Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses

Tobias Schmidt*
Technische Universität München
Munich, Germany
tobias.schmidt@in.tum.de

Andreas Kipf*
UTN
Nuremberg, Germany
andreas.kipf@utn.de

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

Gaurav Saxena
Amazon Web Services
Palo Alto, USA
gssaxena@amazon.com

Tim Kraska
Amazon Web Services
Boston, USA
timkrask@amazon.com

ABSTRACT

Cloud data warehouses are today's standard for analytical query processing. Multiple cloud vendors offer state-of-the-art systems, such as Amazon Redshift. We have observed that customer workloads experience highly repetitive query patterns, i.e., users and systems frequently send the same queries. In order to improve query performance on these queries, most systems rely on techniques like result caches or materialized views.

However, these caches are often stale due to inserts, deletes, or updates that occur between query repetitions. We propose a novel secondary index, predicate caching, to improve query latency for repeating scans and joins. Predicate caching stores ranges of qualifying tuples of base table scans. Such an index can be built on the fly, is lightweight, and can be kept online without recomputation.

We implemented a prototype of this idea in the cloud data warehouse Amazon Redshift. Our evaluation shows that predicate caching improves query runtimes by up to 10x on selected queries with negligible build overhead.

CCS CONCEPTS

• **Information systems** → **Data scans**; **Online analytical processing engines**; **Data warehouses**.

KEYWORDS

predicate caching, data warehouses, analytical database, indexing

ACM Reference Format:
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-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653395

*Work done while at Amazon Web Services.

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3653395

1 INTRODUCTION

Over the last ten years, cloud data warehouses have taken over the data analytics market. Systems like Amazon Redshift or Snowflake are fast, scalable, and elastic and provide almost unlimited storage capacity to their customers. Their success is based on highly distributed and parallel query engines that process millions of queries every day on multi-terabyte datasets. Besides, they are easy to use and do not require any maintenance from the customer.

Internally, cloud data warehouses typically use columnar table formats, split the data into smaller blocks, and compress individual blocks. Modern data warehouses scale to more than 100 nodes and process queries in parallel on all nodes. Providers strive to improve their cloud database systems to provide the best possible performance to customers and reduce the cost of running queries.

In this paper, we present *predicate caching*, a practical technique to speed up analytical queries. The idea of predicate caching is to cache frequent base table scan expressions with their qualifying row ranges. Predicate caching can be implemented as a side-product of query processing and is cheap to maintain. It is motivated by the fact that cloud analytics workloads and queries are highly repetitive. Our analysis shows that for more than 50 % of the Amazon Redshift clusters, at least 75 % of the queries repeat within a month.

The canonical technique for exploiting repetitive workloads is caching, or in the broader sense, materializing query results. Cloud data warehouses, including Redshift, rely on result caches and materialized views. (Multi-dimensional) sorting specific to a given workload can also be seen as a way of caching. We classify the efficiency of the techniques according to the following criteria:

Build Overhead The time needed for creating the cache.
Maintenance Overhead The time required to update the cache if the workload or the underlying data changes.
Gain The possible performance improvements.
Hit Rate The ratio of queries that are answered by the cache.

Table 1 compares the caching techniques according to these criteria. Result caching materializes the query result, has low build overhead, is reasonably cheap to maintain, and can lead to massive speedups. However, result caches are invalidated by updates to the scanned tables, and their hit rate is limited. For example, we found that if there are almost no updates, the result cache answers almost 80 % of the queries on a given cluster. However, on average, across the fleet, only around 20 % of the queries are answered by the result cache.

Table 1: Comparison of different caching techniques.

	Build Overhead	Maintenance Overhead	Gain	Hit Rate
Result Caching	++	+	++	−−
MVs	−	−−	+	++
Sorting	−−	+	+	++
Predicate Caching	++	+	+	+

Materialized views (MVs) are expensive to build and maintain and provide lower speedups than result caching as they are often not stored in memory but on disk or cloud storage. Their hit rate, however, is higher for repetitive queries since they generalize query instances and can answer queries with similar structures but different literals.

Sorting or clustering the data is another way to speed up repetitive queries. Systems like Redshift automatically apply sort keys for the observed workload [11]. Recent proposals for instance-optimized data clustering, even customize the data layout to the predicates found in workloads [33]. Sorting a table or a copy of a table is expensive, especially under changing data and workload distributions. However, once a table is sorted, accessing its data is usually more efficient, as parts of the table can be skipped.

Predicate caching fills the gap between result caching and MVs / sorting: it has low build and maintenance overhead with high hit rates. It can quickly adapt to new workloads and data distributions.

Notably, cloud data warehouses typically do not maintain indexes since they are too large and expensive. However, given the repetitive nature of workloads, we believe it is time to rethink that assumption. A query-driven secondary index only indexes the data that is most relevant to the workload. Our work has some similarities to SPA [13], a recent proposal that builds a secondary index based on the observed workload. Similar to database cracking [24], SPA builds the index over time, and subsequent queries share the work of index building. In contrast to SPA, predicate caching builds the index in a single shot, meaning if the same query comes around again, it will have the full benefit of the index. While SPA stores an index per data block, predicate caching stores an inverted index mapping from scan expressions to row ranges. Row ranges in predicate caching are approximate (bound to a maximum number) to keep the index overhead low.

Predicate caching also works for less selective queries. For example, if the first half of the table qualifies, with the exception of a few rows, the predicate cache would just store identifiers of the first and of the last qualifying row. Another key differentiating factor of predicate caching is that it also considers joins. As we show later in the paper, a cache entry can be made up of join as well as base table predicates. Redshift uses sideways information passing (semi-join filters) to prune scanned rows that do not qualify. Predicate caching stores the row ranges surviving possible semi-join filters, capturing the effect of joins.

Our experiments show runtime improvements by up 10x on selected queries while rigorously avoiding slowdowns. The build and maintenance overhead of predicate caching is almost immeasurable, and memory consumption is within a few megabytes per entry. On customer workloads, we observe hit rates up to 90 %.

Our contributions are as follows:

(1) We analyze the workload patterns on Amazon Redshift clusters and provide insights into the repetitiveness of queries and scans in real-world applications (cf. Section 2).
(2) In Section 3, we evaluate three different caching techniques and highlight their advantages and disadvantages.
(3) Based on the gained insights, we propose a novel caching technique that combines the advantages of the existing techniques, avoiding high build overhead while achieving high hit rates (cf. Section 4).

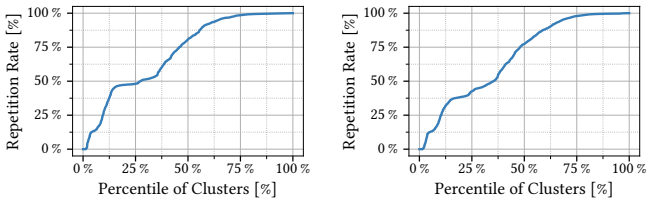
The rest of the paper is structured as follows. Section 5 provides a detailed evaluation of predicate caching in Amazon Redshift. We cover related work in Section 6 and draw a conclusion in Section 7.

2 WORKLOAD ANALYSIS

We analyze the workload patterns from a representative sample of Redshift clusters deployed in the us-east-1 region in January 2023. Our analysis focuses on how effective different caching techniques can be in cloud data warehouses. The two most interesting aspects are the repetitiveness of customer queries and the types of operations executed on the clusters.

2.1 Query Repetitiveness

Caches operate on the assumption that the system's workload repeats over time, and the result or parts of it can be reused. The more repetitive the workload, the more effective the cache. Hence, we start by analyzing how often queries repeat: a query is repetitive if the same statement, including the parameters, is seen at least twice.



