node.

also used to estimate the selectivity of pattern-matching predicates such as LIKE or regular expressions.

Histogram. A histogram approximates the cumulative distribution function (CDF) of a column by dividing the data into equi-depth buckets. The optimizer uses it to estimate selectivity for range queries and inequality predicates by interpolating between bucket boundaries. To avoid distortion from skew, Amazon Redshift uses a "compressed" histogram that represents the data distribution after removing values listed in the MCVs.

2.4 Statistics Collection in Amazon Redshift

Amazon Redshift collects statistics about its data using the ANALYZE command [3]. Prior to incremental statistics collection, the ANALYZE command would fully scan the table to build a sample (used for average width, null fraction, histogram, and MCVs) and HyperLogLog++ [5, 27] (used for NDV) sketch per column. This sample and the sketch are not persisted after deriving the optimizer statistics from them. Note that the minimum and maximum values are maintained during data insertion and stored on a block level, independently from the optimizer statistics.

Statistic collection is triggered in the current user transaction after a manual ANALYZE command, a CREATE TABLE AS (CTAS) query creates a new table, or a COPY inserts into an empty table. Redshift additionally employs some autonomics to collect statistics on its customers' behalf. Automatic ANALYZE [4] periodically performs analyze operations in the background. To minimize performance impact on user workloads, automatic ANALYZE is only run during periods of low system activity. Redshift tracks the set of predicate columns (those used in a filter, join, or group condition) used during query planning. These predicate columns represent the subset of columns where collecting statistics is worthwhile. Automatic analyze (and user analyze with the PREDICATE COLUMNS option) only analyze predicate columns, and distribution/sort keys.

Per table staleness is tracked by counting the number of rows inserted/updated/deleted since the last analyze. By default, a user triggered ANALYZE skips tables where fewer than 10% of rows have changed since the last analyze. The automatic ANALYZE [4] worker uses staleness to prioritize statistics collection for tables that have changed the most.

2.5 Data Sketches

Data sketches (or sketches for short) implement stochastic algorithms and maintain memory-efficient data structures that can be used to provide approximate answers with theoretically-guaranteed error bounds. Sketches have many desirable properties. One key property of sketches is that they are usually small in size (in the order of kilobytes) and the size grows sub-linearly to the size of the raw data, or not at all. Another key attribute of sketches is that they only need to see each element of the underlying data once, i.e., rescanning existing data is never necessary. Sketches are also agnostic and insensitive to the distribution of the data. All sketches we apply are *mergeable*, i.e., sketches over partitions of data can be combined and the resulting merged sketch has the same accuracy as a single sketch built over the entire data set. Details on merging sketches, in particular for quantile and heavy hitter sketches, are discussed in [1].

Sketches can be tuned using accuracy parameters, most commonly expressed as relative error (ϵ). The relative error ϵ defines the maximum acceptable deviation from the actual result as a fraction of the total dataset size N.

Some sketch algorithms also incorporate an error probability (δ), which represents the likelihood that the sketch's estimate deviates significantly from the true value. These sketches are called "randomized" or "probabilistic". For sketches that use both parameters, they provide guarantees that the approximation will be within a factor of $(1 \pm \epsilon)$ of the dataset size with probability at least $(1 - \delta)$. Users can typically configure these parameters to balance the trade-off between accuracy and memory consumption, as tighter error bounds generally require storing more information. A larger sketch size may also result in lower insertion throughput.

3 INCREMENTAL OPTIMIZER STATISTICS IN AMAZON REDSHIFT OVERVIEW

This section provides a high-level overview of *incremental* ANALYZE, Redshift's framework for maintaining statistics incrementally. Figure 1 illustrates both the sampling-based and incremental ANALYZE.

Amazon Redshift's query optimizer statistics (detailed in Section 2.3) cannot be incrementally maintained directly because their generation requires access to the full dataset or a representative

Table 1: Sketches. Overview of the sketches relevant to this work.

Sketch	Space	Update Time	Deletes	Count	Heavy Hitters	Quantiles	NDV
Count Sketch (CS) [10]	$O(\frac{1}{\epsilon}\log\frac{1}{\delta})$	$O(\frac{1}{\delta})$	✓[1]	✓	✓[5]		
Dyadic Count Sketch (DCS) [54]	$O(\frac{1}{\epsilon}\log^{1.5}U\log^{1.5}(\frac{\log U}{\epsilon}))$	$O(\log U\log(\frac{\log U}{\epsilon}))$	✓[1]	✓	✓[5]	✓	
Greenwald-Khanna (GK) [25]	$O(\frac{1}{\epsilon}\log(\epsilon N))$	$O(\log\frac{1}{\epsilon}+\log\log(\epsilon N))$				✓	
Karnin-Lang-Liberty (KLL) [30]	$O(\frac{1}{\epsilon}\log^2\log(\frac{1}{\epsilon}))$	$O(\frac{1}{\epsilon}), O(\log(\frac{1}{\epsilon}))$ [29]	✓[2]			✓	
HyperLogLog (HLL) [20]	$O(\epsilon^{-2}\log\log n+\log n)$	$O(1)$	✓[3]				✓
Space Saving (SS) [37]	$O(1/\epsilon)$	$O(1/\epsilon)$	✓[4]		✓		

[1] Items that have previously been inserted can be deleted
[2] Can be made to support bounded deletes [58]
[3] Can be made to support deletes by using probabilistic counting [22]
[4] Can be made to support bounded deletes [57]
[5] By using an auxiliary min-heap [10]

sample. ANALYZE requires scanning the entire table to estimate NDV using HLL++ sketches, while computing remaining statistics from a sample (Section 2.4), as computing them from the full table would be prohibitively expensive. This process is illustrated in Figure 1a. We retain this approach for tables of small size.

To address the computational burden for large tables, we introduce an incremental maintenance strategy that processes only newly inserted rows since the last ANALYZE execution. Our approach leverages probabilistic sketches (Section 2.5) to efficiently generate query optimizer statistics, significantly improving both statistics freshness and reducing the computational overhead associated with ANALYZE operations. This is depicted in Figure 1b.

The system operates hierarchically, beginning at the partition (slice) level where local column sketches are created by incorporating all values within each partition. These partition-level sketches are then consolidated at the compute node (CN), which merges sketches from all its assigned partitions into a single one before transmitting it to the leader node (LN). At the final stage, the LN combines all received CN sketches into one unified sketch, generates the corresponding statistics, and persists them in the sketch store.

When table growth reaches a predetermined threshold, the system triggers delta ANALYZE. This process focuses exclusively on newly added rows, generating sketches using the same methodology as full analyze. Delta ANALYZE then merges these newly created sketches with the single existing sketch maintained for each column. This approach ensures that only the initial full analyze bears substantial computational cost, while subsequent delta ANALYZEs efficiently maintain statistical accuracy with minimal resource consumption.

4 INCREMENTAL STATISTICS VIA SKETCHES

To incrementally maintain Amazon Redshift's statistics, we leverage several sketches. Unlike sampling-based approaches, we do not retain a representative sample of the table, as maintaining such samples would require prohibitive storage at Redshift's large scale and raises planning latency concerns. All sketches relevant to this work are summarized in Table 1.

Amazon Redshift's query optimizer statistics (see Section 2.3) consist of basic statistics, NDV, MCVs, and compressed histogram. The basic statistics *Row Count*, *Average Width* and *Null Fraction* can be computed in a single pass over the data using a few counters. Counters from multiple parts of the data can easily be merged. For NDV, we continue estimating them using the mergeable HLL++ sketch.

