Pruning in Snowflake: Working Smarter, Not Harder

Andreas Zimmerer*
University of Technology Nuremberg
Nuremberg, Germany
andreas.zimmerer@utn.de

Damien Dam
Snowflake Inc.
Berlin, Germany
damien.dam@snowflake.com

Jan Kossmann
Snowflake Inc.
Berlin, Germany
jan.kossmann@snowflake.com

Juliane Waack
Snowflake Inc.
Berlin, Germany
juliane.waack@snowflake.com

Ismail Oukid
Snowflake Inc.
Berlin, Germany
ismail.oukid@snowflake.com

Andreas Kipf
University of Technology Nuremberg
Nuremberg, Germany
andreas.kipf@utn.de

ABSTRACT

Modern cloud-based data analytics systems must efficiently process petabytes of data residing on cloud storage. A key optimization technique in state-of-the-art systems like Snowflake is partition pruning—skipping chunks of data that do not contain relevant information for computing query results.

While partition pruning based on query predicates is a well-established technique, we present new pruning techniques that extend the scope of partition pruning to `LIMIT`, top-k, and `JOIN` operations, significantly expanding the opportunities for pruning across diverse query types. We detail the implementation of each method and examine their impact on real-world workloads.

Our analysis of Snowflake's production workloads reveals that real-world analytical queries exhibit much higher selectivity than commonly assumed, yielding effective partition pruning and highlighting the need for more realistic benchmarks. We show that we can harness high selectivity by utilizing min/max metadata available in modern data analytics systems and data lake formats like Apache Iceberg, reducing the number of processed micro-partitions by 99.4% across the Snowflake data platform.

CCS CONCEPTS

• **Information systems** → **Database query processing**; **Query optimization**; **DBMS engine architectures**; **Cloud based storage**; *Top-k retrieval in databases.*

KEYWORDS

data warehouses; data skipping; partition pruning; top-k queries; LIMIT pruning; join pruning; analytical query processing

ACM Reference Format:
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 (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724447

*Work done while at Snowflake Inc.

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

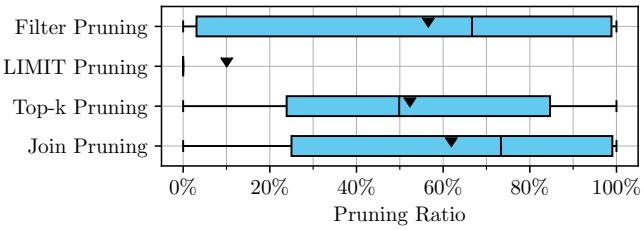


Figure 1: Pruning ratios of different pruning techniques for eligible queries in Snowflake. ▼ indicates mean values. Based on representative samples across all customers from Nov 5th to Nov 7th 2024.

1 INTRODUCTION

The fastest way of processing data is to not process it at all. Excluding chunks of data, i.e., *pruning* them, is indispensable for query performance. In cloud-based data analytics systems, pruning becomes even more important: such systems decouple compute and storage. Hence, pruning serves not only to avoid processing and loading data from disk but is also essential in avoiding costly network I/O. Given that scan and filter operations typically account for more than 50% of processing time [50], effective data pruning becomes a key driver for query performance in cloud-based data analytics systems. For instance, Google's PowerDrill attributes its performance to skipping 92.41% of data on average [29]. Snowflake, in turn, has achieved a staggering 99.4% of data being skipped across all customer workloads[1]. Though not directly comparable[2], both figures underscore the importance of data skipping.

Data in cloud-based OLAP systems is typically broken down into implicit, horizontal *micro-partitions* that can be accessed independently by the query engine. Using partition-level metadata, an engine can skip over micro-partitions that do not contain any qualifying tuples. Besides simple min-max values per column (zone maps [27] or SMAs [39]), there is a vast array of techniques that improve data skipping. These include secondary index structures [35], approximate set-membership data structures (e.g., Bloom-filters [10, 36], Cuckoo-filters [26], or Xor-filters [17, 28]), and optimizations for joins through sideways information passing [37, 40].

[1]Measured from Nov 18th to Nov 24th 2024.
[2]PowerDrill performs an initial pruning pass on a table's clustering-key, which is not reflected in this number.

In the context of cloud-based systems with decoupled compute and storage architectures, the high latency and costs associated with network I/O amplify the importance of accurate, aggressive, and efficient data pruning. Skipping irrelevant data early in the query life-cycle prevents not just unnecessary data loads from remote storage when scanning data, but also reduces communication overhead between nodes when exchanging information on what data needs to be scanned. Besides its highly optimized proprietary format, Snowflake also supports the Apache Iceberg [2] open table format with the Apache Parquet file format [3]. All techniques presented in this paper are also applicable to Iceberg tables in Snowflake which will be further discussed in Section 8.1.

Despite significant research in improving pruning through auxiliary metadata and indexing techniques, these approaches often operate under idealized assumptions. Many fail to account for complex, real-world scenarios. For instance, Ives and Taylor evaluated their *sideways information passing* technique [32] using a modified TPC-H benchmark. However, as we demonstrate in Section 8.3, TPC-H exhibits significantly different pruning behavior compared to real-world workloads.

The objective of this paper is to provide insights into the pruning process and its efficacy on real-world workloads in large cloud-based data systems, such as Snowflake. We examine how the decoupled compute and storage architecture influences pruning design, evaluate various pruning techniques in real-world settings, and propose novel enhancements to improve query performance. Figure 1 illustrates the pruning ratios of various specialized techniques in Snowflake, demonstrating their substantial impact. Although few queries benefit from `LIMIT` pruning, its impact on those is significant, as shown by the high mean relative to the low median.

It is worth noting that, regardless of the implemented pruning techniques, the number of data partitions that can be skipped primarily depends on how data is distributed among micro-partitions, e.g., via sorting or clustering, and on the concrete workload. Data layout optimizations and adaptive partitioning strategies based on workload characteristics for better pruning (e.g., [7, 9, 14, 22, 49, 52, 53]) are topics on their own and beyond the scope of this paper.

Our contributions are as follows:

- We evaluate and quantify the impact of four different pruning techniques: filter pruning, `LIMIT` pruning, top-k pruning, and `JOIN` pruning.
- We propose a novel technique for pruning LIMIT queries, both with and without predicates.
- We demonstrate how pruning techniques traditionally used in search engines for top-k queries can be adapted and integrated into SQL engines, yielding significant performance improvements. Additionally, we propose optimizations to further enhance the effectiveness of these techniques.
- Unlike previous works that focus predominantly on single pruning techniques in isolation, we illustrate their combined effectiveness through a guiding example and showcase how they complement each other.

The paper is structured as follows: We first provide an architectural overview of the Snowflake Data Platform in Section 2, outlining how pruning integrates into and is influenced by the platform's overarching design, as detailed in Section 2.1. Next, we examine several pruning techniques implemented in Snowflake, assessing their effectiveness and impact. Specifically, Section 3 demonstrates how filter pruning effectively reduces the number of micro-partitions processed for queries with predicates. In Section 4, we explore a specialized pruning technique tailored to `LIMIT` queries, leveraging extensions to filter pruning to extract meaningful table structure insights. For *top-k* queries, Section 5 introduces a pruning strategy inspired by concepts from the Information Retrieval (IR) domain, achieving substantial performance improvements. Section 6 focuses on partition pruning for `JOIN` queries. Finally, in Section 8, we discuss pruning in the context of data lake environments and critically examine the limitations of synthetic benchmarks such as TPC-H in accurately capturing the impact of partition pruning.

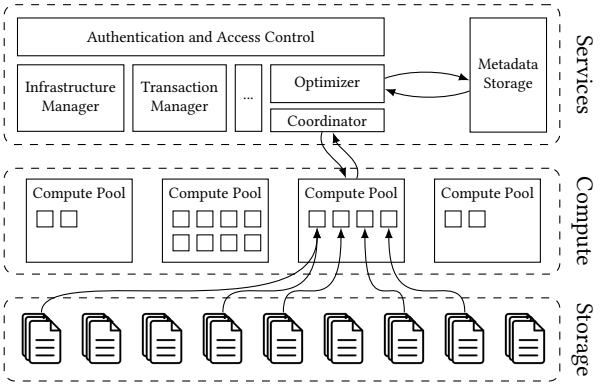


Figure 2: Architecture of the Snowflake Data Platform.

2 ARCHITECTURE OF SNOWFLAKE

Snowflake is a cloud-based data platform, offering, among other services, scalable analytics across multiple cloud providers as a service for enterprise customers. Its foundational design, as detailed in the original Snowflake paper [16], introduced the *multi-cluster, shared data architecture*: storage is decoupled from a pool of compute nodes organized in a shared-nothing setup. Together with a dedicated metadata store, this architecture has, more than a decade later, stood the test of time and has influenced the structure of other cloud-based data systems. At Snowflake, this design shapes the development of essential components, such as pruning.

In the following paragraphs, we will briefly give an overview of the main components of the Snowflake Data Platform, which are also illustrated in Figure 2.