(a) 1 month **(b) 1 week**

Figure 1: Percentage of queries that repeat per cluster (i.e., the same query is seen at least twice).

Figure 1 shows the percentage of queries that repeat per cluster over one month and one week, respectively. On the x-axis, we list the clusters in increasing repetitiveness order. On the y-axis, we plot the repetition rate of the respective cluster. For example, 50 % of the clusters have at least 75 % of the queries repeated. We can also observe that the last 25 % of the clusters constantly evaluate the same queries, and the repetition rate is close to 100 %.

This behavior matches our expectations for a cloud data warehouse: customers typically use these systems for data exploration, reporting, and business intelligence applications. Queries are sent periodically to analyze business data and aggregate it to prepare

dashboards or reports. Consequently, there is a high chance that the clusters evaluate the same queries again and again.

2.2 Executed Statements

Besides the repetition rate of queries, we must also consider the type of statements executed on the clusters. Most caching techniques are only effective as long as the underlying data set does not change. So, we next analyze what type of SQL statements clusters typically execute and whether these impact the effectiveness of caches.

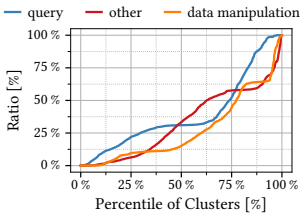


Figure 2: Types of SQL statements executed on the Redshift clusters.

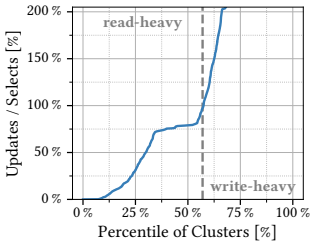


Figure 3: Ratio of data manipulation statements vs. queries.

Figure 2 shows the different types of statements and how they contribute to the query mix on Redshift clusters. We classify the statements into the following four categories: (1) select statements, (2) data manipulation (insert, update, delete, copy statements), and (3) other statements. Select statements make up most of the queries; however, they are less common than expected for an OLAP system. Only for 25 % of the clusters, the executed SQL queries consist primarily of selects (>50 %). Data manipulation statements that modify tables are common and account for almost as much as select statements.

As these statements might invalidate the cached entries, we compare the number of insert, delete, update, and copy statements against the number of select statements in Figure 3. The figure emphasizes the diverse query mix we observe in the real world: 60 % of the Redshift clusters execute more read than write queries. For the remaining 40 % of the clusters, the number of data manipulation statements is higher than the number of read queries.

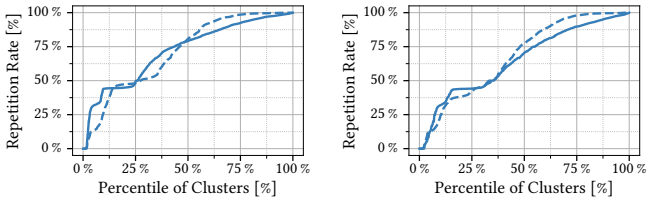
Table 2: SQL statements run on the clusters over one month

Category	Type	Percentage
(1)	`select`	42.3 %
(2)	`insert`	17.8 %
(2)	`copy`	6.9 %
(2)	`delete`	6.3 %
(2)	`update`	3.6 %
(3)	other	23.3 %

Table 2 breaks down the four categories: Besides select statements, the data ingestion statements insert and copy comprise the largest portion (24.7 %) of the workload. In contrast, deletes and updates account for fewer statements (9.9 %). The third category includes the remaining statements and accounts for 23.3 % of the workload. This category includes statements that create, alter, analyze, or vacuum tables and other Redshift-internal operations. Analyze and vacuum statements can be issued by the customer or are executed automatically by the system to update statistics or reorganize the physical data layout.

2.3 Scan Repetitiveness

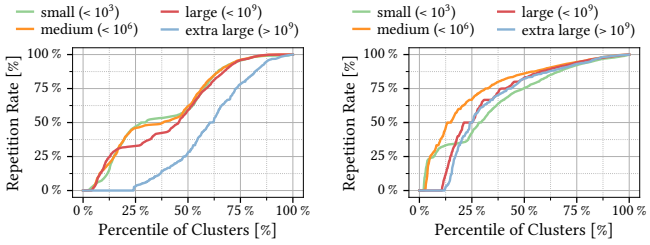
A key insight of our analysis is that queries exhibit a high degree of repetition and operate on data that undergoes regular changes. This workload pattern presents a challenge for implementing caches in cloud data warehouses. In light of these results, we explore if caching only parts of the query is more effective. Our predicate cache operates on a more fine-grained level and only stores the ranges of qualifying tuples in table scans. This avoids accessing tuples not required by the query without caching the actual query results. In this section, we analyze how repetitive the scans with filter conditions are and how beneficial such a technique can be.



(a) 1 month **(b) 1 week**

Figure 4: Percentage of filtered scans that repeat per cluster (that dashed line show the query repetition rate from Figure 1). We only consider base table scans triggered by select statements.

Figure 4 compares the repetition rate of queries and scans per Redshift cluster. Both rates are almost identical, and only minor differences can be observed. The differences are caused by the fact that queries can consist of multiple scans, and we count every scan individually. As a result, different queries might share the same scans, increasing the repetition rate for scans. In addition, we only consider scans with a filter condition. Since some clusters predominantly execute full table scans, we observe a lower repetition rate.



(a) Repeating Queries **(b) Repeating Scans**

Figure 5: Percentage of queries and scans that repeat per cluster grouped by the size of the scanned tables.

Scans and queries are equally repetitive: on average, they repeat 71.9 % and 71.2 % of the time. Besides the repetition rate, the size of the scanned tables is another crucial factor. In Figure 5, we split the scans into four groups based on the number of read rows. Queries are categorized according to the largest table they scan. The two figures show an interesting picture: Queries on extra-large tables are less repetitive than queries on smaller tables (< 10^9 rows), but this is not true for scans. Instead, the repetition rate for scans is mostly the same across the four different categories, with scans on medium-sized tables being slightly more repetitive. Since we observe high scan repetition rates for extra-large tables, caching scans might be an effective alternative to caching queries.

2.4 Takeaways

From our analysis, we can draw the following conclusions:

(1) Cloud data warehouse workloads are highly repetitive, and customers repeatedly send similar queries.
(2) Regular updates on the data set make it difficult to implement caches that store the entire query result.
(3) Select statements account for only 42.2 % of the workload, less than expected for an analytical system.
(4) Scans are as repetitive as queries, and caching them instead might be worthwhile, especially since scans on large tables are more likely to repeat as queries.

3 DESIGN OBJECTIVES

This section discusses three different caching techniques that exploit the repeating workloads in cloud data warehouses. The first technique, result caching, is employed by multiple cloud database systems to cache the results of queries. The second and third techniques compute materialized views over repeating query templates and adapt the data layout to the workload.

Based on this analysis, we derive the four design objectives that minimize the overhead of the caches and maximize the hit rate.

3.1 Result Caching

Result caching is straightforward to integrate into cloud data warehouses. The cluster leader node stores the result of recently executed queries in a local cache. If the customer sends the same query a second time, the database immediately returns the result from the cache without executing the query. However, a hit in the cache requires that both the query text and the dataset are identical.

The advantage of a result cache lies in its simplicity. It is a lightweight technique that does not require changes to the database or the query execution engine. Result caching has almost no build overhead: the query results are simply stored in the cache as they are computed. Maintenance overhead is also low, as the cached entries are only invalidated if one of the scanned base tables changes and new entries can be added on the fly. In addition, the performance improvement is significant in case of a cache hit; queries that would otherwise take minutes to execute run in a few milliseconds.