4.1 Most Common Values (MCVs)

Identifying the most common values in the data requires efficiently tracking heavy hitters while accurately estimating their frequencies. We address this challenge using two complementary sketches: Space Saving (SS) for fast identification of frequent items, and Count Sketch (CS) for accurate frequency estimation. This pairing is inspired by [12], which combines Count Min Sketch (CMS) with a heap; however, we opt for CS because it provides unbiased estimates. SS cheaply guarantees that items above a coarse frequency threshold appear in its candidate set, but its own frequency estimates only honor that same threshold; CS, sized with a much tighter error bound, then refines the candidates' frequencies. Making SS accurate enough to skip the CS refinement would be prohibitively slow on the insertion path.

Space Saving (SS) [37]. The SS sketch is a deterministic algorithm for identifying and tracking frequent items (heavy hitters) within streaming data. The structure maintains k item-count pairs, where k represents the space-accuracy trade-off parameter. For each incoming item, the algorithm either increments its existing counter or, if the item is not currently monitored, replaces the entry with the smallest count. In the latter case, the new item's counter is initialized to the minimum count plus one. This replacement mechanism ensures that frequency estimation errors are bounded by N/k, where N represents the stream size. The sketch provides a strong guarantee: all items whose true frequency exceeds N/k will be captured within the structure, making it particularly effective for heavy hitter detection.

Count Sketch (CS) [10]. CS is a probabilistic data structure designed for frequency estimation. It maintains a two-dimensional array of counters organized into d rows and w columns. The parameter d controls the trade-off between error probability (δ), and memory consumption and insertion performance, while w determines the balance between relative error (ϵ) and memory usage. Initially, all counters are set to zero. For each row, CS employs two hash functions: one maps items to column indices, while the other generates a sign (+1 or −1) for counter updates. During frequency estimation, the algorithm applies these same hash functions to select one counter per row, multiplies each by its corresponding sign, and returns their median value.

Unlike the Count Min Sketch (CMS) [12], which only performs counter increments and returns the minimum counter value during queries, CS can both increment and decrement counters. This key difference allows CS to avoid the inherent positive bias present in CMS estimates, where frequencies can never be underestimated.

Table 2: Sketch Insertion Performance and Size.
Single-threaded for a dataset of 1M uniform random 32-bit integers.

Sketch	Inserts/s	Size (Bytes)
Count Sketch (CS)	111M	81 920
Space Saving (SS)	49M	1 152
Dyadic Count Sketch (DCS)	5M	1 687 544
Greenwald-Khanna (GK)	21M	1 412
Karnin-Lang-Liberty (KLL)	33M	3 976
HyperLogLog (HLL)	237M	2 048
CS + SS + KLL + HLL	15.5M	89 096

4.2 Compressed Histogram

Constructing compressed histograms requires estimating quantiles over data. While Amazon Redshift already employs a Greenwald-Khanna (GK) implementation for `APPROXIMATE PERCENTILE_DISC`, we selected the more modern Karnin Lang Liberty (KLL) sketch for this work. The key concern is long-term stability: GK sketches grow unbounded during merge operations, which becomes problematic as sketches are repeatedly merged throughout a table's lifetime. This is acceptable for a single SQL function invocation but not for our use case, where sketches must be maintained indefinitely. In contrast, KLL maintains bounded size while supporting mergeability with preserved accuracy guarantees.

We also considered dyadic sketches (DCS or DCMS) paired with a heap, which could provide both MCVs and compressed histograms in a unified structure. However, these approaches require a fixed-size universe, precluding support for variable-length types such as strings. Furthermore, their $O(\log U)$ space complexity makes them impractical: for 64-bit integers, they would be up to 64 times larger than a single CS, an unacceptable overhead for our use case. Additionally, their insert throughput was too slow.

Greenwald-Khanna (GK) [25]. The GK sketch is a deterministic data structure designed for computing approximate quantiles. GK maintains a sample of the input values, each containing a value and additional tracking information about minimum and maximum ranks. The sketch ensures that any quantile query can be answered within a guaranteed relative error ϵ while using $O(\frac{1}{\epsilon}\log(\epsilon N))$ space, where N is the dataset size. The structure maintains its accuracy by merging and compressing tuples when certain criteria are met, effectively summarizing the distribution of the data stream. Each tuple tracks a range of possible ranks for its value, and the compression mechanism ensures these ranges remain tight enough to guarantee the desired error bounds. The sketch can *not* be merged into another one while keeping the same error guarantees [1]. There exist merging algorithms for GK that work well in practice, however the sketch will grow unbounded when using them [13]. To get an estimated rank of a given item, we scan the sketch for the position where the item would be put, and sum up all ranks up to that point.

Karnin-Lang-Liberty (KLL) [30]. KLL is a randomized sketch for computing approximate quantiles that offers practical improvements over deterministic approaches such as GK. It maintains a hierarchy of levels containing sorted items, with each successive level employing exponentially decreasing sampling rates. The parameter k controls the space-accuracy trade-off: it is the capacity of the sketch's bottom level and determines the total memory footprint, with the error bound ϵ shrinking as k grows via $k = (C/\epsilon)\sqrt{\log(2/\delta)}$ for a small constant C. New elements enter at the bottom level and progressively migrate upward through a probabilistic compaction mechanism, which preserves items with probability $1/2$ between levels. This design ensures that higher levels maintain increasingly sparse but representative samples of the input stream. A significant optimization by Ivkin et al.[29] improved the worst-case update complexity from $O(1/\epsilon)$ to $O(\log 1/\epsilon)$ while preserving accuracy guarantees. The sketch maintains an error bound of ϵ with probability at least 1-δ, utilizing space complexity of $O(\frac{1}{\epsilon}\text{loglog}(1/\delta))$.

Unlike GK, KLL is fully mergeable, allowing sketches built from distinct subsets of the full dataset to be combined while maintaining the same accuracy guarantees as a sketch built over the entire dataset.

Dyadic Count Sketch (DCS) [54]. The DCS extends the Count Sketch by maintaining multiple sketches organized in a hierarchical, dyadic structure. This arrangement enables it to support frequency estimation as well as quantile estimation both in one sketch. The structure maintains $\log U$ Count Sketches, where U is the size of the domain, with each level corresponding to a different granularity of intervals. At level i, the domain is partitioned into intervals of size 2^i, and a Count Sketch tracks frequencies for each such interval. This dyadic decomposition allows for arbitrary range queries to be answered by combining estimates from at most $2\log U$ dyadic intervals. The last level is just a normal Count Sketch, and can be used for frequency estimation. Like its base structure, the Dyadic Count Sketch provides unbiased estimates, but with additional space complexity proportional to $\log U$ times the size of a single Count Sketch. The trade-off between accuracy parameters (ϵ, δ) and memory usage follows the same principles as the basic Count Sketch for each level.

This principle can also be used with a Count Min Sketch to create a Dyadic Count Min Sketch (DCMS) [54], but the DCS is preferable for the same reason that CS is preferable over CMS.

4.3 Sketch Parameters

We selected sketch parameters to match or outperform the accuracy of Redshift's existing `ANALYZE` implementation in our extensive experiments. For Count Sketch, we use width w of 2048 and depth d of 5, yielding a relative error upper bound ϵ of ~0.01% and theoretical failure probability δ of ~0.67%. Space Saving uses $k = 96$ entries, guaranteeing capture of all elements with frequency exceeding ~1%. CS is sized with a much tighter error bound than SS precisely to *refine* the candidates' frequencies: SS's own frequency estimates inherit the same ~1% threshold, which is too coarse for the optimizer. Tightening SS enough to skip this refinement would require roughly a 100× larger k, which is prohibitively slow when inserting into the SS. For Karnin-Lang-Liberty, we set $k = 200$, achieving an average relative error ϵ of ~1.8%. These configurations consistently deliver statistics quality that meets or exceeds the previous implementation.

4.4 Insert-Optimized Sketch Implementation

Insertion performance dominates because every value inserted into every column passes through the sketches; merge and query are comparatively rare. Reference implementations from other works and the Apache DataSketches library did not meet our throughput

Table 3: Summary of key challenges, design decisions, and rationale.