Data Storage. Snowflake leverages cloud object storage, e.g., AWS S3 [1], Azure Blob Storage [4], or GCP Cloud Storage [5], as its disaggregated storage layer. Tables are implicitly horizontally partitioned at row boundaries at different granularities depending on the table format. Regular Snowflake tables are split into *micro-partitions*, typically containing between 50 and 500 MB of uncompressed data. Tables in the Apache Iceberg table format backed by Apache Parquet files follow a similar pattern, but the partition sizes depend on the writer of the file. Both adhere to a PAX-style storage layout [8]. Additional information about Iceberg tables in Snowflake is discussed in Section 8.1.

Virtual Warehouses. Virtual warehouses provide the compute resources for query processing and are ephemeral, user-specific resources. Each virtual warehouse comprises a fleet of up to several hundred compute nodes organized in a shared-nothing architecture, facilitating highly parallel query execution. When a query is scheduled for execution, the warehouse receives an optimized query plan that contains *scan sets*, i.e., a serialized list of micro-partition identifiers to be processed as part of the query. Note that a scan set might have already been subject to partition pruning at query compilation time, and a compute node in the virtual warehouse might further prune the scan set before loading data as some pruning techniques can only be executed at query runtime.

Cloud Services. Snowflake's control plane operates as a highly multi-tenant service layer. It processes incoming queries, performs compilation and optimization steps and schedules queries for execution on virtual warehouses. Part of Snowflake's cloud services is a dedicated metadata service: a scalable, transactional key-value store that manages metadata for partitions stored in the data storage layer. This design enables fast, independent access to metadata without loading the actual data, which is essential for fast and effective partition pruning during query optimization. Additionally, the control plane handles access control, transaction management, encryption keys, administration of virtual warehouses, and so forth.

2.1 Pruning at Snowflake

Pruning, also known as data skipping [48], is a common technique in data analytics systems to minimize the volume of data being processed. In Snowflake, pruning is primarily performed at the level of *micro-partitions*; for tables in Apache Parquet format, it occurs at the file, row-group, and page levels. However, for simplicity we use the term *(micro-)partition* interchangeably throughout this paper. Snowflake performs min/max pruning based on lightweight metadata maintained for each micro-partition, similar to zone-maps [27] or small-materialized-aggregates (SMAs) [39]. By comparing this metadata against the query's predicates, the database engine can efficiently identify micro-partitions that do not contain relevant data, allowing them to be skipped entirely.

Consider, for instance, two micro-partitions f_1 and f_2 containing the value ranges `0..9` and `10..19`, respectively. If a query contains the predicate `WHERE x >= 15`, the query engine can skip f_1 because its maximum value, 9, is less than 15 and therefore does not contain any matching data. Conversely, f_2 must be processed because the predicate's value 15 falls within its range of `10..19`.

Pruning is designed to guarantee no false negatives, meaning it guarantees correctness by ensuring all relevant data is included in the query result. It might, however, still produce false positives because sometimes a micro-partition deemed relevant does actually not contain matching data. At Snowflake, partition pruning is performed in various places depending on the query type, but it can generally be categorized into two phases: *compile-time* pruning and *runtime* pruning.

Compile-time pruning happens during query compilation and is therefore driven by the cloud services layer. Performing pruning at such an early stage in the query life cycle has the advantage that the pruning result can be leveraged during query compilation for further optimizations, potentially resulting in a better query plan. These optimizations can range from more accurate cardinality estimates—and therefore better join ordering—to the elimination of entire sub-trees in the query plan. To allow pruning during compilation, fast access to micro-partition metadata is essential.

Runtime pruning happens during query execution on the execution layer in virtual warehouses. This implies that pruning algorithms need to be designed for high degrees of parallelism and with minimal inter-node communication. Despite these constraints, runtime pruning offers two main advantages: First, pruning is performed in parallel on multiple machines, which can make it beneficial to dynamically push compile-time pruning to a virtual warehouse if pruning itself is time-consuming. Second, while compile-time pruning is limited to *static* pruning, pruning at runtime can incorporate information that becomes available only during query execution, allowing data-dependent pruning techniques like top-k pruning (see Section 5) or join pruning (see Section 6). Such pruning techniques require a flexible execution engine capable of passing information both horizontally and vertically.

Summary. In conclusion, effective pruning has multiple benefits:

(1) Faster table scans with significantly reduced data needing to be loaded over network or from disk.
(2) Enhanced cardinality estimation, leading to better join ordering and overall improved query plan quality.
(3) More accurate work estimation, enhancing the precision of query scheduling.
(4) Due to Snowflake's architecture, effective pruning results in smaller *scan sets*, reducing the (de)serialization work and resulting in less data transported over the network, especially for large tables.

In the following sections, we will examine four types of pruning techniques in the Snowflake Data Platform and analyze their impact on customer workloads.

3 FILTER PRUNING

Min-max pruning for filter predicates represents the traditional approach to pruning in data analytics systems [48]. The concept is simple: Using the query's predicates, the query engine attempts to deduce whether a micro-partition might contain relevant data based on the partition's metadata. If it can conclusively confirm that a partition does not contain any rows that satisfy the predicates, the partition's identifier is removed from the *scan set*. Despite its widespread adoption and a general agreement of its effectiveness, we found no studies that measure the impact of this technique on real-world workloads. We present such an analysis in Section 3.3.

Snowflake performs filter pruning both during compilation and query runtime, carefully optimizing the balance between these phases to achieve best overall query execution times. Partition pruning at compilation time offers the advantage that the pruning result can be utilized for further plan optimizations, such as improved cardinality estimates for join ordering, query sub-tree elimination, constant folding, and other cost-based decisions, potentially leading to improved query plans. Filter pruning at compile time is designed as a dynamic multi-step process: as new filters are identified, the pruning results are incrementally refined.

As an illustrative example, we use the scenario of the International Union for Conservation of Nature (IUCN)[3] identifying suitable locations for an animal observation post on a mountain ridge. Beyond this data analysis case, similar examples can be envisioned in areas such as cybersecurity or process monitoring etc.

Initially, a data analyst may aim to list all potential locations, specifically those situated along an existing trail on a mountain ridge above a specified altitude. Since the altitude could be provided in either feet or meters, it must first be converted to a consistent unit. A possible query for this scenario could look like the following:

```
SELECT * FROM trails
WHERE IF(unit='feet', altit * 0.3048, altit) > 1500
  AND name LIKE 'Marked-%-Ridge';
```

Based on this example, we examine two key aspects: pruning complex expressions and balancing compile-time and runtime pruning.

3.1 Pruning Complex Expressions

Pruning micro-partitions based on simple base-value predicates is not enough for most queries. In reality, many predicates make use of unary—or even *n*-ary functions involving multiple columns—and perform non-trivial computations with their parameters as we can also see in our example query. To further illustrate the scenario, we consider the following available metadata:

Column	Min	Max
unit	"feet"	"meters"
altit	934	7674
name	"Basecamp-..."	"Unmarked-..."

How can the predicate in our example be used to efficiently prune micro-partitions when only basic min/max metadata is available? To address this, consider each component of the predicate individually.

Deriving Min/Max Ranges. For `(altit * 0.3048)`, the original range of the `altit` column is scaled, resulting in a transformed range of around `(min=284.68, max=2339.04)`. For the `IF(...)`, where it is not always possible to determine which values in `unit` are equal to `'feet'`, a conservative approach must be adopted. Here, this means that the resulting min/max range is extended to encompass the min/max ranges of both sub-expressions, yielding `(min=284.68, max=7674)`. If there are micro-partitions where the min/max metadata indicates that either none or all values of `unit` are equal to `'feet'`, the ranges can be adjusted accordingly. The comparison `>1500` partially overlaps with the range `(min=284.68, max=7674)`, meaning the first boolean expression evaluates to `true`. This indicates the micro-partition *might* contain rows satisfying the predicate. For effective pruning, every function must provide a mechanism to derive transformed min/max ranges from its input.

Imprecise Filter Rewrites. For the second filter `(name LIKE ('Marked-%-Ridge'))`, we employ a technique we refer to as *imprecise filter rewrite*. While predicates can only be rewritten to semantically equivalent expressions to ensure correctness for query evaluation, pruning allows for a different approach: predicates can be *widened* to facilitate more coarse-grained pruning. This enables predicates that are otherwise not directly applicable for pruning to be transformed into less precise forms that support pruning, as demonstrated in this case. For `'Marked-%-Ridge'`, the predicate can be widened to `STARTSWITH('Marked-')`, effectively relaxing the constraint that the string has to end with `'-Ridge'`. Pruning for `STARTSWITH` is then performed by checking whether the string falls within the min/max range of the column.
The resulting expression used for pruning micro-partitions is:

$$
\begin{aligned}
&\texttt{name}_{min} \texttt{ <= 'Marked-' <= name}_{max} \wedge (\\
&\quad ((\texttt{altit}_{min} = \texttt{'feet'} \wedge \texttt{altit}_{max} = \texttt{'feet'}) \rightarrow \\
&\qquad \texttt{altit}_{max} * 0.3048 > 1500) \vee \\
&\quad ((\texttt{altit}_{max} < \texttt{'feet'} \vee \texttt{altit}_{min} > \texttt{'feet'}) \rightarrow \\
&\qquad \texttt{altit}_{max} > 1500) \vee \\
&\quad ((\texttt{altit}_{max} \geq \texttt{'feet'} \wedge \texttt{altit}_{min} \leq \texttt{'feet'}) \rightarrow \\
&\qquad \max(\texttt{altit}_{max} * 0.3048, \texttt{altit}_{max}) > 1500))
\end{aligned}
$$

Evaluating this expression against the provided metadata in the example indicates that the micro-partition should not be pruned.