The main drawback of result caching is the low hit rate. Figure 6 shows the hit rate of the result cache for the clusters from Section 2. Although queries are highly repetitive in this workload, the hit rate is very low. In only 15 % of the clusters, result caching can answer more than 50 % of the queries from the cache.

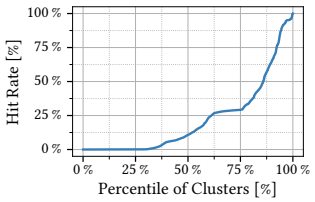


Figure 6: Hit rate of the result cache in Redshift.

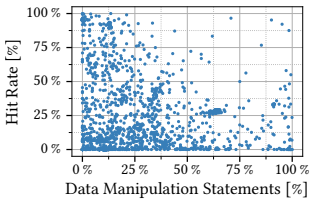


Figure 7: Result cache hit rates depending on the ratio of writes vs. reads.

In the first place, Redshift is an analytical system focused on long-running queries. Nevertheless, updates account for a significant portion of the workload. In particular, insert and copy statements, i.e., loading new data into the database, constitute almost one-fourth of all queries sent by the customer. The impact of queries that modify the database on the hit rate is significant, as shown in Figure 7: For clusters with almost no updates, the result cache answers more than 80 % of the queries. However, if the update rate increases, the hit rate drops significantly; clusters that answer more than 50 % of the queries from the result cache are the exception.

Despite the low hit rate, it is still worthwhile to install the result cache. The lightweight design and negligible build and maintenance overhead make it a good choice for a first caching technique. Besides Redshift, other cloud data warehouses like Snowflake, Databricks, or Google BigQuery also implement result caching.

3.2 Materialized Views

Besides result caching, Redshift implements a second technique to exploit repeating workloads. It automatically identifies repeating query templates in the queries and creates materialized views to cache them. Similar queries that share the same template scan the materialized view instead of recomputing the result every time [2].

Automated materialized views are implemented on top of the database system, exploiting the existing mechanisms to create, store, and update views and use them to rewrite queries originally issued on base tables. This technique only needs three additional components: (1) a mechanism to identify repeating query templates and extracting generalized versions to create materialized views, (2) a cost/benefit estimator that determines the creation and maintenance cost of an MV as well as its performance impact, and (3) an optimizer step that finds eligible views and rewrites the queries to scan the view instead of executing the original statement.

Unlike a result cache entry that is kept on the leader node, materialized views are stored in the cluster's storage and require synchronization with the compute nodes. The maintenance overhead is significantly higher as the view must be refreshed if the underlying data in the base tables changes. Although Redshift implements an incremental view refresh, this is an expensive operation.

With a few dozen views, a single insert, delete, or update operation can trigger multiple refreshes. Redshift employs background threads to perform the refresh and delta updates. However, compared to a result-cache, MVs are still more expensive and creating and keeping the materialized views up-to-date must be carefully weighed against the expected gain.

```sql
-- TPC-H Query 6
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate between date '1994-01-01'
                     and '1994-12-31'
  and l_discount between 0.05 and 0.07
  and l_quantity < 24;
```

(a) Original query.

```sql
-- Materialized View
create materialized view as automv (
  select l_shipdate, l_discount, l_quantity,
    sum(l_extendedprice * l_discount) as revenue
  from lineitem
  group by l_shipdate,
    l_discount, l_quantity);
```

(b) Materialized view created by the system.

```sql
-- Rewritten TPC-H Query 6
select sum(revenue) as revenue
from automv
where l_shipdate between date '1994-01-01'
                     and '1994-12-31'
  and l_discount between 0.05 and 0.07
  and l_quantity < 24;
```

(c) Rewritten query using the mv.

Figure 8: Automated materialized view for TPC-H Q6. Parts of the original query are extracted to create a materialized view. Subsequent queries are rewritten to scan the view. Filters that restrict the result are extracted to improve the hit rate.

Automated materialized views extract a sub-plan from the query and are, therefore, more likely to match than the result cache. In addition, Redshift generalizes the extracted sub-plan to improve the hit rate for the materialized view. For instance, Redshift uses *predicate elevation* to move restrictions that filter the result outside of the materialized view. Figure 8 demonstrates the technique for a query on the TPC-H benchmark. The original query contains three filter predicates on the `l_shipdate`, `l_discount`, and `l_quantity`. The materialized view, created by the system, removes the filter and adds the column to the aggregation to restrict the tuples later. The rewritten query now scans the materialized view and applies the filter on the aggregated results.

AutoMVs are more versatile than result caching and apply to a broader range of queries. Hit rates improve by rewriting and generalizing the cached queries. However, the expected gain is lower, and keeping the materialized views up to date results in a higher maintenance overhead.

3.3 Sorting

Result Caching and AutoMV operate on the query level and cache the results of SQL statements. The main drawback of these techniques is that the cached results are invalidated if the underlying data changes. Besides these two approaches, sorting can be used to cluster the data such that scan filter predicates can eliminate entire blocks. However, sorting by columns in the table is data-driven and adapts the table layout, not the workload. Yang et al. propose query-driven data layouts that co-locate rows according to the workload. Their query-data routing tree (Qd-tree) framework creates a multidimensional index that cuts a relation into disjoint ranges according to filter predicates in the queries [33].

The Qd-tree extracts common filter predicates on base tables and cuts the table into two parts: one that satisfies the predicate and one that does not. The partitions allow the scan to skip entire parts of the table if the customer sends the same predicates again. Figure 9 illustrates this concept for a table with two integer columns, x and y. The table is split into four parts depending on whether the rows satisfy the two predicates `x < 10` and `y > 42`. A scan with the predicates `x < 10` and `y > 42` skips the partitions of the table where `x >= 10` or `y <= 42` and accesses only one of the four parts.

Unlike other techniques, the Qd-tree exploits the repetitiveness of the scan filters and adapts the data layout accordingly. This technique can improve performance significantly for repeating scan expressions as the number of accessed rows reduces. Even more important in the cloud context is that the Qd-tree skips entire blocks. Modern cloud data warehouses store the data in columnar formats

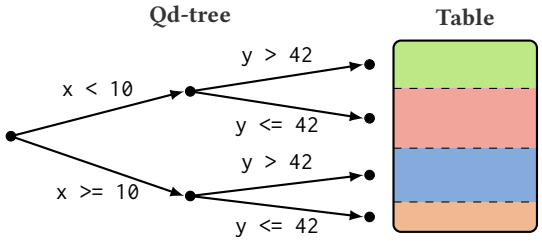


Figure 9: Example of a table split into four parts by a Qd-tree using the predicates `x < 10` and `y > 42`.

and compress them into blocks of a few thousand tuples [11]. The system retrieves these blocks from an external storage system, like Amazon S3. Qd-tree data layouts reduce the number of retrieved blocks and are particularly beneficial for cloud data warehouses.

Another advantage of the Qd-tree is that it does not require a perfect match, such as result caching. For instance, a query that filters on `x < 5` can also exploit the cut from Figure 9 and scan only the two parts where `x < 10`. The Qd-tree supports not only simple predicates that involve a single column but also more complex expressions with multiple columns, and it can integrate sideways information passing [18, 33].

In terms of hit rate, the Qd-tree provides the same benefits as AutoMV; the technique does not require a perfect match and exploits queries with similar patterns. However, the build overhead is now the dominating bottleneck as the Qd-tree reorganizes the entire table. Rewriting large tables with terabytes of data takes several hours. Keeping the index up-to-date is less expensive since new tuples can be merged into the individual partitions of the Qd-tree.

Although inserting or deleting data from the Qd-tree is inexpensive, adapting it to a new workload is not. If the query filter predicates change over time, the Qd-tree will need to select a new set of predicates to compute a more efficient cut. Changing the cut, however, also restructures the table layout, rewriting all partitions. Hence, sorting-based techniques can adapt well to data changes but are less flexible regarding workload changes.