§	Challenge	Design Decision	Reason
4.4	Existing sketch implementations provide insufficient insertion throughput.	Employ insert-optimized implementation strategies.	Novel sketch designs fail to meet performance expectations in practice, whereas insert-optimized implementations achieve required throughput.
5.3	Must minimize plan changes, avoid performance regressions, and limit modifications to user-facing catalog interface.	Convert sketches into single-column optimizer statistics.	The conservative initial implementation requires no modifications to the query optimizer while maintaining a clear path for future enhancements.
5.5	Data deletion can occur, altering the underlying data distribution. Note that an update is a deletion followed by an insert.	Perform a full data re-scan once a deletion threshold is reached.	While sketches can be extended to support deletions, doing so incurs penalties in both size and insertion speed. Since deletions are infrequent, the chosen approach suffices. If needed, we can transition to per-partition sketch storage to rebuild only affected portions upon deletion.
5.6	Statistics must be kept as fresh as possible.	Refresh statistics asynchronously while reducing the staleness threshold by 10×.	Performing updates on the ingestion path introduces excessive overhead that would measurably degrade ingestion performance.

targets, so in prior work [47] we developed insert-optimized implementations of CS, SS, and KLL that deliver up to 12.3×, 2.0×, and 1.52× speedups, respectively, while preserving the algorithms' theoretical guarantees. We use those implementations unchanged here and refer the reader to [47] for the optimization techniques.

Table 2 reports the single-threaded throughput and size of each sketch at the parameters in Section 4.3, on 1M uniform 32-bit integer inserts, measured on an AWS `m5.12xlarge` instance. GK and DCS are configured to match KLL's error bound. Two takeaways from the table: first, the combination we use (CS + SS + KLL + HLL) achieves 15.5M inserts per second at roughly 90KB; second, DCS is ∼ 20× larger and ∼ 3× slower than this combination at 32 bits and scales worse at 64 bits, which rules it out despite its unified frequency-plus-quantile support (Section 4.2). Hash sharing [47] lets CS reuse HLL's hash computation. Performance on other fixed-size types is comparable to 32-bit integers; skew improves SS and KLL throughput by up to 2× and 1.33×, while CS and HLL are unaffected. Strings are 2–7× slower depending on length, at which point hashing dominates.

5 INCREMENTAL ANALYZE

Incremental ANALYZE is the framework that maintains statistics incrementally for large tables in Amazon Redshift. Figure 1b gives an overview. Table 3 presents the unique challenges that came with implementing a solution for a petabyte-scale data warehouse, and how we addressed them.

5.1 Sketch-based ANALYZE

Our implementation of sketch-based ANALYZE is similar to standard SQL query execution: the process begins with a table scan, followed by insertion into sketches, and concludes with an aggregation step that merges sketches to obtain a single unified sketch per column.

The system employs a hierarchical architecture for sketch construction and aggregation. At the partition level, local column sketches are created by inserting all values within each partition. These partition-level sketches are subsequently consolidated at each compute node (CN), which merges sketches from all assigned partitions into a unified representation before transmitting it to the leader node (LN). Finally, the LN combines all received CN sketches into a single sketch, derives the statistics, and persists them in the sketch store.

We employ sketch-based ANALYZE exclusively for large tables (tens of billions of rows) for two key reasons. First, small tables can be analyzed exactly without sampling, yielding perfectly accurate statistics that are better than sketch-derived approximations, while medium-sized tables incur negligible overhead from repeated scans. Second, sketch-based ANALYZE imposes higher per-row cost than sample-based ANALYZE, which inserts each row into an HLL sketch and cheaply computes the remaining statistics from a small sample. Consequently, the efficiency gains of sketch-based analysis emerge only for large tables, where the amortized ANALYZE cost becomes favorable despite higher per-execution overhead.

A key advantage of sketch-based ANALYZE is its ability to process new data incrementally without affecting the quality of resulting statistics. This property enables us to reduce the staleness threshold for inserts from the existing 10% to just 1% of data change – more frequent statistics refreshes at basically the same cost. We retain the existing catalog logic for determining when to skip ANALYZE, but introduce two modifications: a substantially lower row change threshold (1% instead of 10%) and an additional time-based threshold (maximum staleness of 1 day). These adjustments yield significantly fresher statistics while preserving ANALYZE performance.

To address the impact of deletions on data statistics, we've implemented a new mechanism. When a table reaches a 10% deletion threshold, the next ANALYZE will trigger a full re-scan of the table. This process rebuilds the sketches entirely from the current data, ensuring that the statistics accurately reflect the table's state after significant deletions have occurred. This full re-scan process is computationally expensive. However, we anticipate that such deletions will be occurring infrequently, as discussed in detail in Section 5.5. We store the serialized per-column sketches as a blob in Redshift Managed Storage (RMS) alongside per-table MVCC metadata, enabling transactional durability and isolation.

There are two kinds of sketch-based ANALYZE runs: The first is a full-scan bootstrap sketch-based ANALYZE, which constructs sketches from all currently visible data. This mode is triggered when a table first becomes eligible for sketch-based analysis, when enough deletions have occurred, or for new columns. The second mode is delta sketch-based ANALYZE, which processes only the data inserted since the previous run. For large tables, this delta approach offers significant performance advantages over full scans.

5.2 Delta ANALYZE

The mergeability property of our chosen sketches enables incremental construction of column statistics through a hierarchical merging strategy. During delta ANALYZE, we first construct sketches over the data inserted since the last incremental analyze. These new sketches are built independently per partition, mirroring the approach used in full sketch-based analyze. We then merge the per-partition sketches into a single global sketch representing all newly inserted data. Finally, this global sketch is merged with the existing sketch retrieved from the sketch store, producing an updated sketch that reflects the complete dataset. This incremental approach avoids re-scanning previously analyzed data while maintaining statistical accuracy equivalent to a full analyze. Figure 1b illustrates this process.

Leveraging Redshift's MVCC implementation, delta ANALYZE identifies and scans only newly inserted rows using metadata. Specifically, we record (1) the transaction ID of the last analyze command (`analyze_xid`) and (2) the set of active transaction IDs at that time (`inflight_xids`). During delta ANALYZE, we augment the standard MVCC visibility checks to include only rows satisfying one of two conditions: (1) inserted by a transaction that began after the last analyze committed (`insert_xid > analyze_xid`), or (2) inserted by a transaction that executed concurrently with the last analyze and subsequently committed (`insert_xid < analyze_xid and insert_xid in inflight_xids`). This approach closely parallels Redshift's incremental materialized view refresh mechanism [6], adapted for the insert-only case. Since RMS blocks store `insert_xid` metadata, we perform block-level pruning using the minimum of `inflight_xids` and `analyze_xid`, scanning only blocks containing rows that satisfy the visibility criteria.

5.3 Converting the sketches to Redshift's single-column optimizer statistics

After each sketch-based ANALYZE run, we transform the global sketch for each analyzed column into statistics to be used by the Redshift query optimizer (Section 2.3). Basic statistics are obtained from simple counters, the Most Common Values (MCVs), compressed Histogram, and Number of Distinct Values (NDV) are inferred from the sketches.

Directly querying CS for equality selectivity, KLL for inequality selectivity, and SS's frequent values for predicates such as LIKE or REGEX would be more straightforward, but we instead convert sketches at collection time into the existing optimizer statistics format. Sketches are substantially larger than single-column statistics and cannot all remain resident in memory on the leader node; loading them on demand would add per-query planning latency on top of the cost of deriving estimates from them. Converting at collection time sidesteps both overheads and avoids any changes to the cost-based optimizer, isolating the rollout of incremental ANALYZE from future optimizer-side work. Direct sketch-based estimation remains an option for future work.