3.2 Balancing Compile-Time and Runtime Filter Pruning

As previously noted, compile-time pruning introduces additional overhead to the compilation process, which can become prohibitively expensive for queries on extremely large tables with complex and deeply nested predicates. To maintain fast query execution times, Snowflake employs two key strategies: (i) reordering the evaluation sequence of filters during partition pruning to prioritize fast and effective filters and (ii) halting pruning for filters that prove ineffective or slow during compile time, with the option of deferring their execution to the highly parallelized virtual warehouses and hence pushing their evaluation to query execution time.
Both approaches rely on monitoring the partition pruning ratio for each filter as well as the average time required to evaluate the filtering predicate on a micro-partition.

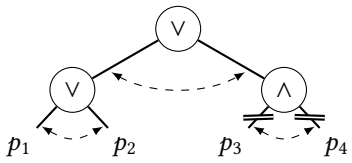


Figure 3: Exemplary pruning-tree with predicates p_i for the expression $(p_1 \vee p_2) \vee (p_3 \wedge p_4)$ showing re-ordering and cutoff options. Cutoff can only be performed below an $\wedge$.

Filter Reordering. For (i), when pruning involves multiple filter predicates, these predicates are connected by boolean operators, forming a tree-like structure. An example for this is depicted in Figure 3. The filter predicates serve as the leaves, while the boolean operators ($\wedge$ or $\vee$) act as the inner nodes. The order in which the direct children of $\wedge$ and $\vee$ nodes are evaluated can be rearranged freely (dashed arrows in Figure 3). This flexibility allows prioritizing fast and highly selective filters for $\wedge$ expressions, as they quickly reduce the number of micro-partitions. Conversely, for $\vee$ expressions, it is better to prioritize fast filters with low selectivity.

[3] https://iucn.org/

An initial pruning order can be established heuristically. As pruning progresses, Snowflake tracks the pruning ratio and evaluation time for each node in the pruning tree, making local adjustments to improve efficiency. Tracking these metrics incurs little overhead and gradually performing local decisions ensures the complexity of reordering does not explode. Over time, this adaptive, localized approach may lead to a more optimal global execution order.

Filter Pruning Cutoff. If reordering filters during pruning does still not meet the desired performance requirements, strategy (ii), i.e., halting pruning for slow or ineffective filters, is applied. We call this "filter pruning cutoff" and apply this to filters that are either slow, or ineffective, or both. Similar to reordering, Snowflake monitors execution speed and pruning ratio for each node in the pruning-tree and repeatedly check for nodes violating the performance requirements. A conservative, but simple, metric for locally checking if a pruning node is slow or ineffective is to model and compare two scenarios: We can either continue using the filter for pruning and extrapolate its pruning ratio to the remaining *scan set*, considering its computational overhead. Or we can estimate the impact on overall query performance if we would stop pruning with that filter and let the parallel execution engine process all remaining micro-partitions. The scenario yielding the better prediction regarding overall query execution time should be chosen. However, it is important to note that this simple metric does not account for specific query characteristics; for instance, removing a pruner might prevent other optimizations from being applied, e.g., sub-tree elimination or constant-folding.

Additionally, care must be taken when removing pruning filters from the predicate tree to avoid incorrect results. When deciding to stop pruning for a filter, a conservative approach should be used, i.e., we should assume that every micro-partition passes that filter from the point on where we stop evaluating it. If a filter is part of an $\wedge$-expression, it can be safely removed since its role is to further narrow the scan set, though this may reduce the overall pruning effectiveness of the $\wedge$-expression. Henceforth, pruning for filters $p3$ and $p4$ in Figure 3 can be stopped if they prove ineffective. In contrast, if the filter is part of an $\vee$-expression, removing that filter—thus assuming that every subsequent micro-partition *might* contain qualifying rows—would effectively stop further pruning. The removed branch from the $\vee$-expression would mark every micro-partition as potentially containing relevant rows and hence the $\vee$-expression itself becomes ineffective. The $\vee$-expression itself may even be removed instead, which might recursively affect higher levels, potentially removing all pruning filters. For instance, if $p1$ in Figure 3 is removed, the left sub-tree would lose its ability to prune any micro-partitions. Since the root node is also an $\vee$-expression, the entire pruning tree would be rendered ineffective. To avoid this situation, only filters below an $\wedge$-expression may be removed.

If pruning for a filter is halted, the filter itself is still retained for query execution. Additionally, if pruning for a slow filter is stopped during query optimization, pruning might still be deferred to the highly parallel query execution stage.

3.3 Impact of Filter Pruning

In Snowflake, filter pruning is executed as the first of all pruning techniques and therefore its pruning ratio is independent of the pruning results of other techniques. Further, filter pruning is generally applicable to all queries that contain at least one table scan with filters that allow pruning.

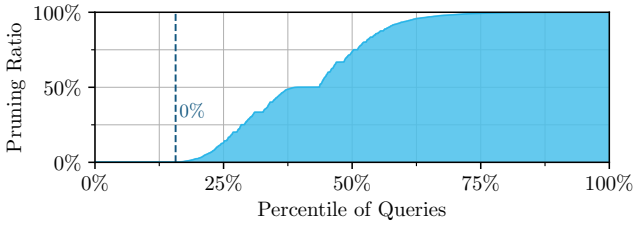


Figure 4: Impact of filter pruning on a representative random sample of `SELECT` queries that had at least one predicate. The sample is across all customers from Nov 5th to Nov 7th 2024.

Figure 4 shows the impact of filter pruning. The pruning ratio should be understood relative to the total number of micro-partitions that are to be processed by a query, including micro-partitions from table scans without any filters. We chose to do so to give insights into the effect of pruning across the whole query and not just single table scans.

Overall, filter pruning is extremely effective, pruning at least around 90% of partitions for 36% of queries. In some cases, filter pruning removes the whole scan set of a table scan, allowing additional optimizations such as sub-tree elimination in queries. As can be seen in Figure 4, around 27% of queries have filters that allow for pruning but do not see a reduction in scan set size. This can have two possible reasons. Either, all partitions indeed contain qualifying rows or the data is poorly distributed with wide min/max ranges.

4 PRUNING FOR LIMIT QUERIES

`LIMIT` queries are common for BI workloads and frequently used during exploratory analysis of unfamiliar datasets [30]. They account for a significant portion of queries executed on Snowflake (see Table 1). Because such queries return only a subset of the qualifying rows, they show a significant potential for pruning.

Most existing database systems simply execute the entire query and halt query processing when the specified `LIMIT` has been reached. Kim et al. [34] proposed the use of auxiliary index structures, called *density maps*, together with specialized retrieval algorithms that prioritize processing "dense" partitions first. These density maps are similar to per-partition histograms. While their approach is elegant, our approach is globally IO-optimal for supported queries as it reads only the minimal number of files required. Further, our approach can be implemented with minimal modifications using only existing min/max metadata. However, it does come with the trade-off that our approach has stronger prerequisites and therefore supports fewer queries.

4.1 Approach

Continuing the example with the animal observation post, the next step a data analyst might take is getting an overview over the tracking data of alpine animals. To do this, they issue the following `LIMIT` query on a table consisting of four micro-partitions as

depicted in Figure 5[4]. In other scenarios, a cybersecurity expert might investigate a few connections from a specific IP address, a dashboard tool might automatically append a default LIMIT to all queries, or a data scientist might apply a LIMIT to source tables as a quick sampling method to reduce response times during model development.

```
SELECT * FROM tracking_data
WHERE species LIKE 'Alpine%' AND s >= 50
LIMIT 3;
```

Applying filter pruning based on the query's predicate still requires scanning three micro-partitions. If we happen to scan partitions 2 and 4 first, the LIMIT would only be reached once the last partition is processed. Ideally, we would identify partition 3 during query compilation as sufficient, allowing us to process only that micro-partition. If the LIMIT exceeds the number of rows in partition 3, the ideal approach would be to process the micro-partitions in descending order of the number of qualifying rows to reduce the number of partitions that need to be scanned, similar to the density-map-based approach by Kim et al. [34]. However, knowing the number of qualifying rows in a partition is not always possible.