3.4 Objectives

Given these trade-offs, we saw an opportunity to develop a new technique that complements the existing ones with the following design goals:

Online. It is important to keep the cached data up-to-date with minimal overhead. For instance, the result cache invalidates the

cached entries if the underlying data changes. However, it is too expensive to recompute the queries on every change, and it is impossible to keep the cache online.

Lightweight. Minimize the resource usage and ideally keep the cache independent of other nodes in the cluster to reduce synchronization overhead. While the result cache stores all data on the leader node and does not require synchronization with the compute nodes, the AutoMV technique creates a materialized view and shares it with all compute nodes. Hence, the data must be cached outside of one node, resulting in a more involved implementation.

On-the-fly Construction. Building the cache as a side-product of query execution without a separate build phase. The Qd-tree requires a separate build phase to reorganize the table according to the workload. Similarly, creating materialized views triggers an additional transaction to evaluate the cached statement. In contrast, the result cache stores the query output on the first computation and avoids the expensive build overhead.

Data-independence. The cache entries must remain valid even if the underlying data changes. The Qd-tree is the only one of the three techniques that achieves this objective. New tuples do not invalidate the cached entries as they are merged into the existing parts of the Qd-tree. Caching query results, on the other hand, make the cache entries dependent on the underlying data and invalid once the scanned tables change.

Our novel *Predicate Cache* is designed to meet all these objectives. It complements existing techniques like result caches or MVs to take even more advantage of the repetitive nature of real-world workloads. A predicate cache integrates with various cloud data warehouse designs, including data lakes.

4 PREDICATE CACHE

Predicate Caching is inspired by sorting-based techniques that improve query performance by scanning less data. However, instead of reorganizing the data, we keep the table unchanged and remember which parts qualified for a scan. The predicate cache stores the scan's filter conditions and the row ranges that satisfy the filter. If the same scan is executed again, we can skip rows that do not qualify using the cached entries.

4.1 Design

Consider the following query:

```
select count(*) from lineitem, orders
where l_discount = 0.1 and l_quantity >= 40
and o_orderkey = l_orderkey
and o_orderdate between '1995-01-01' and '1995-01-31'
```

It scans the `lineitem` and `orders` table from the TPC-H benchmark and searches for orders in January 1995 that ordered many lineitems with a discounted price. Our predicate cache creates two entries for this query, one for the predicate `l_discount = 0.1` and `l_quantity >= 40` on the `lineitem` table and one for the predicate `o_orderdate between '1995-01-01' and '1995-01-31'` on the `orders` table. The next time the customer sends a query with an identical scan, the predicate cache skips the tuples that do not qualify and accesses only a fraction of the table. For example, only 2 % of the `lineitem` table passes the filter predicate, and only 1 % of the `orders` table. Note that we cache the conjunction of the two predicates on the `lineitem` table, not the individual statements.

We provide two different variants for the predicate cache. Implementation-wise, they are similar and rely on in-memory hash tables to store the entries. They differ in their representation of the qualifying tuples: the first variant allocates a fixed amount of space per worker node, and the second grows the entries with the table size. Both variants are configurable regarding the amount of space they use and their granularity.

4.1.1 Range Index. The first implementation of our predicate cache stores a list of row ranges in a hash table. The table maps a predicate to the list of row ranges; a row range consists of two row ids denoting the qualifying tuples' start and end rows. This simple implementation requires only a hashable version of the filter predicates, e.g., a text representation.

However, the space consumption of this design is not ideal. Consider, for instance, a scan where every second tuple qualifies; in this case, the list contains as many values as rows in the table (2 values per entry). Ideally, we want to limit the ranges to a constant number, e.g., 1024 ranges per cached entry, to minimize space consumption.

We, therefore, merge adjacent ranges to reduce the number of ranges. For instance, the ranges $[1, 2]$, and $[4, 6]$, are merged into a single range $[1, 6]$. Merging the ranges limits the space consumption, reduces the cache's precision, and introduces false positives. Hence, the system must re-evaluate the predicate on the cached rows to eliminate false positives. The merged ranges are computed on the fly while scanning the data. We maintain a heap of the largest gaps where no records qualify. The size of the heap is limited to the number of ranges to store per cached entry. After the scan, the gaps are converted into ranges and inserted into the predicate cache.

4.1.2 Bitmap Index. The second implementation also uses a hash table but maps the keys to a bitmap. Each bit in the map represents a block of tuples, and if a bit is set, the block satisfies the predicate; if not, none of the tuples in the block pass the filter, and the entire block can be skipped. While constructing the bitmap is faster than computing and merging the list of ranges, it is not as precise, and the entries are not as selective. A block in the bitmap index contains typically between 1 000 and 16 000 records.

This is similar to min-max-bounds or small materialized aggregates that also skip entire blocks.

Both implementations are very similar and achieve comparable speedups on the evaluated datasets (cf. Section 5). Yet, there is a small difference in the required space and precision: The range index stores a fixed number of ranges per entry, while the entries in the bitmap index grow with the table size.

Our current implementation caches all predicates pushed into the table scans by Redshift's query optimizer. A cost-based optimizer could decide which predicates to cache based on the selectivity and repetitiveness. In addition, to keep our implementation simple and lightweight, we do not normalize the predicates and store them as strings using the optimizer's representation. SMT solvers can simplify and normalize the predicates into conjunctive normal form (CNF) before caching them, increasing the hit rate [19]. However, our analysis shows that the string representation is already highly

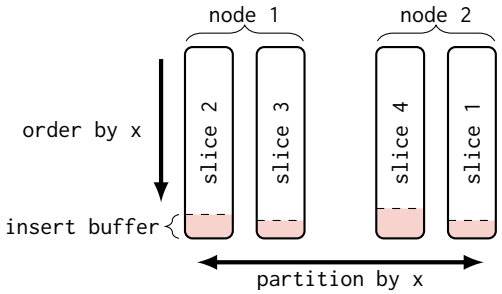


Figure 10: Redshift storage architecture. The table is split into data slices (horizontal partitions) and sorted. The data slices are equally distributed to worker nodes. The insert buffers store newly inserted tuples without sorting them.

repetitive and we do not expect a significant improvement from normalization (cf. Section 2).

4.2 Integration into Amazon Redshift

Redshift is a scalable data warehouse design for the AWS cloud environment. Today, Redshift processes exabytes of data, is fully managed, and scales to hundreds of machines. Several components extended the initial version of PostgreSQL to improve analytical performance. Most notably, Redshift implements a columnar storage engine, optimized compression formats, a distributed and multi-threaded query engine, and a query-compilation framework based on C++.

4.2.1 Storage Architecture. With Redshift's columnar storage engine, several changes were made to the storage layout and data distribution to support distributed query processing efficiently: Redshift splits relations into data slices based on a distribution key. Every data slice can further be ordered by a global sort key. The leader node in a Redshift cluster assigns the data slices to one of the compute nodes; this node is responsible for evaluating the queries on the part of the relation. Figure 10 illustrates this architecture for a two-node cluster with two data slices per node.

The leader node assigns data slices only logically to compute nodes, physically they are stored on Redshift Managed Storage (RMS). RMS makes the slices available to all compute instances in AWS and relies on Amazon S3 for durability. Compute nodes download the data slices from RMS and cache them locally on SSD. In case of a node failure, a new compute node is started and downloads the data slices from S3.

4.2.2 Scan Processing. Redshift splits data slices into smaller compute slices for scanning. Every compute slice is processed by a single thread to leverage multi-core CPUs and consists of compressed per-column blocks. Redshift implements compression techniques like frame-of-reference, run-length encoding, or dictionary compression.