Our decision to use sketches, rather than reservoir sampling with HLL sketches [22] or relying solely on block metadata [15], was driven by this cautious approach and the unique requirements of our system. We do not keep a representative sample of the table, as storing samples at Redshift's scale would demand excessive storage and may increase planning latency.

Algorithm 1 Converting the sketches to the single-column statistics the Amazon Redshift optimizer uses for cost-based optimization.

```
function CONVERT(target, rows, cs, ss, kll, hll)
    ndv ← hll.ndv()

    mcvs ← []
    for value,weight ∈ ss do
        mcvs.append((value,weight))
    for value,weight ∈ mcvs do
        weight ← cs.count(value)                    ▷ More accurate.
    SORT(mcvs)                                      ▷ By frequency.
    target ← COMPUTE_NUM_MCV(target,rows,ndv,mcvs)
    mcvs.resize(target)

    hist ← []
    s ← SUM(mcvs)                                   ▷ Sum of MCV frequencies.
    SORT(mcvs)                                      ▷ By value.
    m ← rows − s − 1                                ▷ Max rank.
    cov ← 0                                         ▷ Sum of covered MCV frequencies.
    boundary ← ∅                                    ▷ Current histogram boundary.
    j ← 0                                           ▷ Current MCV index.
    for all i ∈ {0,...,target} do
        br ← (m∗i)/target                           ▷ Base rank.
        pr ← ∞                                      ▷ Previous rank.
        for r ← br+cov;r ≠ pr;r ← br+cov do
            boundary ← kll.quantile(r)
            pr ← r
            while mcvs[j].value <= boundary do
                cov ← cov+mcvs[j].weight
                ++j
        hist ← hist+boundary

    return {mcvs, hist, ndv}
```

The conversion process is outlined in Algorithm 1. This algorithm takes several parameters: *target* (the desired number of MCVs and histogram buckets), the number of *rows*, and *cs*, *ss*, *kll*, *hll* sketches. The MCV count and histogram bucket count are sized with the same parameter to match the existing Redshift behavior.

Number of Distinct Values. The NDV is obtained directly from the HLL++ sketch.

Most Common Values. For MCVs, we first extract candidates from the Space Saving, then refine their estimated frequencies using the Count Sketch, which provides more accurate results. The `compute_num_mcv` function determines the optimal subset of MCVs to retain in the final statistics. It applies several filtering criteria: MCVs must exceed a minimum frequency threshold, represent a significant portion of the table's data, and occur more frequently than the average value (estimated as num_rows/ndistinct). Given the approximate nature of sketch-based statistics, we extended this function by introducing an additional constraint: we now exclude MCVs whose frequency falls below the count sketch's absolute error bound ϵN. The rationale is straightforward: if a candidate's estimated frequency falls below ϵN, it is within the CS error margin, meaning it is indistinguishable from noise. This modification successfully addressed several performance regressions we observed in TPC-DS benchmark queries where the estimated MCV frequencies led to a bad plan.

Histogram. To construct the histogram, we utilize both the MCV list and the KLL sketch. Given that a compressed histogram is required, we must account for and effectively "remove" the MCV frequencies from the estimated distribution. The process of generating a conventional equi-depth histogram involves partitioning the interval $[0,\text{rows}-1]$ into target equally-sized buckets resulting in target+1 ranks. To incorporate the MCVs, we iteratively adjust each boundary by adding the ranks of all MCVs less than or equal to the current boundary to the queried rank, and then re-query the KLL sketch. This process continues until the boundary stabilizes, at which point it is emitted. This approach ensures that the resulting histogram represents the data distribution without the MCVs.

A notable behavioral change exists between the statistics resulting from existing, sample-based `ANALYZE` and sketch-based incremental `ANALYZE`. In the sample-based method, MCVs were removed from the sample before histogram construction. This means that MCVs could never appear as histogram boundaries. With our sketch-based approach, due to the inherent approximation in sketches, MCVs may occasionally appear as histogram boundaries. However, the optimizer's selectivity estimation logic naturally works in this scenario without any modifications.

Worked Example. In the following, we present a worked example for Algorithm CONVERT. Suppose a column with $rows=100$ over the values $[1,50]$, where the non-MCV values have uniform frequency. Let $target=2$, i.e., two MCVs and two histogram buckets. Further, for the sketch inputs, suppose the HLL sketch reports 20 NDV and the SS reports the heavy hitter values $\{10,30,42\}$.

The algorithm first extracts NDV from HLL and heavy hitters from SS, then looks up their frequencies in CS (Line 6). Suppose CS returns $(10,22),(30,20),(42,7)$. All candidates exceed the average frequency $100/20=5$, but COMPUTE_NUM_MCV eliminates the least frequent due to $target=2$, giving $mcvs=[(10,22),(30,20)]$.

Next, we illustrate the histogram construction, starting in Line 11. Recall that we iteratively find equi-depth bucket boundaries over the non-MCV portion of the data by querying the KLL sketch and adjusting for MCV frequencies encountered along the way. To this end, we sort MCVs by value and compute $s=42$, $m=100-42-1=57$. Since $target=2$, we need the $target+1=3$ histogram bucket boundaries ($i\in\{0,1,2\}$). For boundary $i=0$, the base rank is $br=(57\times 0)/2=0$. We query KLL.quantile$(0+0)=1$. Since no MCV has value ≤ 1, the coverage cov remains 0 and the rank does not shift. The boundary stabilizes at 1. Next, for boundary $i=1$, the base rank is $br=(57\times 1)/2=28$, representing the midpoint of the non-MCV distribution. In the first iteration, we query KLL.quantile$(28+0)=15$. The MCV $(10,22)$ has value ≤ 15, so we add its frequency to the coverage: $cov=22$. This shifts the rank upward because those 22 MCV rows occupy space in the KLL but should not count toward our equi-depth target. In the second iteration, we query KLL.quantile$(28+22)=25$. No additional MCVs have value ≤ 25, so cov remains 22 and the rank no longer changes. The boundary stabilizes at 25. Finally, for boundary $i=2$, the base rank is $br=(57\times 2)/2=57$. Note that $cov=22$ carries over from the previous boundary. In the first iteration, we query KLL.quantile$(57+22)=40$. The MCV $(30,20)$ has value ≤ 40, so $cov=22+20=42$. In the second iteration, we query KLL.quantile$(57+42)=50$. No further MCVs are encountered, so the boundary stabilizes at 50.

Finally, in Line 28, a result tuple with $ndv=20$, $mcvs=\{10:22,30:20\}$, and histogram boundaries $[1,25,50]$ is returned.

5.4 Sampling

Our benchmarks (see Section 6.1) reveal that incremental `ANALYZE` can incur up to 6× performance overhead compared to sample-based `ANALYZE`. While the cumulative execution time would eventually favor incremental analysis, we aimed to optimize this further. Sampling for statistics generation is a well-established technique, as inferring statistics from a representative sample typically yields sufficiently accurate results while reducing computational overhead.

We implement random sampling at 25% of the input data to improve performance. However, we apply this sampling selectively: only the SS and KLL sketches process the sampled data, while the HLL and CS continue to process the full dataset. This decision was driven by our previous experience with HLL, where sampling led to poor NDV estimates for sample-based `ANALYZE`. Moreover, since computing the hash for HLL is necessary, we can efficiently utilize the same hash value for the CS with minimal additional overhead.

The conversion algorithm in Algorithm 1 adapts readily to this sampling approach by scaling the ranks when querying the quantile sketch for histogram boundaries.

We chose 25% as a conservative starting point that met our exit criteria on collection time and avoided plan regressions. Lower rates are plausibly viable, but the returns will diminish at some point: HLL and CS continue to process every row, so reducing the sampling rate only lowers the cost of SS and KLL inserts. We leave the exploration of lower sampling rates to future work.