To achieve this, we introduce a third category of micro-partitions: In addition to *not-matching* partitions—those pruned away by filter pruning—and *partially-matching* partitions—those that *might* contain qualifying rows and are therefore retained in the scan—we define *fully-matching* partitions, where we can guarantee that all rows of that partition qualify all query predicates. By definition, *fully-matching* partitions are a subset of *partially-matching* partitions. A similar idea called *subsumed partitions*, although used in a different context, was proposed by Ding et al. [21].

Given the definition of fully-matching partitions, we propose a pruning algorithm as follows: if the total row count across all fully-matching partitions meets or exceeds the threshold defined by k in the LIMIT clause, the scan set is constructed to include only the minimum number of fully-matching partitions required to satisfy k. Conversely, if the row count of fully-matching partitions is less than k, or if no fully-matching partitions exist, LIMIT-pruning is not possible. Nevertheless, starting the table scan with fully-matching partitions promises faster query execution times.

[4]Column S contains the height in [cm] for each animal, using realistic values.

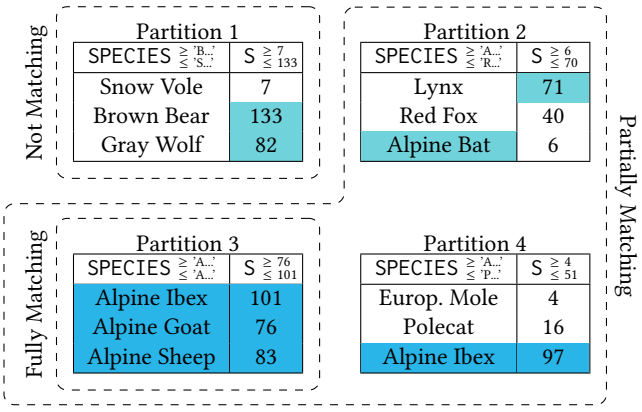


Figure 5: Example of partially and fully matching partitions for a query with predicate `species LIKE 'Alpine%' AND s >= 50` on a table with four micro-partitions. Metadata for SPECIES is shortened. Colors highlight which rows/cells (partially) match the predicate.

Table 1: Relative frequency of different types of LIMIT queries out of all SELECT queries in Snowflake over the course of three days from Nov 5th to Nov 7th 2024, based on pattern-matching on SQL texts.

Type	Percentage
LIMIT queries	2.60 %
LIMIT without predicate	0.37 %
LIMIT with predicate	2.23 %
Top-k queries	5.55 %
ORDER BY x LIMIT k	4.47 %
GROUP BY x ORDER BY x LIMIT k	0.12 %
GROUP BY y ORDER BY agg(x) LIMIT k	0.96 %

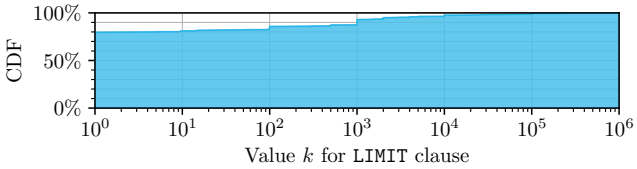


Figure 6: CDF of the distribution of k in LIMIT queries (only $k > 0$). If the query contained an OFFSET, the value for the offset is included. 97% of queries have $k \leq 10,000$ and 99.9% have $k \leq 2,000,000$. Queries are based on a representative random sample of queries with a LIMIT clause across all customers from Nov 5th to Nov 7th 2024.

Although our approach does not enable processing micro-partitions strictly in the order of their number of qualifying rows, in practice, the substantial size of partitions means that identifying one or two fully-matching partitions is typically sufficient to produce the query result. Moreover, the value of k in LIMIT queries is typically small[5], with most queries having $k = 0$ or $k = 1$ (Figure 6). This suggests that optimizing for small values of k is practical.

4.2 Identifying Fully-Matching Partitions

A fully-matching partition is defined as a micro-partition only containing rows satisfying all query predicates. In fact, the filter pruning process can be extended to identify such partitions by including a second pass with inverted base predicates, but without modifying the scan set during this step.

For instance, in the scenario illustrated in Figure 5, the first pruning pass uses the predicate `species LIKE 'Alpine%' AND s >= 50`. This removes partition 1 from the scan set based on its metadata for column SPECIES, leaving partitions 2, 3, and 4 marked as partially-matching. In the second pass, the inverted predicate `species NOT LIKE 'Alpine%' AND s < 50` is applied, under which partition 3 is

[5]Some BI-tools issue queries with `LIMIT 0` appended to receive schema information of the output before sending the actual query.

identified as not-matching. But instead of excluding partition 3 from the scan set, this step confirms that it contains no rows failing the inverted predicate. Consequently, partition 3 is marked as fully-matching and retained in the scan set. Trivially, if a query does not contain predicates, all partitions are considered fully-matching. For queries with predicates, the existence of fully-matching partitions is not guaranteed. Nonetheless, if such partitions do exist, the outlined procedure will accurately identify them.

4.3 Considerations

The first step of pruning LIMIT queries is to determine which table scan's scan set should be pruned. This requires "pushing down" the LIMIT information within the compiler, from the LIMIT operator to the table scans. Generally, operators that reduce the number of rows prevent this pushdown. This includes, e.g., filters where identifying fully-matching partitions is not possible, aggregations, joins, etc. A noteworthy exception to this is propagating the `LIMIT` information through the build side of a (LEFT) OUTER JOIN, which still yields a correct query result. When the LIMIT information reaches a table scan, the corresponding scan set can be pruned accordingly.

4.4 Impact of LIMIT Pruning

Table 2 shows the effect of pruning for `LIMIT` queries. The majority of queries already has a minimal scan set of just a single micro-partition after filter pruning, leaving no opportunity for additional `LIMIT` pruning. Out of the remaining queries we observe that there is a significant number of queries *with predicate* that have an unsupported shape or without *fully-matching* partitions that could be used for pruning. This shows that there is still a significant opportunity to improve pruning for `LIMIT` queries when no *fully-matching* partitions are found. Conversely, if the prerequisites for pruning are met, doing so reduces the number of partitions significantly, mostly to just 1 partition, regardless of the original number of partitions. The cases where there are more than 1 partition left are due to large k in the `LIMIT` clause and still result in the optimal number of partitions. All in all, we conclude that there are still many queries that we can not apply this pruning technique to—but if we can, pruning results in a drastic reduction of the scan set.

Pruning for `LIMIT` queries should be performed *after* filter pruning, as a simple extension to filter pruning can provide the necessary information about fully-matching partitions.

It should be noted, however, that for `LIMIT` queries, the pruning ratio does not always result in a corresponding runtime or even IO improvement. The reason for this is that the query engine would halt execution after enough rows have been processed and most micro-partitions would not have been scanned anyways. However, there is a catch for parallel query execution: if no pruning is applied, the work might be distributed across n machines, each required to scan up to $\lceil k/n \rceil$ rows. This usually means that the query engine reads at least n partitions, even though 1 might have been enough.

5 PRUNING FOR TOP-K QUERIES

Snowflake employs a specialized pruning technique for top-k queries during execution, complementing other pruning mechanisms. Similar to `LIMIT` queries, BI workloads often contain a significant number of top-k queries. Notably, 5.55% of all `SELECT` queries executed on Snowflake fall into this category. Table 1 presents a breakdown of different types of `LIMIT` queries, highlighting their relative frequencies within Snowflake over a one-week period. Top-k queries are characterized by the use of an `ORDER BY` clause in conjunction with a `LIMIT` clause. This means that—logically—the query engine first needs to perform sorting on all qualifying tuples before applying the `LIMIT`. However, few systems actually do this, but instead employ a heap-based approach, as we briefly describe in the next paragraph.

Table 2: Breakdown of `LIMIT` pruning applicability, showing the percentage of `LIMIT` queries falling into each category. We further make a distinction between `LIMIT` queries with and without predicates. Queries are based on a representative random sample of eligible queries across all customers from Nov 5th to Nov 7th 2024.

Queries with...	Without Predicate	With Predicate	Overall
already minimal scan set	79.60%	61.65%	64.22%
unsupported shapes	1.74%	36.23%	31.28%
pruning to = 1 partition	16.58%	1.71%	3.85%
pruning to > 1 partitions	1.54%	0.01%	0.23%

To illustrate this concept, we build upon the last example. A good animal observation post should have a high chance of seeing animals of a certain kind, so the data analyst might try to increase the number of animal sightings by ordering by `num_sightings`:

```
SELECT * FROM tracking_data
WHERE species LIKE 'Alpine%' AND s >= 50
ORDER BY num_sightings DESC LIMIT 3;
```

Similarly, a BI-dashboard might show the top-10 users or a threat-detection tool might identify recent log-in attempts. Note that the `ORDER BY` expression might additionally be wrapped in a function.

Standard Heap-Based Approach. A widely used method (e.g., [44, 47]) for processing top-k queries involves maintaining a max-heap (for queries with `DESC` ordering) that holds up to k elements. Qualifying tuples are progressively inserted into the heap as the table is scanned, and the heap's contents are returned as the final query result. This method scales linearly with the size of the table, becoming expensive for large tables in data analytics systems, as it requires scanning the entire table.