Each compute slice is then passed to Redshift's query engine to extract and decompress the qualifying rows. Redshift uses a two-step scan process: (1) first it eliminates blocks based on the min-max bounds of the predicates, and then (2) the vectorized scan evaluates the filters on the rows using SIMD instructions. The vectorized scan produces a list of row ranges for the qualifying rows to pass on to the next operator in the query plan.

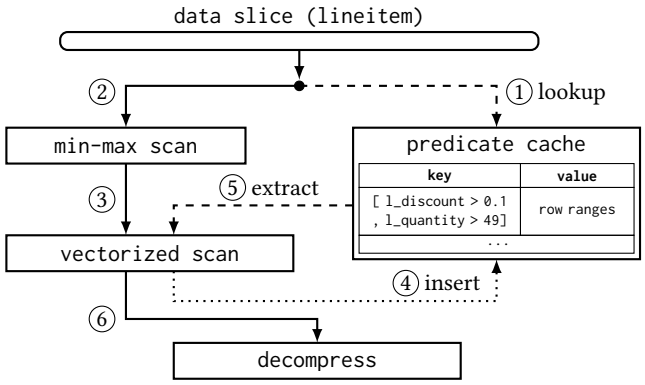


Figure 11: Redshift scan with predicate caching. The solid lines denote the normal scan path, dashed lines accesses to the predicate caching, and the dotted line adds new entries.

4.2.3 Predicate Caching. Redshift's scan process makes it easy to integrate our predicate caching approach. Figure 11 illustrates how we integrate an additional cache into the scan path. In the first step ①, Redshift checks if the predicate cache contains a row range for the current predicate. If the predicate is not cached ②, we use the standard scan process, including a range-restricted scan followed by the vectorized scan to evaluate the predicates. After filtering the data, Redshift produces a list of row ranges for the qualifying tuples ③, which we insert into the predicate cache ④.

If the predicate is found in step ①, we can skip the range-restricted scan and directly use the row ranges from the predicate cache. Although the row ranges are extracted after the vectorized scan, we still pass them into the vectorized scan ⑤ to eliminate false positives. However, this time, the vectorized scan processes fewer rows, as parts of the table are eliminated. After filtering the data, the required columns from the qualifying rows are loaded and decompressed ⑥, and the next operator processes the data.

This design makes the predicate cache a drop-in replacement for the standard scan process. It can easily be turned on or off, and implementing it requires only a few changes in the actual code that performs the scan. No additional work is required to fill the cache and generate the row ranges; the scan already produces the information during the vectorized scan.

4.3 Inserts, Deletes, and Updates

A key advantage of predicate caching is that it is online and easy to build. In particular, the cached entries remain valid even if the customer inserts new rows, deletes, or updates them. As a result, the predicate cache is more versatile than other caching approaches like the result cache or AutoMV.

4.3.1 Inserts. SQL statements that add new data to tables (`insert` and `copy`) are the most common operations after `select` statements on Redshift clusters. Since Redshift is designed as an analytical system for petabyte-scale workloads, adding a large amount of new data is highly optimized: Redshift does not enforce constraints like

primary keys and foreign keys, and no indexes like B-trees are built on the data. Inserting new tuples is fast, even if the data is sorted. If the customer specifies an order key, we add tuples to the end of the table in the insert buffer, which are then merged as part of a `vacuum` process.

The insert buffer for staging new tuples is perfect for the predicate cache. As new tuples are appended at the end, Redshift assigns them higher row numbers. Hence, the row ranges of the cached predicates remain valid, and we do not have to invalidate them. Instead, we remember the row number of the last cached row in the data slice, and newer tuples are scanned using the normal combination of range-restricted and vectorized scans. We can then add the new row ranges to the predicate cache to keep it up-to-date.

4.3.2 Deletes. Like PostgreSQL, Redshift implements multi-version concurrency control based on timestamps. For every row, it maintains a creation and a deletion timestamps that determine the visibility of the row. Deletions set the second timestamp and mark the tuple as invisible for subsequent transactions. The predicate cache effortlessly handles deletions: even if a deleted row is part of a cached row range, subsequent scans eliminate the row during the visibility check. Redshift checks the visibility of a tuple during the vectorized scan and tests if the timestamp of the current transaction is between the creating and deletion timestamp of the row.

Like inserts, the `vacuum` removes the row from the table once it is globally invisible and rewrites the compressed blocks. Tuples are no longer globally visible if no transaction older than the transaction that deleted the tuple exists. As before, `vacuum` changes the row ranges in the predicate cache, and we invalidate the cache.

4.3.3 Updates. Like many OLAP systems with a columnar storage layout, Redshift implements updates out-of-place. The old tuple remains in the relation and is only marked as deleted. The new (updated) version of the tuple is inserted into the relation. By doing so, Redshift implements updates as a delete followed by an insert operation. The predicate caches, therefore, remains valid as none of the two operations changes the row numbers of the tuples.

4.4 Extension as Join Index

For now, we considered the predicate cache only for filtered scans over relations. Redshift, however, also implements *Semi-Join Filters* that push hash joins into table scans on the probe side. Semi-join filters eliminate rows without a join partner early on: they evaluate the join predicate during the scan and test if a tuple is part of the hash table built for the join. Redshift builds a bloom filter [12] while inserting the tuples on the build side of a hash join and passes the filter to the table scan on the probe side. Figure 12 shows the query execution plan for the query from before with a semi-join filter.

Since Redshift evaluates the semi-join filters during the vectorized scan, we can also cache the result. The predicate cache can exploit filters and index the join, i.e., the next time Redshift processes a query with the same join, only the tuples with a join partner are scanned. Thus, the predicate cache combines the benefits of scan-based caching with the ones of a query-based result cache.

However, this comes at a cost: the keys for the cached entry must now include relevant parts of the join and build side to match

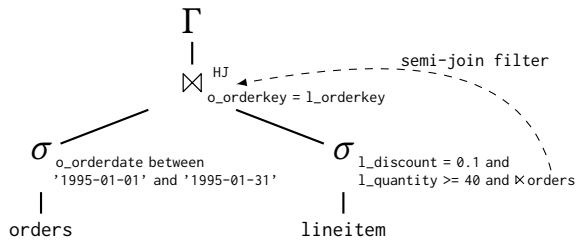


Figure 12: Query execution plan with a semi-join filter. The `lineitem` tablescan probes the hash table built on the `orders` table to eliminate rows without a join partner.

correctly with subsequent expressions. For our example, the full key looks as follows

```
{ table: lineitem,
  filters: [ l_discount = 0.1, l_quantity >= 40,
    { semi_join_filter: o_orderkey = l_orderkey,
      build_side: [{
        table: orders,
        filters: [o_orderdate between '1995-01-01' and '1995-01-31']
      }]
}]}
```

We include the join predicate `o_orderkey = l_orderkey` and a representation of the build side. For the build side of the join, the key stores the scanned table and its filter predicates. With the semi-join filter, the cached entries become 100 times more selective, and only 0.02 % of the rows in `lineitem` are accessed. Entries with and without a semi-join filter are stored in the same cache, and we choose the most selective matching entry.

Combining the predicate cache with semi-join filters also has disadvantages: (1) the hit rate reduces as the cached keys are more complex and also depend on filter predicates on other tables, and (2) entries with a semi-join filter are invalidated by inserts, deletes, and updates on the build side of the join. As modern cloud data warehouses often store the data in a star- or snowflake-like schema, semi-join filters can eliminate many rows early on, and including joins in the predicate cache is beneficial for many queries. Hence, we must weigh the benefits of including the join and reducing the number of rows to scan against the disadvantages of a more complex key and the increased invalidation rate.