5.5 Delete Support

We deliberately chose sketches that do not support deletes. Although there are methods to enable delete support for KLL, SS, and HLL sketches [22, 57, 58], and some sketches like CS and DCS inherently support deletion of inserted values, these options come with tradeoffs in terms of size, insertion speed, and/or accuracy. Given our focus on insert performance, these compromises were not acceptable.

Moreover, deleting entire rows matching a predicate (e.g., `DELETE FROM t WHERE i = 5;`) in a column store like Redshift typically requires scanning only the `i` column and marking the associated rows as deleted in the `delete_xid` column. However, to update the sketches accurately, we would need to scan all columns to determine which values to delete, adding significant overhead, especially for tables with thousands of columns. Furthermore, if a `VACUUM` command, which reclaims disk space occupied by rows marked for deletion by previous `UPDATE` and `DELETE` operations among other things, occurs before the `ANALYZE` refresh, the deleted column values may have already been physically removed, making it impossible to delete them from the sketches.

Fortunately, deletes are relatively infrequent in Redshift: 86.6% of tables experience no deletes at all, and among those that do, approximately 85% delete only 10 rows daily [53]. Updates are roughly half as common as deletes in the fleet, and the tables where incremental `ANALYZE` is most valuable (tens of billions of rows, append-heavy ingestion pipelines) typically see few modifications. Our approach maintains incrementally updated sketches for newly inserted values, reverting to a full-scan when the cumulative deletes (10%) justify the cost.

5.6 Statistics Refresh Strategy

We initially built sketches on the insert path (between parsing the incoming data and writing it to storage). Updating statistics immediately after insertion provides the freshest possible statistics but adds latency to the critical commit path. We hypothesized that this latency overhead could be concealed within S3 latency. However, upon testing, this proved not to be the case: the overhead for large insert operations was too high. `INSERT`, `COPY`, and `CTAS` queries account for over 50% of query runtime in the Redshift fleet [53], so significant slowdown on this path is unacceptable. Consequently, we decided to extend the existing automatic `ANALYZE` infrastructure for now.

6 EVALUATION

In this section, we will evaluate the sketch-based incremental query optimizer statistics collection in Redshift, and compare to the existing sample-based `ANALYZE`. We emphasize that the primary goal of incremental `ANALYZE` is to reduce the computation spent on statistics collection while keeping statistics quality on par with sample-based `ANALYZE` as this directly translates to fresher statistics and lower operational cost for our customers. Improved query plans arising from the new statistics collection methodology are a welcome consequence but not a design goal on their own. Experiments were performed on a provisioned Redshift cluster with one leader node and ten compute nodes, where each node is of type `ra3.4xlarge` and is equipped with 12 vCPU cores and 96 GiB of RAM.

6.1 Incremental Statistics Collection Performance

To evaluate the execution time of sketch-based incremental `ANALYZE` compared to sample-based `ANALYZE`, we designed an experiment that resembles real-world usage patterns. Our analysis of the Redshift fleet data shows that for tables exceeding 10 billion rows, the median daily growth rate is approximately 1%. This growth rate aligns with the chosen insert threshold that triggers incremental `ANALYZE`, while sample-based `ANALYZE` operates with a higher 10% threshold.

We conduct our experiment using the `lineitem` table from TPC-H at scale factor 10000 (10TB). Starting with an initial size of 10 billion rows, we simulate daily growth by performing 1% incremental inserts until reaching its final size of approximately 60 billion rows, resulting in 182 iterations. We take the median execution time of three warm runs, and compare sample-based `ANALYZE` to incremental `ANALYZE` with no sampling, 50% sampling, and 25% sampling. The results are visualized in Figure 2 which shows the end-to-end execution time.

Our results reveal that without sampling, the initial bootstrapping phase of sketch-based incremental statistics collection takes ~ 6× longer than sample-based `ANALYZE`, ~ 3.9× longer with 50% sampling, and ~ 2.8× longer with 25% sampling. This is due to the higher cost of initial sketch construction from 10 billion rows of data, however, this one-time cost is amortized after a certain number of runs. The subsequent incremental `ANALYZE` operations are up to 23.7× faster than `ANALYZE` without sampling, 35.5× faster with 50% sampling, and 68.4× faster with 25% sampling. Incremental `ANALYZE` beats sample-based `ANALYZE` in cumulative execution time in this configuration after 121 iterations without sampling, 56 iterations with 50% sampling, and 34 with 25% sampling – all while maintaining much fresher statistics (1% vs 10% insert threshold).

We opt to use the performance advantage we get from incremental `ANALYZE` to get 10× fresher statistics. We also investigated the cost improvements we could get if we had the same level of statistic freshness. For this, we set the threshold for incremental `ANALYZE` to the same 10% that sample-based `ANALYZE` uses. In this case, incremental `ANALYZE` beats sample-based `ANALYZE` in cumulative execution time in just 8 iterations without sampling, 5 iterations with 50% sampling, and 3 with 25% sampling. We can also observe that the cumulative cost of running incremental `ANALYZE` more frequently incurs a higher cumulative cost due to the constant overhead associated with each execution.

6.2 Incremental Statistics Accuracy

This section compares the statistics generated by sample-based `ANALYZE` with those produced by incremental `ANALYZE`. This comparison is crucial, as lower-quality statistics may degrade performance relative to the existing solution due to worse query plans or worse ML predictions.

Our analysis utilizes data generated from a modified TPC-H data generator [38] that introduces Zipfian skew with a configurable parameter $z \in 0,1,2,3,4$. Here, $z = 0$ represents no skew, corresponding to the standard TPC-H data with uniform random distribution. Values of $z \in 1,2,3,4$ introduce increasing levels of Zipfian skew, with higher z values indicating more pronounced data skew.

We evaluate the statistics generated by incremental `ANALYZE` against the sample-based `ANALYZE` statistics. This comparison allows us to assess the accuracy of incremental `ANALYZE` across various data distributions, from uniform to highly skewed. By examining how closely the incremental statistics approximate both the sample-based `ANALYZE` and exact statistics, we can gauge the potential impact on query optimization and execution performance. This analysis is particularly important for ensuring that the benefits of incremental statistics collection do not come at the cost of reduced plan quality or query performance regression. After taking a look at the generated statistics, we also check how the statistics influence query execution performance.

6.2.1 Most Common Values (MCVs)

To evaluate MCV identification accuracy, we employ the F-Score metric, a standard measure in classification problems. The F-Score combines Precision (P), which represents the proportion of correctly identified MCVs among all identified values, and Recall (R), which indicates the proportion of actual MCVs successfully detected. This balanced metric is computed as $2*(P*R)/(P+R)$, producing values between 0 and 1, where 1 indicates perfect classification. For assessing the accuracy of MCV frequency estimates, we utilize the relative error ϵ between the actual and estimated frequencies.

The results presented in Figure 3 demonstrate that both sample-based and incremental `ANALYZE` achieve perfect MCV classification on average under uniform distributions (no skew). Performance decreases with increasing data skew, though neither approach's F-Score falls below 0.7. While both methods show comparable classification accuracy, incremental `ANALYZE` exhibits significantly superior performance in frequency estimation, maintaining consistent accuracy even as skew increases. In contrast, sample-based `ANALYZE`'s estimation error grows proportionally with skew. Notably, incremental `ANALYZE`'s empirical error consistently outperforms its theoretical