The Theoretically Optimal Solution. Ideally, if the table is physically sorted by the `ORDER BY` key, the query engine should only need to scan the number of micro-partitions required to locate the first k qualifying rows. For instance, in the absence of predicates and assuming each micro-partition contains more than three rows, the engine would only need to scan a single partition if the table is sorted by `num_sightings`. Conversely, if the data is not sorted by `num_sightings`, the query engine should—ideally—scan only the micro-partitions containing the three highest values in the `num_sightings` column. In this example, this would involve reading at most three micro-partitions.

5.1 Existing Research

A substantial body of research in the Information Retrieval (IR) domain focuses on optimizing the deterministic retrieval of top-k elements when joining multiple "posting lists" containing `(doc_id, attributes)` pairs [31]. In this context, the final score for a document is computed using a scoring function that aggregates the individual attributes distributed across these lists.

The Threshold Algorithm (TA). When these lists are sorted by the scoring attribute, the challenge is to efficiently find and communicate a halting point when iterating through the lists. Once a halting point is found, it ensures that no additional tuples can qualify for the result. This traversal technique is often referred to as Term-At-A-Time (TAAT). Fagin et al. [25] proposed the Threshold Algorithm (TA) for this kind of problem which has been extensively studied and refined [13, 43].

The WAND Algorithm. In contrast, when the lists are sorted by *doc_id*, the traversal strategy known as Document-At-A-Time (DAAT) resembles the relational model where tables are split into separate columns. This data model aligns closely with processing top-k queries in relational databases, making it particularly relevant.

Broder et al. [12] introduced the WAND algorithm, which employs a continuously updated *threshold value* to skip documents with scores lower than the threshold. This threshold, representing the k-th largest score encountered thus far, is conceptually similar to the pruning boundary in our approach.

Around the same time, Chakrabarti et al. [15] and Ding et al. [23] independently proposed extensions to the WAND algorithm that introduced the concept of a *block-max* value, enabling skipping of larger segments of data. [15] further suggested to adjust the processing order of blocks based on their *block-max* value. However, both works lacked ideas for initializing the threshold value besides setting it to zero. [23] initially assumed fixed-sized blocks of arbitrary size, a design later enhanced in [38] to use variable-sized blocks. In contrast, Chakrabarti et al. [15] had strict requirements for splitting lists into blocks, resulting in very small partitions. Here, the *score* value is allowed to change only at block boundaries, meaning each block must contain a uniform *score* value. This assumption is impractical for large-scale data processing as it necessitates physical reorganization data. Subsequent research expanded upon the concept of data skipping using *block-max* indexes [11, 18, 19, 33, 45]. All of the aforementioned work lies within the IR space and does not explore ways to fully integrate these approaches into a relational query engine, besides some initial ideas presented in [25].

Provenance Sketches. Another notable approach for processing top-k queries is presented by Niu et al. [41], who utilize per-partition *provenance sketches* to enhance data pruning for such queries. Notably, their method supports data skipping even when the `ORDER BY` clause involves an aggregated column. However, maintaining these sketches efficiently under data updates poses significant challenges.

5.2 Approach

For simplicity, the following section assumes *descending* ordering of a top-k query. Similar to [15] and [23], we extend the heap-based approach and leverage the smallest element in the top-k heap—referred to as the *boundary value*—for partition pruning. Initially, no boundary value exists, as the heap is empty or contains fewer than k rows. Once k rows have been inserted, the boundary value is updated with each new insertion. Snowflake's flexible execution engine supports passing this updated boundary value to the table scan, enabling effective pruning. Before scanning a micro-partition, its metadata is compared against the boundary value. If the partition's maximum value for the `ORDER BY` column is smaller than the boundary value, the micro-partition can be safely skipped, as none of its rows would qualify for the top-k heap. As the query progresses and the heap approaches the final result, the boundary value becomes tighter, allowing for increased partition pruning.

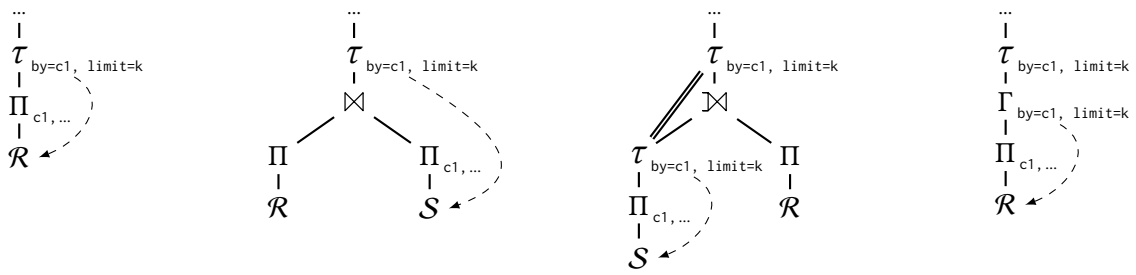
Identifying Supported Table Scans. In general, this pruning technique can only be applied if the TopK operator is part of the same pipeline as the table scan that produces the `ORDER BY` column, meaning that the TopK operator has to sit "on top" of the table scan without any pipeline breakers in between. Specifically, this means that there might be a filter between the table scan and the TopK operator, reducing the number of rows that make it into the top-k heap (see Figure 7a). Still, partition pruning is possible in this case because the boundary value is created based on the rows that make it into the heap. Further, a TopK operator might also be located on top of a `JOIN` and prune micro-partitions coming from the *probe side* of that `JOIN` if the `ORDER BY` column is produced by that side (see Figure 7b). Additionally, for a `(LEFT) OUTER JOIN` and for the case that the `ORDER BY` column is produced by the build side, the TopK operator can be replicated to the *build side* of the `JOIN`, effectively supporting partition pruning on the build side in such cases (see Figure 7a). This is because we can guarantee that all k rows from the build side will be forwarded beyond the `JOIN`.

Supporting Aggregations. In certain cases, it is possible to prune micro-partitions via the TopK operator when there is an aggregation, i.e., a `GROUP BY`, between the TopK node and the table scan (see Figure 7d). If the ordering is not performed on an aggregate column, i.e., the `ORDER BY` columns are a subset of the `GROUP BY` keys, pruning is possible. However, this requires changes to the `GROUP BY` operator to maintain its own top-k heap.

5.3 Processing Order of Data Partitions

The effectiveness of runtime partition pruning depends largely on identifying a tight threshold early and the extent of overlap in micro-partition min/max values. Since our work deliberately assumes we do not have control over the physical layout of data, we cannot influence by how much micro-partitions overlap. Still, the min/max information can be used to process micro-partitions in an order that promises a good threshold value early on.

While the basic idea is simple, there are several considerations when applying this approach in a large-scale, cloud-based data analytics system. For top-k queries with `DESC` ordering, the micro-partitions would be sorted by their *max* values, starting processing with the partition that has the largest *max* value. This strategy prioritizes scanning partitions with larger values first, enabling the construction of a top-k heap that closely approximates the final query result early in the processing phase. Subsequently scanned partitions then refine the result further if their rows are not fully filtered out by predicates.



(a) TopK above Table Scan (b) TopK Join Probe Side (c) TopK Outer Join Build (d) TopK Aggregation

Figure 7: Supported query plans for top-k pruning where τ is used to represent a TopK operator.

However, a naive sorting approach can negatively impact performance. Sorting, especially for datasets with millions of micro-partitions, incurs a non-negligible computational overhead. Additionally, for queries with highly selective filters, naive sorting might lead to a worse processing order of partitions. In such a case, naive sorting might accidentally de-prioritize scanning micro-partitions that actually contain matching rows, potentially leading to a situation where lots of partitions with "large" values are processed that do not contain any qualifying tuples. This, in turn, means that more micro-partitions need to be processed until the top-k heap is fully populated and pruning can start. Unfortunately, sorting based on min/max metadata only cannot guarantee the presence of qualifying rows. Thus, a sorting strategy needs to account for that. We evaluated the following sorting strategies:

(1) **None/random**. No explicit sorting is applied. Instead, a random processing order is used.
(2) **Full sort**. A standard sorting algorithm sorting all micro-partitions by their min/max values.

Figure 8 shows the influence of sorting on partition pruning together with a baseline of "no sorting". Overall, sorting significantly improves the average pruning ratio compared to a random partition order. This not only becomes visible in the improved median but also in the distribution tails, resulting in better worst-case performance. It demonstrates that sorting can often produce a tighter boundary value early, enabling more effective partition pruning.

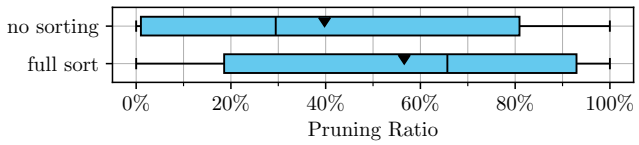


Figure 8: Influence of sorting on the pruning ratio for top-k queries on a representative random sample of eligible queries across all customers on a 7-day interval from Nov 13th to Nov 19th 2024. The sample contains only queries with a minimum runtime of 1 second when top-k pruning was disabled.