4.5 Supporting Open Data Formats

We integrated predicate caching into Amazon Redshift to scan its proprietary relation format stored on Redshift Managed Storage. Modern cloud data warehouses, however, also support scanning and processing open data formats like CSV, Apache Parquet [9], or Apache ORC [8]. Redshift offers the Spectrum accelerator to scan exabytes of data stored in Amazon S3 in these formats [11, 16]. With the recent rise of data lakes in the form of Apache Iceberg [7] or Delta Lake [10], the possibility to scan these new open table formats is a key feature for cloud data warehouse customers.

In data lakes, it is difficult for analytical systems to efficiently index the stored data, as other systems can modify the dataset. For instance, AWS Glue can be used to ingest data in the Apache Iceberg data lake, and a Redshift cluster can read the data and analyze it. The Redshift cluster has no control over the physical layout of the

data and can not reorganize it. However, the predicate cache does not need ownership of the data, and it must only detect changes to the row numbers of the tuples. Both formats, Delta Lake and Apache Iceberg, only add or delete Parquet files to the data lake, and a predicate cache can index the qualifying Parquet files.

Other techniques, like sorting, cannot provide this flexibility. The cloud data warehouse itself must manage the data to rearrange the tuples and generate an optimized data layout. With the option to use predicate caching for Apache Iceberg and Delta Lake, other cloud data warehouse vendors could also easily integrate predicate caching into their systems.

Consequently, predicate caching can be applied to any data format that fulfills the following requirements: (a) individual rows can be uniquely identified, e.g., through a global row id or offsets in external files, (b) the row numbers change only infrequently to avoid frequent invalidations, and (c) changes to the table can be detected to invalidate the cached entries. Additionally, the storage format should support fine-grained access to parts of the table. We observe the highest performance improvements with predicate caching when the database avoids downloading or decompressing entire blocks of data. For instance, in a data lake with Apache Parquet files, the predicate cache could index the row groups in the Parquet files. Non-qualifying groups can be skipped by the query engine, reducing the number of records to load from the data lake.

4.6 Putting It All Together

Predicate caching meets all the design objectives we identified in Section 3.4: It is an online index; inserts, deletes, or updates do not invalidate existing entries. The state is maintained per node, avoiding communication and synchronization with other workers. Redshift constructs the cached entries on the fly while scanning the data and performs no additional work. We avoid data dependencies between cached entries and the actual data, as we only store the row ranges, not the records. Only the join index optimization introduces dependencies to the tables on the build side of the join.

Furthermore, the approach is versatile and integrates well with (external) data formats like Apache Iceberg or Delta Lake. It is workload-driven, indexes only the data queried by the customer, and easily scales to large clusters with more than a hundred nodes. Therefore we believe that predicate caching can further help to take advantage of repetitive workloads and together with result caching, MVs, and (multi-dimensional) sort-keys help to take advantage of repetitive workloads.

5 EVALUATION

We now evaluate the performance of the predicate cache on different benchmarks We investigate the following aspects: (a) the memory consumption of different data-driven indexes and caches, (b) the hit rate of the predicate cache on real-world data sets, (c) the build overhead of the predicate cache, (d) the performance improvements the predicate cache can provide on different benchmarks, and (e) we compare it against predicate sorting, a simplified version of the Qd-tree approach, that organizes rows based on whether they fulfill common predicates in the workload. Like other sorting-based techniques, predicate sorting physically reorganizes the table.

5.1 Experimental Setup

We integrated a prototype of the predicate cache into the latest version of Amazon Redshift in September 2023. We implemented both variants, range-based and bitmap-based indexes. If not otherwise stated, we use the bitmap index implementation with 1 000 rows per block. The experiments are conducted on a 4-node Redshift cluster with ra3.16xlarge instances (48 vCPUs and 384 GiB of memory).

Our experiments use the synthetic TPC-H, TPC-DS, and star-schema (SSB) [30] benchmarks. These benchmarks are commonly used to evaluate the performance of data warehouses and use a variety of analytical queries to answer business questions in a warehouse setting. To represent real-world applications more accurately, we additionally use a skewed version of the TPC-H data generator [3]. We report the end-to-end query execution time as runtime.

5.2 Memory Consumption

Table 3 compares the memory consumption of data-driven indexes and workload-driven caches for query 6 from the TPC-H benchmark. We index the columns `l_shipdate`, `l_discount`, and `l_quantity` used by query 6 to filter `lineitem`. A standard B^+-tree index requires more than half a terabyte of memory to index the 18 B rows. ZoneMaps are significantly smaller, as they only store the minimum and maximum value for every 1 000 rows.

Table 3: Size of different indexes and caches for `lineitem` TPC-H 3 TB. We index the columns `l_shipdate`, `l_discount`, and `l_quantity` used by Q6 to filter `lineitem`.

Category	Type	Size
Sec. Index	B-tree	~ 540 GB
Sec. Index	Zonemap	~ 0.8 GB
Cache	Result Cache	8 B
Cache	AutoMV	42 MB
Cache	Predicate Cache (range)	16 MB
Cache	Predicate Cache (bitmap)	2 MB
Cache	Predicate Sorting	(0 MB)

The caches are significantly smaller than the indexes, with result caching being the most space-efficient, as Q6 only produces a single row with one value. A result cache entry is, therefore, only 8 bytes large. Automated materialized views are also well suited for this query: the three columns contain less than 1.4 M distinct values, and the materialized view stores only four values per row (the three columns for the filter predicates and the result value).

Depending on the used implementation, a predicate cache entry consumes between 2 MB and 16 MB. The range-based implementation stores a list with 16 384 ranges for every data slice (64 slices in total). The bitset index is eight times smaller as it stores a single bit for every one thousand rows. For both implementations, we can adapt the granularity, i.e., the number of ranges per entry or rows per block, and thereby the space consumption. Lastly, we consider a sorting-based approach. Although this technique does not require additional memory, it reorganizes the table, and the database needs to read and write the `lineitem` table that consumes 750 GB.

Although the result caches are small for this example, other queries might require significantly more space if they store multiple

rows with more than one value. The size of the result cache entries and AutoMVs, ultimately, depend on the query sent by the customer. Predicate caching, in contrast, allocates memory proportional to the table size. It is, therefore, easier to manage the entries in the predicate cache as they are all the same size for one table.

5.3 Hit Rate

Next, we evaluate the hit rate of the predicate cache two internal workloads (Workload A & B). Both workloads simulate customer clusters and are based on the query streams we observe in Redshift.

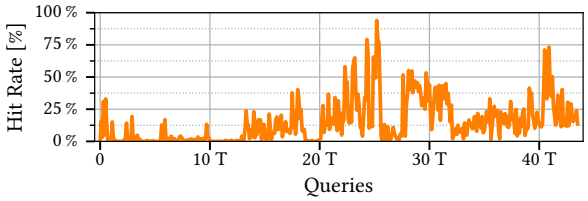
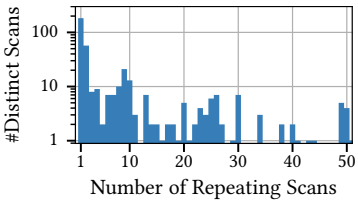


Figure 13: Hit rate of the predicate cache on Workload A.

Figure 13 shows the hit rate of the predicate cache on Workload A over time. The workload executes 44,000 queries in a few hours. The cache is empty at the start of the workload, and the hit rate is low. After processing the first 15,000 queries, the hit rate increases, and more queries can use the predicate cache.



Repetitions	#Scans
[1, 2)	183
[2, 10)	544
[10, 50)	2 002
[50, 100)	473
[100, 300)	768
[1, 300)	3 970

Figure 14: Histogram of the number of repetitions per scan in Workload B. The left plot shows how many distinct scans with the given number of repetitions occur. The right table shows the total number of scans for the given repetitions.