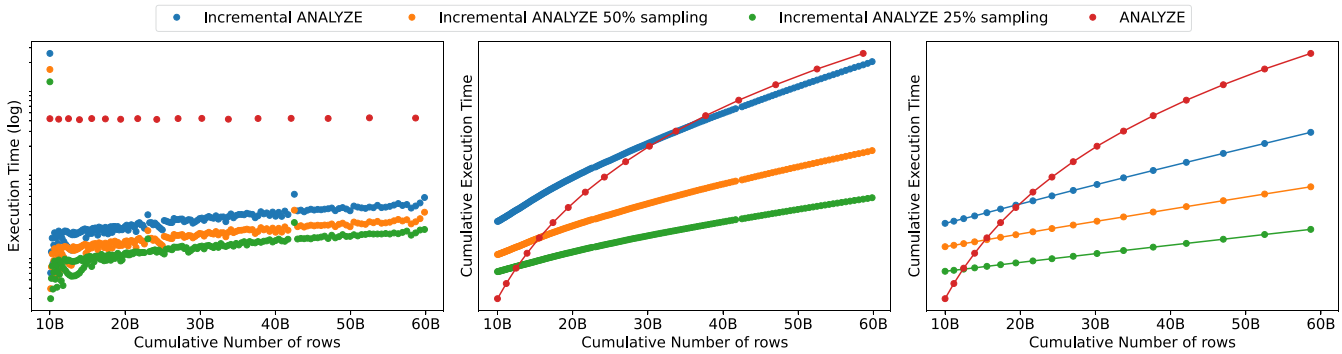


Figure 2: Incremental Statistics Collection Performance. Comparison of end-to-end ANALYZE execution time between sample-based ANALYZE and incremental ANALYZE with different sampling factors on a table starting at 10B rows and increasing by 1% in size each iteration.
Left: Absolute execution time for each ANALYZE run – y axis is log scale.
Center: Cumulative execution time for each ANALYZE configuration for different thresholds (10% vs 1%).
Right: Cumulative execution time for each ANALYZE configuration if the threshold was the same (1%).

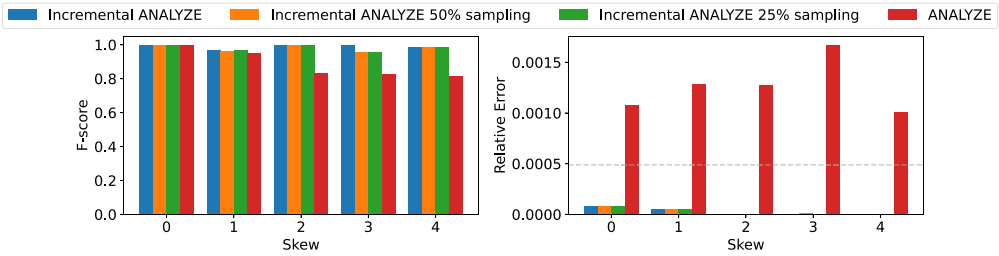


Figure 3: MCV statistics quality. Comparison between sample-based ANALYZE and incremental ANALYZE with different sampling factors and different skew configurations. Left: Average F-score for MCV classification. Right: Average Relative Error for MCV frequencies.

bound (indicated by the gray dashed line) derived from the underlying Count Sketch algorithm, never exceeding this threshold.

The MCV frequency advantage stems from a structural difference. The Count Sketch that refines MCV frequencies is exempt from sampling and processes every row, so its error bound scales with the full N regardless of skew. Sample-based ANALYZE estimates frequencies from a smaller sample, which increasingly under-represents heavy-hitter frequencies as skew grows. Space Saving, while sampled, still sees more rows on large tables than sample-based ANALYZE, improving candidate identification. Consequently, the MCV frequency error curves for the three incremental ANALYZE configurations in Figure 3 coincide, independent of sampling rate.

6.2.2 Histogram

To compare the distributions approximated by perfect and estimated histograms, we employ the Kolmogorov-Smirnov (K-S) test [34]. This non-parametric test calculates the maximum absolute difference between the empirical cumulative distribution functions (ECDFs) of two samples, without assuming any specific underlying probability distribution. We consider the estimated histogram significantly different from the exact histogram when the test's p-value is below the significance level $\alpha = 0.05$.

Additionally, we utilize the Weighted Mean Absolute Percentage Error (WMAPE) to evaluate the accuracy of estimated histogram boundary quantiles. WMAPE, a robust metric that accounts for the magnitude of actual values, is calculated as:

$$WMAPE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{\sum_{i=1}^{n} |y_i|} \times 100\%$$

where y_i and $\hat{y}_i$ represent actual and predicted values, respectively.

The results in Figure 4 demonstrate comparable performance between sample-based and incremental ANALYZE across all skew levels. Both methods maintain an average K-S p-value of ~0.7, with only 3% of columns falling below the $\alpha = 0.05$ threshold (indicated by the grey dashed line). This similarity extends to the K-S statistic values and WMAPE, which remain consistently low across all configurations. Notably, the columns exhibiting significant differences are largely consistent between sampling and incremental ANALYZE, independent of skew level. These findings suggest that incremental ANALYZE produces histograms of equivalent quality to sample-based ANALYZE, regardless of data skew.

6.3 Impact on Query Execution Performance

To evaluate the influence of incremental ANALYZE statistics on query performance, we conduct experiments using both sample-based and incremental ANALYZE statistics. Our evaluation uses TPC-DS and TPC-H benchmarks at scale factor 3000 (3TB). We set the minimum

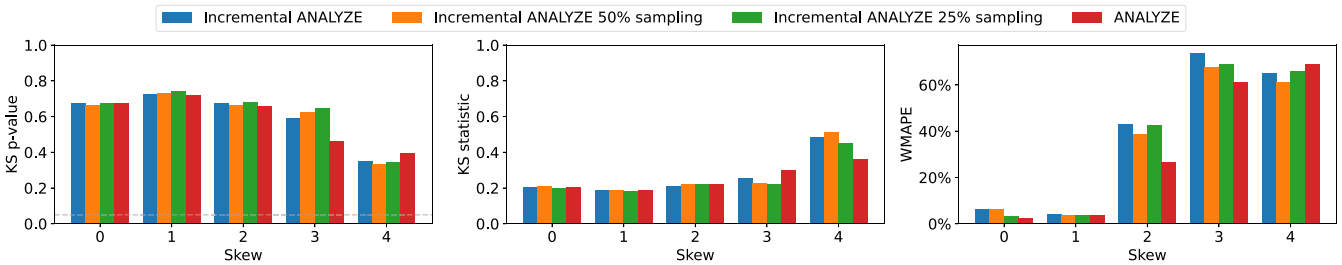


Figure 4: Histogram statistics quality. Comparison between sample-based ANALYZE and incremental ANALYZE with different sampling factors and different skew configurations. Left: Average KS p-value for histogram distribution comparison. Center: Average KS statistic for histogram distribution comparison. Right: Average WMAPE of histogram boundaries' quantiles.

row count threshold for incremental ANALYZE to 1 billion rows to ensure that large tables (`catalog_sales`, `inventory`, `store_sales`, and `web_sales` for TPC-DS and `lineitem`, `orders`, and `partsupp` for TPC-H) qualify for incremental statistics collection. We compare the median execution time of 5 warm runs.

In TPC-DS, we identified 8 queries that exhibited plan changes, resulting in performance variations ranging from a 5% slowdown to a 9% speedup. However, the overall workload execution time remained constant. For TPC-H, we observed plan changes in queries 7 and 9, yielding performance improvements of 14.6% and 54.1% respectively. These enhancements stem from improved join ordering and distribution strategy decisions made by the planner.

While TPC-DS and TPC-H utilize normally distributed data, which may not fully represent real-world scenarios, we extended our evaluation using the modified TPC-H data generator that introduces Zipfian skew [38]. The results demonstrated consistent behavior across all skew factors: No regressions occurred, but Q9 no longer showed improvement. Instead, Q10 exhibited performance gains. The speedups varied by skew parameter: For $z=1$, Q7 improved by 3.4% and Q10 by 27%; for $z=2$, Q7 by 15.3% and Q10 by 24.9%; for $z=3$, Q7 by 22.3% and Q10 by 29.6%; and for $z=4$, Q7 by 6.4% and Q10 by 31.5%. These findings are expected and align with the theoretical properties of data sketches, whose error guarantees remain constant regardless of the underlying data distribution.