5.4 Upfront Initialization of Boundary Values

In our algorithm, pruning begins only after the top-k heap contains enough rows. Early in query execution, any row can enter the heap, often resulting in a sub-optimal boundary value. However, if the metadata includes details like the number of rows per micro-partition and whether a column contains null values, we can pre-compute an initial boundary value during query compilation, enabling pruning from the very first partition.

The approach again uses *fully-matching* partitions, as described in Section 4.2. Given a set of fully-matching partitions, for `DESC`-ordering queries, the boundary can be initialized using either the k-th largest max-value of the `ORDER BY` column of all fully-matching partitions or by sorting these micro-partitions by their min-values of the `ORDER BY` column and selecting the min-value of the first micro-partition where the cumulative row count of all previous partitions meets or exceeds k, whichever yields a stricter boundary. Typically, for highly overlapping micro-partitions, the k-th max-value is generally more effective, while for (partially) sorted tables, the largest min-value is often the better choice.

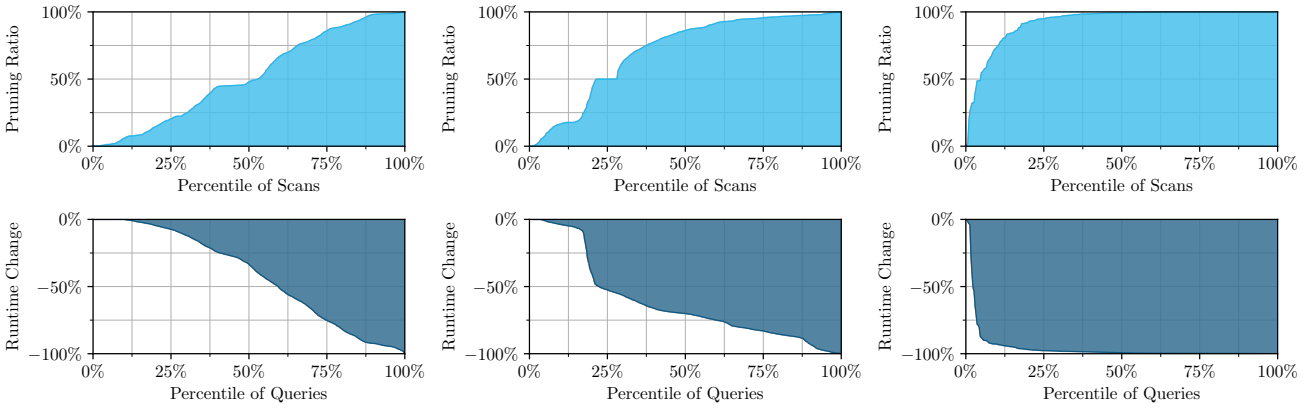
5.5 Impact of Top-K Pruning

In Snowflake, top-k pruning is applied after filter pruning, `JOIN` pruning, and potentially `LIMIT` pruning, meaning its effectiveness depends on the combined outcome of these preceding techniques. Figure 9 shows pruning ratios for table scans where top-k pruning was successfully applied and its effect on overall query runtimes. The CDFs have similar distributions, indicating a strong correlation between pruning and runtime improvement. We categorized queries into buckets based on their runtimes when top-k pruning was disabled, allowing us to observe its impact on both fast and slow queries. Across all buckets, we observed query runtime improvements of more than 99.9%. Out of the queries in our sample, 4.8% also applied `JOIN`-pruning. Overall, the average pruning ratio of successfully applied top-k pruning is around 77%.

6 PRUNING FOR JOIN QUERIES

Joins are some of the most time-consuming and compute-intensive operations in a database system, making it crucial to avoid unnecessary work during these operations. This section explains how partition pruning can be used to improve join queries significantly.

Join pruning can be seen as a form of sideways information passing [32]. Unlike row-level techniques like *bloom-joins* [37, 40], join pruning in Snowflake operates at a coarser granularity, excluding entire micro-partitions from join processing on the probe side.



(a) $1\,s < t \leq 10\,s$ **(b)** $10\,s < t \leq 60\,s$ **(c)** $t > 60\,s$

Figure 9: The first row shows CDFs of the pruning ratio (higher is better) of top-k pruning on table-scan level where top-k pruning was successfully applied. The plots in the second row show CDFs of the relative runtime improvement of queries. Queries are randomly sampled from representative real-world workloads over a period of 7-days from Nov 13th to Nov 19th 2024, bucketed by query execution times. The execution time t of a query was measured when top-k pruning was disabled.

6.1 Approach

Continuing the example, we identify a suitable location for the animal observatory. The goal is to select a site along an existing hiking trail on a mountain ridge that offers high chances of observing large, local wildlife:

```
SELECT *
FROM trails t JOIN tracking_data d ON t.mountain = d.area
WHERE true
  AND IF(unit='feet', altit * 0.3048, altit) > 1500
  AND name LIKE 'Marked-%-Ridge'
  AND species LIKE 'Alpine%' AND s >= 50
ORDER BY d.num_sightings DESC LIMIT 3;
```

The overarching goal of join pruning is to reduce the work done on the probe side of the join. Work can be reduced at different levels. (i) Computational work can be reduced by not having to check if a probe side value is part of the build side's hash table. (ii) In addition, and most important for a cloud-based data processing system with decoupled compute and storage, we can reduce (network) I/O by avoiding to load probe side micro-partitions altogether.
The general idea of join pruning can be divided into four steps:

(1) Summarize the entirety of the build side values during the hash join's build phase.
(2) Ship the previously generated summary from the build side to the probe side.
(3) Match the summary against the join probe side's micro-partitions' min/max values.
(4) Prune those micro-partitions whose min/max values do not overlap with the summary.

In our example, selective filters on the build side substantially reduce the `trails` table, creating pruning opportunities on the probe side. Further, the probe side already underwent filter pruning and top-k pruning can be applied during the join—resulting in three distinct pruning techniques being used on the `tracking_data` table.

Summarizing Build-Side Values. Summarizing build side values is a trade-off between accuracy and the memory size of the employed data structure. In a distributed setting, this summary needs to be sent to other workers over the network, which forbids excessive data sizes. A very cheap but inaccurate value summary would only store the global min/max values of the build-side which comes with negligible storage overhead but low pruning potential. Most of the time, a Bloom-filter [10] or a similar data structure is used in sideways information passing to summarize build-side values. The Bloom-filter is then used to skip probing the join's hash table for individual rows, which reduces CPU usage significantly.

The value summary technique employed by Snowflake allows pruning whole micro-partitions and strikes a balance between accuracy and storage cost: while spending only a small fraction of the build-side size for the summary, Snowflake does still manage to prune 79% of micro-partitions of probe-side scans. In practice, we observed probe-side scan sets being reduced by up to 99.99%. This trade-off demonstrates the probabilistic character of the approach. For pruning micro-partitions of the probe-side, build-side value summaries are overlapped with probe-side partition min/max metadata to efficiently check whether a micro-partition might contain any joinable tuples. Notably, this does not just reduce CPU usage by avoiding checking tuples against the hash table of the build-side, but also significantly reduces network I/O because micro-partitions are pruned before they are loaded from storage.

6.2 Limitations

The join pruning technique discussed in this section is targeted towards hash-based joins. In addition, join pruning requires a storage model as explained in Section 2 consisting of micro-partitions with per-column min/max metadata. The join pruning technique presented is probabilistic, i.e., it might miss pruning a micro-partition that could theoretically be pruned, but will never prune a partition that must not be pruned.

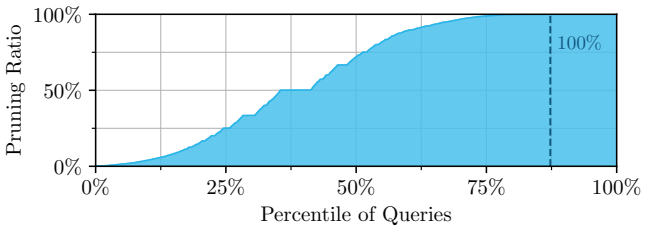


Figure 10: Impact of join pruning for SELECT queries that were able to successfully use join pruning. Based on a representative random sample of eligible queries across all customers from Nov 5th to Nov 7th 2024.

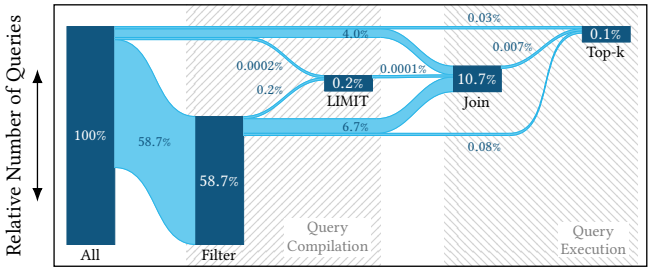


Figure 11: Flow diagram visualizing how many queries are subject to which pruning technique(s). Pruning techniques are executed from left to right in order. Queries are based on a representative workload sample across all customers from Mar 25th to Mar 28th 2025.