Figure 14 shows how often scans repeat in Workload B. The queries execute roughly 4 000 scans on the dataset, yet we only observe 401 unique scans. Hence, this workload is highly repetitive, and more than 90 % of the scans repeat. Of these 401 scans, 218 run multiple times, and only 183 scans occur once. We can also observe that a vast majority of the scans are highly repetitive: scans that repeat ten times or more account for 3 243 scans, more than 80 % of all scans executed in this workload.

5.4 Build Overhead

A key aspect of the predicate cache is the minimal build overhead. We specifically designed the cache as a lightweight data structure that can be built as a side-product of query processing. In Figure 15, we measure the overhead of adding new entries to the cache. We execute TPC-H and TPC-DS on an empty cache and force every scan with a filter predicate to insert a new entry into the cache. After every query, we clear the cache and repeat the experiment. We also ensure that the scan operator does not use the cached entries in case a query executes the same scan multiple times.

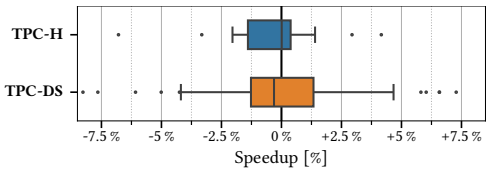


Figure 15: Build overhead of the predicate cache (bitmap).

Most queries observe no significant performance difference, and speedups and slowdowns are within 1 %. For some queries, the runtime fluctuations are more substantial and reach up to 8 %. However, these are isolated cases, and average runtime degrades by less than 0.5 % for both TPC-H and TPC-DS. The overhead of adding new entries to the predicate cache is almost immeasurable, and the actual query execution dominates the query runtime.

5.5 Query Performance

For the remaining part of this section, we focus on the query performance improvements the predicate cache can offer. Table 4 shows detailed statistics for executing the skewed version of TPC-H on Redshift with and without predicate caching. Overall, we observe a 10 % performance improvement with predicate caching. While this performance improvement is not as significant, the number of scanned rows shrinks by almost 4x and the accessed blocks by 30 %.

Runtime speedups mostly correlate with the number of scanned blocks[1]. For instance, the runtime of query 19 improves by 5.2x as the predicate caching reduces the accessed blocks by 7.3x. With the number of accessed blocks, the number of bytes Redshift reads from S3 or the local cache also reduces. The number of rows does not impact the overall performance as much: queries 6 and 19 scan more than 30 times fewer tuples, but query 6 still accesses half of the blocks, and the runtime decreases only by 20 %. Most of the work has to be done before the rows are scanned, i.e., reading the block from disk and decompressing it. Even though we already use a skewed version of TPC-H, the data is still too evenly distributed. We expect real-world data sets to be more skewed, and the predicate cache can further reduce the number of accessed blocks.

The bitmap-based and range-based implementations of the predicate cache perform very similarly. The range-based implementation uses more memory and, thus, eliminates more rows and slightly more blocks. We further investigate the impact of semi-join filters on the predicate cache and evaluate the cache on other benchmarks.

5.5.1 Impact of Semi-Join Filters. Integrating semi-join filters into the predicate cache improves the query performance significantly. We assume that cloud data warehouses often use a snowflake-like schema, where fact tables are filtered and joined with dimension tables. Eliminating rows without a join partner can greatly reduce the number of scanned rows and blocks.

Figure 16 evaluates the impact of caching semi-join filters on the skewed version of TPC-H. Without semi-join filters, the predicate

[1]Blocks accessed is the number of compressed blocks from the data slice the vectorized scan accesses. Rows scanned is the number of records that are decompressed and passed to subsequent operators (cf. Figure 11)

Table 4: Runtime, rows scanned, and blocks accessed for TPC-H (skewed) in different versions. The original Redshift version does not use our predicate cache (PC) or predicate sorting (PS). PC^B and PC^R use the bitmap-based and range-based predicate cache, respectively. Predicate sorting clusters the `lineitem` table according to the most selective predicates in the TPC-H queries.

Query	Runtime [s]				Rows scanned				Blocks accessed			
	Orig.	PC^B	PC^R	PS	Orig.	PC^B	PC^R	PS	Orig.	PC^B	PC^R	PS
1	4.26	4.55	4.55	3.94	6.0 B	5.99 B	5.97 B	6.00 B	13.8 T	13.8 T	13.8 T	43.5 T
2	1.50	1.34	1.34	1.33	2.01 B	1.09 B	1.03 B	1.09 B	13.3 T	12.3 T	12.3 T	12.3 T
3	6.83	6.57	6.81	6.30	7.65 B	5.29 B	4.49 B	3.79 B	21.0 T	20.2 T	19.9 T	32.7 T
4	5.01	5.03	4.98	5.82	7.50 B	5.40 B	4.76 B	3.52 B	20.7 T	17.4 T	17.2 T	34.4 T
5	6.43	5.78	5.58	6.28	7.66 B	2.60 B	1.61 B	3.58 B	39.0 T	35.7 T	34.8 T	64.1 T
6	0.22	0.17	0.17	0.19	6.00 B	325 M	186 M	495 M	8.78 T	5.54 T	5.54 T	13.5 T
7	3.39	3.18	3.17	3.31	7.66 B	2.35 B	1.82 B	2.95 B	41.9 T	34.0 T	33.4 T	41.2 T
8	8.99	7.82	7.95	7.85	7.86 B	510 M	471 M	574 M	46.8 T	10.4 T	10.4 T	16.3 T
9	10.2	9.94	9.96	11.2	8.51 B	4.92 B	4.35 B	7.95 B	55.0 T	48.5 T	47.7 T	100 T
10	6.01	5.88	5.81	5.20	7.65 B	3.12 B	2.22 B	1.32 B	38.2 T	18.7 T	18.4 T	17.4 T
11	0.69	0.63	0.59	0.63	1.62 B	1.21 B	1.00 B	1.21 B	7.98 T	7.67 T	7.66 T	7.74 T
12	3.44	3.22	3.13	3.03	7.50 B	590 M	330 M	1.55 B	41.5 T	21.6 T	21.6 T	19.2 T
13	3.15	3.16	3.11	3.06	1.65 B	1.65 B	1.65 B	1.65 B	78.7 T	78.7 T	78.7 T	78.7 T
14	0.99	0.94	0.95	0.79	6.20 B	236 M	131 M	355 M	13.8 T	5.94 T	5.94 T	12.3 T
15	0.42	0.40	0.39	0.41	6.00 B	516 M	299 M	758 M	514	514	514	514
16	3.03	3.05	3.03	3.02	1.01 B	831 M	831 M	831 M	6.32 T	5.50 T	5.50 T	5.51 T
17	1.01	1.02	1.02	1.04	7.74 B	7.55 B	7.55 B	7.17 B	12.4 T	12.3 T	12.3 T	39.2 T
18	21.4	21.0	21.1	24.2	13.6 B	13.6 B	13.6 B	13.6 B	33.3 T	33.3 T	33.3 T	72.9 T
19	2.36	0.45	0.43	0.26	6.20 B	58.3 M	33.0 M	24.3 M	50.4 T	6.91 T	6.90 T	3.36 T
20	2.49	2.45	2.52	2.36	7.81 B	7.81 B	7.81 B	4.36 B	37.3 T	37.3 T	37.3 T	37.0 T
21	16.0	15.65	15.6	18.6	19.5 B	14.4 B	13.8 B	12.6 B	37.4 T	32.5 T	32.1 T	42.6 T
22	2.18	1.93	1.97	1.81	1.80 B	1.67 B	1.67 B	1.67 B	3.53 T	3.53 T	3.53 T	3.53 T
GeoMean	2.97	2.61	2.6	2.57	5.46 B	1.80 B	1.45 B	1.80 B	19.0 T	13.7 T	13.6 T	19.0 T

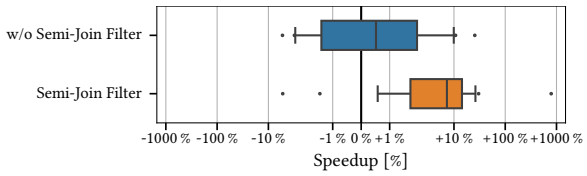


Figure 16: Impact of caching semi-join filters on the query performance in TPC-H (skewed).

cache can only skip a few blocks, the performance improvements are limited, and several queries do not improve at all. Once we also cache the filters, we achieve the 10 % performance improvement we observed in Table 4.