Furthermore, we evaluate the impact of sampling by testing both 50% and 25% sampling rates. Consistent with our analysis of statistics quality above, we observe no significant negative effects on query performance when using these sampling rates, suggesting that the reduced statistics collection overhead doesn't come at the cost of plan quality.

6.4 Redshift Fleet

Incremental query optimizer statistics in Amazon Redshift are deployed in production and actively used across thousands of clusters by thousands of customers. Hundreds of thousands of tables qualify for incremental ANALYZE, with hundreds of thousands of executions already completed. Approximately 75% of these executions perform delta scans on modified data, while the remaining 25% perform full scans for bootstrap initialization or periodic full refresh operations as can be seen in Figure 5. Most full scans represent initial bootstrap runs as tables transition to incremental statistics for the first time.

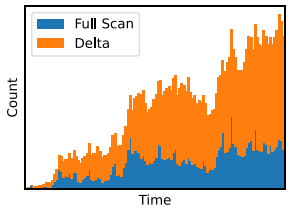


Figure 5: Types of incremental ANALYZE over time while the feature rolls out.

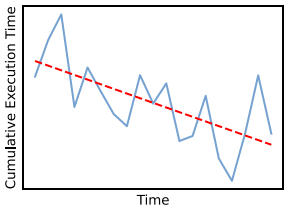


Figure 6: Cumulative ANALYZE execution time on large tables

We expect the ratio to shift further toward delta scans as the feature continues rolling out to additional customers.

Production fleet data demonstrates significant and sustained impact on statistics collection overhead. Comparing the week before initial enablement to the most recent measurement period reveals a 40% reduction in weekly time spent collecting statistics for large tables. Trend analysis over the measurement period shows a consistent downward trajectory (see Figure 6) as more tables adopt incremental statistics and benefit from delta-only collection runs.

Two production cases highlight the practical impact of incremental statistics. The first involves a customer table with hundreds of billions of rows where sample-based ANALYZE consistently aborted due to resource constraints, leaving statistics years out of date and query performance severely degraded. After enabling incremental statistics, the initial bootstrap scan completed successfully in one day, establishing the baseline sketches. Subsequent delta scans now complete in approximately thirty minutes on average, enabling regular statistics updates that were previously impossible. The second case involves a table with tens of billions of rows where the bootstrap incremental ANALYZE run took twice as long as sample-based ANALYZE due to the overhead of initializing sketch data structures. However, this one-time cost quickly amortized: subsequent delta scans demonstrate a 50× speedup compared to sample-based full-scan statistics collection. These cases illustrate the fundamental trade-off of our approach: a modest upfront investment during bootstrap enables dramatic ongoing savings for large, frequently updated tables.

Since deploying to production, we have not received any customer reports of query plan degradation attributable to the feature. This confirms that sketch-based statistics maintain accuracy to preserve plan quality while substantially reducing collection cost.

7 RELATED WORK

Comprehensive overviews of optimizer statistics methods are given in [11, 41]. The earliest approach by Selinger et al. [52] relies on uniformity and independence assumptions. A key aspect of our work is the combination of sketches to derive histograms. In the following, we discuss both in detail and contrast our approach with statistics maintenance in other large-scale cloud data warehouses.

7.1 Sketches

HLL sketches build on two ideas: logarithmic counting, i.e., exploiting the uniformity of 1-bits in hash values, first presented in [21], and stochastic averaging, i.e., scaling up multiple sub-population averages, introduced in [17]. A different approach to NDV estimation are (A)KMV sketches [7, 8], which hash attribute values to [0,1] and scale the k-th smallest hash value h_k to the NDV estimate k/h_k.

Beyond NDV estimation, sketches have been applied to join size estimation; Rusu et al. [49] give a comprehensive overview. Mueller et al. show how to derive join size estimates from HLL sketches [42], notably allowing selection predicates on base tables, and later extend this to AKMV sketches for multiple join size estimation [43]. The CMS [12], discussed in Section 2.5, can also be applied to this problem.

For quantile estimation, all notable approaches rely on a compaction mechanism, starting with Manku et al. [33]. Error guarantees are typically expressed in terms of the difference between the true and estimated rank of a value. Most sketches provide quantile *level* guarantees: for a target quantile level p (e.g., $p = 0.5$ for the median), the estimate $\hat{q}$ corresponds to the true value at some level $p \pm \epsilon$. In contrast, Masson et al. [35] present a sketch with relative quantile *value* guarantees, where $\hat{q}$ differs from q by at most $q\epsilon$. Fernando et al. [19] provide a recent experimental evaluation of many quantile sketches.

7.2 Histograms

Virtually all database systems employ histograms, differing primarily in their bucketing scheme, i.e., how they partition the domain of the attribute(s) they describe. The earliest approach is the equi-width histogram [32]. Later work showed that equi-depth histograms, whose bucket boundaries correspond to quantiles, yield improved selectivity estimates [48]. Many systems store variations of equi-depth histograms, including Redshift's elastic histograms [31]. Further schemes such as max-diff and v-optimal are summarized in [28].

Histograms can also be built over multiple attributes, which complicates bucketing further. Samet [50] gives a detailed overview of multidimensional histograms, and STHoles [9] is a multidimensional histogram designed specifically for selectivity estimation. Similar to sketches, histograms have also been applied to join size estimation, with a recent approach by Wu et al. [56].

7.3 Statistics in Cloud Data Warehouses

Other large-scale cloud data warehouses face the same statistics-freshness problem and take contrasting approaches. Snowflake and BigQuery push most statistics computation and maintenance into the storage layer. Snowflake writes per-column min/max, NDV, and null-presence information into every immutable micro-partition at ingest time [15, 59]. BigQuery writes per-block column statistics (min/max and additional properties used for query optimization) into each Capacitor block and into a system-managed column metadata index that is refreshed incrementally in the background [18, 36, 46]. In both systems, the cost-based optimizer uses this metadata at compile time (for join ordering, build-vs-probe side, and broadcast-vs-shuffle decisions) and complements it with runtime adaptation: Snowflake revisits join strategy once the build-side size is observed [14], and BigQuery reuses actual cardinalities from prior executions of the same query shape via history-based optimizations [24]. Because neither planner consumes full single-column histograms or MCVs (the statistics our pipeline targets), the freshness problem largely dissolves, at the cost of ruling out histogram- and MCV-based estimators.

Azure Synapse's dedicated SQL pool is closer to our setting because it inherits the SQL Server optimizer and depends on single-column histograms, density vectors, and per-table row counts [2, 39]. Statistics are created automatically on first predicate or join use and auto-updated after a material change in the data distribution [39]. Microsoft's own guidance recommends supplementing this with scheduled `UPDATE STATISTICS` during ETL, especially for ascending-key columns whose new values fall outside the existing histogram.

Our approach charts a middle path between these extremes: we retain the richer single-column statistics a cost-based optimizer can consume (MCVs, compressed histogram, NDV), and we make them cheap to maintain by computing them incrementally from mergeable sketches over inserted data, rather than replacing them with storage-level metadata or rebuilding them from full table scans.

8 CONCLUSION AND FUTURE WORK

In this paper, we present Incremental Query Optimizer Statistics in Amazon Redshift. Our approach leverages data sketches to allow scanning only new data and merge sketches computed over new data with existing ones. From these sketches, we derive single-column query optimizer statistics that the Redshift query planner can utilize without modification. These statistics are comparable in quality to those generated by sample-based `ANALYZE` and even improve query runtime for some queries, even though that was not our goal.

Incremental `ANALYZE` outperforms sample-based `ANALYZE` in cumulative execution time while providing 10× fresher statistics with the parameters deployed to the Amazon Redshift fleet. In production, incremental `ANALYZE` has already reduced the weekly cumulative time spent on statistics collection by approximately 40%.