6.3 Impact of Join Pruning

Join pruning is executed after compile-time pruning and therefore its pruning ratio depends on filter pruning and—in theory—on LIMIT pruning. However, a LIMIT below a join rarely makes sense. Filter pruning can incorporate filters from the build side to a limited extent only, as demonstrated by a technique known as *data-induced predicates* [42]. Join pruning, on the other hand, has a full overview over the tuples on the build side and therefore is able to effectively prune micro-partitions on the probe side. Figure 10 depicts the average scan set reduction of probe-side tables resulting from join pruning. Most notably, around 13% of queries see a pruning ratio of 100%, meaning that the probe-side scan does not need to be performed. This might be caused by an empty build-side. Further, we see that join pruning is generally very effective, with 50% of queries seeing a scan set reduction of at least 72%.

7 THE PRUNING FLOW IN SNOWFLAKE

In the previous sections, we examined pruning techniques in isolation. Here, we briefly summarize their combined applicability. Figure 11 shows the share of queries with successful partition pruning by different technique combinations and the order in which Snowflake applies them. Filter pruning at execution time is omitted. Note that this plot includes both DML and SELECT queries in Snowflake, reflecting a corresponding distribution of query types. All percentages are relative to 100%, though minimum heights have been applied to lines and boxes for readability. As shown, most queries benefit from filter pruning. Note that the percentages indicate queries where at least one partition was pruned using a given technique—they do not imply, e.g., that only 58.7% of queries have a WHERE clause. A query may contain a WHERE clause but still be non-selective or non-prunable.

8 DISCUSSION

In the following sections, we will discuss how pruning can be done for open table formats like Apache Iceberg, extend and discuss the idea of *predicate caching*, and lastly show that synthetic benchmarks do not accurately capture the importance of pruning.

8.1 Pruning for Iceberg Tables in a Data Lake

Besides its highly optimized proprietary format, Snowflake also supports the Apache Iceberg [2] open table format backed by the Apache Parquet file format [3]. Snowflake's query engine seamlessly handles both formats, with pruning techniques operating transparently across them. Consequently, all pruning methods discussed in this paper apply equally to Iceberg tables in Snowflake. Similar to Snowflake's own file format, the Apache Parquet file format also follows a PAX-style layout [8], allowing columnar metadata that can be used for pruning on row-group level. Parquet files may also include page-level indexes [6] and manifest files in Apache Iceberg can further contain metadata on file-level, which is also utilized by Snowflake for pruning.

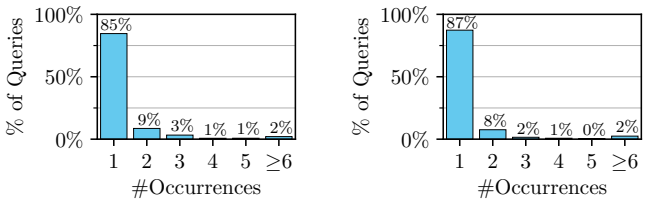
Metadata is the cornerstone of pruning—without it, no pruning is possible and therefore the quality and correctness of metadata plays a key role in query performance. Consequently, backfilling missing metadata is crucial for analytical systems operating on data lakes. If a Parquet file contains metadata, Snowflake can immediately use it for pruning. However, if there is no metadata, Snowflake can reconstruct it by performing a full table scan to compute missing metadata entries, which can then be used for subsequent queries. Similarly, if an Iceberg table does not contain metadata in manifest files, Snowflake can reconstruct it using the metadata from the underlying Parquet files.

8.2 What about Predicate Caching?

Schmidt et al. recently proposed *predicate caching* [46]. While they focus on caching relevant partitions for filters only, an extension to top-k queries can be envisioned as follows:

For repetitive top-k queries (see Figure 12), the micro-partitions contributing to the final top-k result can be identified by recording partition information alongside each tuple in the top-k heap during query processing. This list of contributing micro-partitions can then be stored in a global "predicate cache." When the same top-k query is executed again, the query engine performs a cache lookup, and if a matching entry exists, it processes only the cached micro-partitions. If predicate caching was perfect—yielding no false-positive partitions—the query engine would scan only the partitions necessary to produce the top-k result. This approach could outperform our pruning-based method, especially for randomly sorted datasets with mostly overlapping min/max ranges where pruning may struggle to exclude many partitions.

However, due to the space limitations of the predicate cache, the caching result might include a significant number of false-positive



(a) Over 3 days (b) Over 1 month

Figure 12: Repetitiveness of top-k query plan shapes in Snowflake over a 3-day and 1-month period. Most query plan shapes appear only once. Queries are based on a representative workload across all customers from Nov 5th to Nov 7th 2024, and from Oct 8th to Nov 7th 2024, respectively.

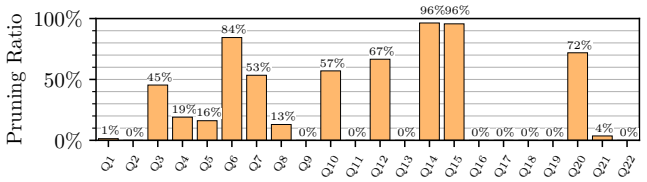


Figure 13: Pruning ratios of queries in TPC-H SF100, clustered on `l_shipdate` and `o_orderdate`, executed on an XSMALL virtual warehouse in Snowflake as of March 2025. Average pruning ratio over the whole workload is 28.7%, with a median per-query pruning ratio of 8.3%.

partitions, especially for large datasets. For large, (partially) sorted tables, our pruning-based method can more effectively exclude non-relevant micro-partitions, likely outperforming predicate caching.

To maximize efficiency, we argue that both techniques should be implemented, as their respective strengths can complement each other and address their individual shortcomings.

The integration of predicate caching with top-k queries introduces additional complexities when dealing with `UPDATE`s and `DELETE`s. While Schmidt et al.'s predicate caching for filters supports `INSERT`, `UPDATE`, and `DELETE` operations on the base table, the same flexibility does not fully extend to top-k queries.

If a row in the top-k result is deleted, another row must take its place. However, there is no guarantee that this replacement row (the $k+1$-th row) resides within the cached micro-partitions, potentially leading to incorrect results. Similarly, `UPDATE`s to the ordering column may reorder rows in a way that invalidates the cached partitions. In contrast, `UPDATE`s to non-ordering columns and `INSERT`s are safe and do not affect the correctness of the cache.

In contrast, our pruning-based approach for top-k queries is naturally robust against DML operations that modify the ordering column, maintaining correctness without the need for cache invalidation, making it a robust choice particularly for dynamic datasets. Additionally, we showed that top-k queries are generally not particularly repetitive (see Figure 12), which diminishes the potential of *predicate caching*. In contrast to that, our approach also supports ad-hoc top-k queries.

8.3 Drawing a Line to TPC-H

Comparing the pruning capabilities of real-world customer workloads to artificial benchmarks like TPC-H showed some stark differences that are briefly discussed in the following. We ran TPC-H SF100 on an XSMALL virtual warehouse in Snowflake to ensure pruning behavior was not influenced by effects of data placement across different machines. We clustered the dataset by `l_shipdate` and `o_orderdate` to allow better partition pruning and exploitation of column correlations as discussed by Dreseler et al. [24]. Figure 13 shows the pruning ratios of TPC-H SF100 queries in this setup. Notably, these ratios are significantly lower than those observed for queries executed on Snowflake. As most pruning came from filter pruning on tables `LINEITEM` and `ORDERS` and no pruning happened with default data clustering, it is clear that this discrepancy comes from the relatively low selectivity of query predicates in TPC-H compared to real-life workloads. Further, no query in TPC-H was able to exploit top-k pruning and due to the deterministic nature of the benchmark, no `LIMIT` pruning was possible. Join pruning was also greatly underrepresented.

The key takeaway is that the impact of partition pruning is not adequately captured by standard synthetic benchmarks like TPC-H, reinforcing our earlier claim about the challenges of accurately evaluating pruning techniques. We are not the first to notice this. Previous work also aimed to address similar issues with synthetic benchmarks [20, 51]. While synthetic benchmarks can approximate real-world pruning behavior, careful query design is essential. We identify the key aspect to be a more realistic selectivity of predicates. Further, enough opportunity for join pruning needs to exist, e.g., by designing very small build sides or having a sufficient correlation in data layout between build and probe sides. Lastly, such a benchmark should also contain non-deterministic `LIMIT` queries.

9 CONCLUSION

In this paper, we presented a comprehensive analysis of partition pruning techniques of the Snowflake Data Platform. We explained the functioning and evaluated the impact and interplay of four pruning strategies: filter pruning, `LIMIT` pruning, top-k pruning, and `JOIN` pruning. Our findings, based on real-world customer workloads, demonstrate the substantial performance benefits these techniques offer, in particular for cloud-based systems. Every presented pruning technique achieves substantial pruning ratios for applicable queries with 99% for filter pruning, 70% for `LIMIT` pruning, 77% for top-k pruning, and 79% for join pruning. Overall, pruning is one of the enablers of cloud-based data processing with Snowflake pruning 99.4% of micro-partitions across all queries.

We further showed that synthetic benchmarks do not accurately capture the importance of partition pruning for query performance. By sharing our experiences with Snowflake's pruning mechanisms, we aim to inspire further research and innovation in this area.

ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of the many members of the Snowflake team, whose collective efforts formed the foundation of this paper. We specifically wish to thank Abdul Q Munir, Hossein Ahmadi, Berni Schiefer, Thierry Cruanes, Chris Baynes, and Changbin Song for their invaluable support of this work.

REFERENCES

[1] Amazon Web Services, Inc. 2025. *Amazon S3 - Cloud Object Storage.* Amazon Web Services, Inc. https://aws.amazon.com/s3/

[2] The Apache Software Foundation 2025. *Apache Iceberg™: The open table format for analytic datasets.* The Apache Software Foundation. https://iceberg.apache.org/

[3] The Apache Software Foundation 2025. *Apache Parquet.* The Apache Software Foundation. https://parquet.apache.org/

[4] Microsoft Corporation 2025. *Azure Blob Storage.* Microsoft Corporation. https://azure.microsoft.com/en-us/products/storage/blobs

[5] Alphabet Inc. 2025. *GCP Cloud Storage.* Alphabet Inc. https://cloud.google.com/storage

[6] The Apache Software Foundation 2025. *Parquet page index: Layout to Support Page Skipping.* The Apache Software Foundation. https://github.com/apache/parquet-format/blob/master/PageIndex.md

[7] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 359–370. doi:10.1145/1007568.1007609

[8] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 169–180. http://www.vldb.org/conf/2001/P169.pdf

[9] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *Proc. VLDB Endow.* 12, 13 (2019), 2393–2407. doi:10.14778/3358701.3358707

[10] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. doi:10.1145/362686.362692

[11] Edward Bortnikov, David Carmel, and Guy Golan-Gueta. 2017. Top-k Query Processing with Conditional Skips. In *Proceedings of the 26th International Conference on World Wide Web Companion, Perth, Australia, April 3-7, 2017*, Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich (Eds.). ACM, 653–661. doi:10.1145/3041021.3054191

[12] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003.* ACM, 426–434. doi:10.1145/956863.956944

[13] Pei Cao and Zhe Wang. 2004. Efficient top-K query calculation in distributed networks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, Soma Chaudhuri and Shay Kutten (Eds.). ACM, 206–215. doi:10.1145/1011767.1011798

[14] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. 1982. Horizontal Data Partitioning in Database Design. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, Orlando, Florida, USA, June 2-4, 1982*, Mario Schkolnick (Ed.). ACM Press, 128–136. doi:10.1145/582353.582376

[15] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. 2011. Interval-based pruning for top-k processing over compressed lists. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 709–720. doi:10.1109/ICDE.2011.5767855

[16] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. doi:10.1145/2882903.2903741

[17] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games (Lecture Notes in Computer Science, Vol. 5125)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer, 385–396. doi:10.1007/978-3-540-70575-8_32

[18] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. A candidate filtering mechanism for fast top-k query processing on modern cpus. In *The 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland - July 28 - August 01, 2013*, Gareth J. F. Jones, Paraic Sheridan, Diane Kelly, Maarten de Rijke, and Tetsuya Sakai (Eds.). ACM, 723–732. doi:10.1145/2484028.2484087

[19] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. Optimizing top-k document retrieval strategies for block-max indexes. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis (Eds.). ACM, 113–122. doi:10.1145/2433396.2433412

[20] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388. doi:10.14778/3484224.3484234

[21] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. 2022. SageDB: An Instance-Optimized Data Analytics System. *Proc. VLDB Endow.* 15, 13 (2022), 4062–4078. doi:10.14778/3565838.3565857

[22] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 418–431. doi:10.1145/3448016.3457270

[23] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, Wei-Ying Ma, Jian-Yun Nie, Ricardo Baeza-Yates, Tat-Seng Chua, and W. Bruce Croft (Eds.). ACM, 993–1002. doi:10.1145/2009916.2010048

[24] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220. doi:10.14778/3389133.3389138

[25] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66, 4 (2003), 614–656. doi:10.1016/S0022-0000(03)00026-6

[26] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo (Eds.). ACM, 75–88. doi:10.1145/2674005.2674994

[27] Goetz Graefe. 2009. Fast Loads and Fast Queries. In *Data Warehousing and Knowledge Discovery, 11th International Conference, DaWaK 2009, Linz, Austria, August 31 - September 2, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5691)*, Torben Bach Pedersen, Mukesh K. Mohania, and A Min Tjoa (Eds.). Springer, 111–124. doi:10.1007/978-3-642-03730-6_10

[28] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *Journal of Experimental Algorithmics (JEA)* 25 (2020), 1–16.

[29] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Ganceanu, and Marc Nunkesser. 2012. Processing a Trillion Cells per Mouse Click. *Proc. VLDB Endow.* 5, 11 (2012), 1436–1446. doi:10.14778/2350229.2350259

[30] Pat Hanrahan. 2012. Analytic database technologies for a new kind of user: the data enthusiast. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 577–578. doi:10.1145/2213836.2213902

[31] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4 (2008), 11:1–11:58. doi:10.1145/1391729.1391730

[32] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 774–783. doi:10.1109/ICDE.2008.4497486

[33] Omar Khattab, Mohammad Hammoud, and Tamer Elsayed. 2020. Finding the Best of Both Worlds: Faster and More Robust Top-k Document Retrieval. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, Jimmy X. Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu (Eds.). ACM, 1031–1040. doi:10.1145/3397271.3401076

[34] Albert Kim, Liqi Xu, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya Parameswaran. 2016. Optimally leveraging density and locality to support limit queries. *arXiv preprint arXiv:1611.04705* (2016).

[35] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter A. Boncz, and David G. Andersen. 2020. Cuckoo Index: A Lightweight Secondary Index Structure. *Proc. VLDB Endow.* 13, 13 (2020), 3559–3572. doi:10.14778/3424573.3424577

[36] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proc. VLDB Endow.* 12, 5 (2019), 502–515. doi:10.14778/3303753.3303757

[37] Lothar F. Mackert and Guy M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Local Queries. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, Carlo Zaniolo (Ed.). ACM Press, 84–95. doi:10.1145/16894.16863

[38] Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. 2017. Faster BlockMax WAND with Variable-sized Blocks. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*, Noriko Kando, Tetsuya Sakai, Hideo Joho, Hang Li, Arjen P. de Vries, and Ryen W. White (Eds.). ACM, 625–634. doi:10.1145/3077136.3080780

[39] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 476–487. http://www.vldb.org/conf/1998/p476.pdf

[40] James K. Mullin. 1990. Optimal Semijoins for Distributed Database Systems. *IEEE Trans. Software Eng.* 16, 5 (1990), 558–560. doi:10.1109/32.52778

[41] Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. 2021. Provenance-based Data Skipping. *Proc. VLDB Endow.* 15, 3 (2021), 451–464. doi:10.14778/3494124.3494130

[42] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252–265. doi:10.14778/3368289.3368292

[43] HweeHwa Pang, Xuhua Ding, and Baihua Zheng. 2010. Efficient processing of exact top-k queries over disk-resident sorted lists. *VLDB J.* 19, 3 (2010), 437–456. doi:10.1007/S00778-009-0174-X

[44] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. doi:10.1145/3299869.3320212

[45] Oscar Rojas, Veronica Gil-Costa, and Mauricio Marín. 2013. Distributing Efficiently the Block-Max WAND Algorithm. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013 (Procedia Computer Science, Vol. 18)*, Vassil Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Elsevier, 120–129. doi:10.1016/J.PROCS.2013.05.175

[46] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 347–359. doi:10.1145/3626246.3653395

[47] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, Carlo Zaniolo (Ed.). ACM Press, 340–355. doi:10.1145/16894.16888

[48] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1115–1126. doi:10.1145/2588555.2610515

[49] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented Partitioning for Columnar Layouts. *Proc. VLDB Endow.* 10, 4 (2016), 421–432. doi:10.14778/3025111.3025123

[50] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425. doi:10.14778/3583140.3583156

[51] Chengcheng Wan, Yiwen Zhu, Joyce Cahoon, Wenjing Wang, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Alexandra M. Ciortea, Konstantinos Karanasos, and Subru Krishnan. 2023. Stitcher: Learned Workload Synthesis from Historical Performance Footprints. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 417–423. doi:10.48786/EDBT.2023.33

[52] Eugene Wu and Samuel Madden. 2011. Partitioning techniques for fine-grained indexing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 1127–1138. doi:10.1109/ICDE.2011.5767830

[53] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 193–208. doi:10.1145/3318464.3389770