Future work includes collecting statistics for predicate columns on the insert path, using sketches directly in cardinality estimation, and extending incremental statistics to external tables. Supporting partition-level sketches is another direction, both to avoid full table scans for deletions and to enable more accurate cardinality estimates from partitions surviving static elimination. We plan to lower the table size threshold to make more tables eligible for incremental `ANALYZE` and further reduce analysis time. We will also investigate additional sketch algorithm optimizations such as batched inserts.

ACKNOWLEDGMENTS

We extend our thanks to Alexander van Renen, Alexandre Gobeaux, Andreas Kipf, David DeWitt, Davide Pagano, Dominique Sandraz, Martin Milenkoski, Minos Garofalakis, Naresh Chainani, Niranjan Kamat, and Tobias Schmidt, all of whom played a part in making this work possible.

We also thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Transactions on Database Systems (TODS)* 38, 4 (2013), 1–28.

[2] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, et al. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3204–3216.

[3] Amazon Web Services, Inc. or its affiliates. 2025. *ANALYZE*. Retrieved Jan 14, 2025 from https://docs.aws.amazon.com/redshift/latest/dg/r_ANALYZE.html

[4] Amazon Web Services, Inc. or its affiliates. 2025. *Automatic ANALYZE*. Retrieved Jan 14, 2025 from https://docs.aws.amazon.com/redshift/latest/dg/t_Analyzing_tables.html#t_Analyzing_tables-auto-analyze

[5] Amazon Web Services, Inc. or its affiliates. 2025. *HyperLogLog sketches*. Retrieved Jan 14, 2025 from https://docs.aws.amazon.com/redshift/latest/dg/hyperloglog-overview.html

[6] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217. https://doi.org/10.1145/3514221.3526045

[7] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. 2002. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science: 6th International Workshop, RANDOM 2002 Cambridge, MA, USA, September 13–15, 2002 Proceedings 5*. Springer, 1–10.

[8] Kevin Beyer, Rainer Gemulla, Peter J Haas, Berthold Reinwald, and Yannis Sismanis. 2009. Distinct-value synopses for multiset operations. *Commun. ACM* 52, 10 (2009), 87–95.

[9] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 211–222.

[10] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[11] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294.

[12] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[13] Graham Cormode and Ke Yi. 2020. *Small summaries for big data*. Cambridge University Press.

[14] Bjoern Daase and Rudi Leibbrandt. 2024. *Query Acceleration Through Smarter Join Decisions*. Retrieved May 6, 2026 from https://www.snowflake.com/en/engineering-blog/query-acceleration-smarter-join-decisions/

[15] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[16] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, et al. 2024. Automated multidimensional data layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data*. 55–67. https://doi.org/10.1145/3626246.3653379

[17] Marianne Durand and Philippe Flajolet. 2003. Loglog counting of large cardinalities. In *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 605–617.

[18] Pavan Edara and Mosha Pasumansky. 2021. Big metadata: when metadata is big data. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3083–3095.

[19] Lasantha Fernando, Harsh Bindra, and Khuzaima Daudjee. 2023. An Experimental Analysis of Quantile Sketches over Data Streams.. In *EDBT*. 424–436.

[20] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science* Proceedings (2007).

[21] Philippe Flajolet and G Nigel Martin. 1983. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 76–82.

[22] Michael Freitag and Thomas Neumann. 2019. Every row counts: Combining sketches and sampling for accurate group-by result estimates. *ratio* 1 (2019), 1–39.

[23] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. 2002. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems (TODS)* 27, 3 (2002), 261–298.

[24] Google Cloud. 2024. *Use history-based optimizations*. Retrieved May 6, 2026 from https://cloud.google.com/bigquery/docs/history-based-optimizations

[25] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.

[26] Peter J Haas, Jeffrey F Naughton, S Seshadri, and Lynne Stokes. 1995. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, Vol. 95. 311–322.

[27] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*. 683–692.

[28] Yannis Ioannidis. 2003. The history of histograms (abridged). In *Proceedings 2003 VLDB Conference*. Elsevier, 19–30.

[29] Nikita Ivkin, Edo Liberty, Kevin Lang, Zohar Karnin, and Vladimir Braverman. 2019. Streaming quantiles algorithms with small space and update time. *arXiv preprint arXiv:1907.00236* (2019).

[30] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 71–78.

[31] Roger Kim, Mohammed Alkateb, Mengchu Cai, and Ravi Anim. 2024. *Achieve the best price-performance in Amazon Redshift with elastic histograms for selectivity estimation*. Retrieved Jan 14, 2025 from https://aws.amazon.com/blogs/big-data/achieve-the-best-price-performance-in-amazon-redshift-with-elastic-histograms-for-selectivity-estimation/

[32] Robert Philip Kooi. 1980. *The optimization of queries in relational databases*. Case Western Reserve University.

[33] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. 1998. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record* 27, 2 (1998), 426–435.

[34] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[35] Charles Masson, Jee E Rim, and Homin K Lee. 2019. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *arXiv preprint arXiv:1908.10693* (2019).

[36] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.

[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems (TODS)* 31, 3 (2006), 1095–1133.

[38] Microsoft Corporation. 2016. Program for TPC-H Data Generation with Skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430 Accessed: 2025-02-28.

[39] Microsoft Corporation. 2023. *Create and update statistics on tables in Azure Synapse Analytics dedicated SQL pool*. Retrieved Apr 30, 2026 from https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-tables-statistics

[40] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[41] Magnus Müller. 2022. *Selected problems in cardinality estimation*. Universitaet Mannheim (Germany).

[42] Magnus Müller, Daniel Flachs, and Guido Moerkotte. 2021. Memory-efficient key/foreign-key join size estimation via multiplicity and intersection size. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 984–995. https://doi.org/10.1109/ICDE51399.2021.00090

[43] Magnus Müller and Guido Moerkotte. 2022. Translation grids for multi-way join size estimation. In *Advances in Database Technology*, Vol. 25. OpenProceedings, 378–382. https://doi.org/10.48786/edbt.2022.25

[44] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Intelligent scaling in amazon redshift. In *Companion of the 2024 International Conference on Management of Data*. 269–279. https://doi.org/10.1145/3626246.3653394

[45] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[46] Mosha Pasumansky. 2016. *Inside Capacitor, BigQuery's next-generation columnar storage format*. Retrieved May 6, 2026 from https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format

[47] Pascal Pfeil, Dominik Horn, Orestis Polychroniou, George Erickson, Zhe Heng Eng, Mengchu Cai, and Tim Kraska. 2025. Insert-Optimized Implementation of Streaming Data Sketches. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 1–8. https://doi.org/10.1145/3736227.3736238

[48] Gregory Piatetsky-Shapiro and Charles Connell. 1984. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record* 14, 2 (1984), 256–276.

[49] Florin Rusu and Alin Dobra. 2008. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)* 33, 3 (2008), 1–46.

[50] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

[51] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data*. 225–237. https://doi.org/10.1145/3555041.3589677

[52] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[53] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[54] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. 2013. Quantiles over data streams: An experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 737–748.

[55] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query execution time prediction in amazon redshift. In *Companion of the 2024 International Conference on Management of Data*. 280–294. https://doi.org/10.1145/3626246.3653391

[56] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: a new cardinality estimation framework for join queries. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[57] Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. 2021. SpaceSaving±: An Optimal Algorithm for Frequency Estimation and Frequent items in the Bounded Deletion Model. *arXiv preprint arXiv:2112.03462* (2021).

[58] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. 2021. KLL± Approximate Quantile Sketches over Dynamic Datasets. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1215–1227.

[59] Andreas Zimmerer, Damien Dam, Jan Kossmann, Juliane Waack, Ismail Oukid, and Andreas Kipf. 2025. Pruning in Snowflake: Working smarter, not harder. In *Companion of the 2025 International Conference on Management of Data*. 757–770. https://doi.org/10.1145/3722212.3724447