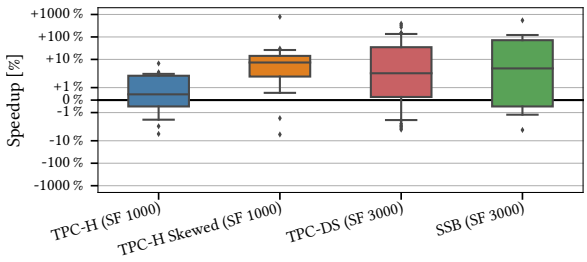


Figure 17: Performance improvements on synthetic benchmarks.

5.5.2 End-to-End Performance. In Figure 17, we evaluate the performance of the predicate cache on other common benchmarks, like TPC-DS and SSB. The results are very similar to the skewed TPC-H benchmark: speedups of more than 100 % are possible, while the bulk of the queries improve by up to 10 %. It's worth noting that these benchmarks have nearly evenly distributed data, making it challenging for selective predicates to eliminate entire blocks. This issue is particularly apparent in the standard TPC-H benchmark, where the data is uniformly distributed, and the predicate cache does not impact the runtime. Based on our observations, we concluded that predicate caching performs better on data sets with a more uneven distribution.

5.6 Predicate Caching vs. Sorting

Table 4 also shows that predicate caching and predicate sorting provide similar performance improvements on the skewed version of TPC-H. Physically partitioning the table based on the predicates reduces the number of scanned rows by two-thirds. However, the number of accessed blocks does not improve unless the query is highly selective. We suspect that the sorted table layout leads to suboptimal data compression, increasing the number of blocks in the table. Redshift uses fixed-size blocks, and a worse compression ratio reduces the number of rows per block. Sorting techniques achieve similar gains as predicate caching, and the overall runtime improvements are slightly larger. We expect sorting to perform better on other workloads with more selective queries or with similar but yet slightly variable workloads.

We also investigate if it is worthwhile to combine these two approaches. However, we do not observe any significant performance improvements (cf. Figure 18). Both approaches provide similar gains, but together, they do not lead to additional benefits on TPC-H.

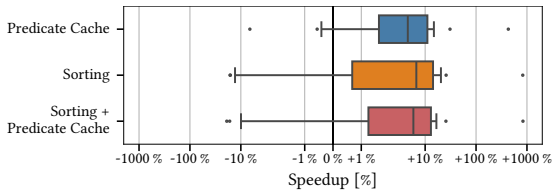


Figure 18: Predicate caching vs. predicate sorting on TPC-H (skewed). The last row combines these two techniques.

For real-world scenarios, combining predicate caching with (multi-dimensional) sorting, like Redshift's new MDDL technique [5], could be beneficial, leveraging the benefits of the two techniques. Sorting the table based on the workload can help skip more blocks and reduce the data retrieval and decompression overhead for cloud data warehouses. However, reacting to changes in the workload is difficult as the table needs to be restructured. Predicate caching bridges this gap and can be used to index the more recent predicates not covered by the sort key. Based on the statistics gathered by the predicate cache, the database can decide to reorganize the table and change the data layout.

6 RELATED WORK

There are multiple techniques and index structures that accelerate scan performance in data warehouses. We categorize them into three groups: data-skipping indexes that eliminate blocks based on predicates, data reorganization techniques that partition the table, and semantic queries that store intermediate or query results.

Data skipping. Small materialized aggregates (SMAs) [29] and ZoneMaps [21] are simple and effective data structures to skip blocks in large-scale data warehouses. They store minimum and maximum values for every column per block (typically a few thousand tuples); the scan operator can eliminate blocks that cannot fulfill the predicates in the query. Most modern columnar databases use this technique to improve query performance, including systems like MonetDB [15], Snowflake [17], Redshift [22], and HyPer [25]. Positional SMAs extend this idea and maintain fine-grained start and end positions within a block to skip individual rows [27].

Besides SMAs, bitmap indexes are commonly used to map values to rows or blocks that contain them. Multiple secondary indexes have been proposed that map individual values or ranges to their positions in a table. The indexes differ in their representation of the data and the granularity. For example, Column Imprints [31], Column Sketches [23], and Bindex [28] rely on bit vectors and index every individual, while the Cuckoo Index [26] uses a cuckoo filter and only indexes blocks to minimize the memory footprint.

New workload-driven approaches index only relevant parts of the table according to the customer queries. The Snowflake Search Optimization Service tracks which values of table columns are found in a micro-partition (large blocks of up to 500 MB of data) [6]. Recently, Boncz et al. proposed the Smooth Predicate Acceleration (SPA) framework that builds secondary indexes on the fly to accelerate scans [14]. The framework identifies predicates and weighs the benefits and costs of building the index. Redshift implements automatic materialized views that generalize customer queries and materialize intermediate results for future queries [2].

Predicate caching shares several design goals with the SPA framework, but it minimizes the overhead even further and exploits the repetitive nature of the queries. We build a more lightweight cache constructed in one step and not in small incremental steps. Compared to Snowflake's Search Optimization Service, the predicate cache can index more complex expressions, conjunctions, and joins using semi-join filters. In addition, Snowflake's search optimization service also requires expensive build and maintenance jobs to keep the indexes up-to-date.

Data reorganization. Sorting the data improves locality and access patterns. Cloud data warehouses, like Snowflake and Redshift, can automatically cluster the data [1, 4]. However, data-driven sorting is limited in the number of dimensions: a sort key usually consists only of a few columns, and the layout is not optimal for all queries. Hence, several papers propose a workload-driven approach that structures the table based on the observed customer queries. The Qd-tree [33] and its successors [5, 18, 19] partition the table into disjoint blocks based on the workload. Similarly, Sun et al. split the table into smaller partitions using feature vectors [32]. The features are derived from representative filter predicates in the workload. However, as outlined in the previous section, sorting data is largely an orthogonal technique that can be used together with predicate caching.

Semantic caching. Cloud data warehouses have a block cache (buffer manager), where each block corresponds to a block from cloud storage. Semantic caching (e.g., Crystal [20]) breaks up this correspondence by reorganizing the cached data according to query predicates. The performance benefits can be significant, especially if aggregations or joins are involved. However, updates invalidate cache entries, reducing hit rates. A predicate cache entry, on the other hand, remains valid for existing data as long as the physical data layout does not change and again complements predicate caches.

7 CONCLUSION

Predicate caching is a lightweight caching technique for large-scale data warehouses. It is query-driven and built on the fly during query processing, avoiding slowdowns at all costs. Updates to the underlying data set do not invalidate the cache, and new data can be integrated incrementally without rebuilding the cache. Its simple design allows for easy integration and support for open data formats, such as Apache Iceberg or Delta Lake.

It bridges the gap between heavyweight caching techniques like materialized views with high build and maintenance costs and lightweight techniques with low hit rates. We show that predicate caching achieves high hit rates on customer workloads, even if tables are frequently updated. Furthermore, performance improvements of up to 10x are possible by including semi-join filters.

ACKNOWLEDGMENTS

We thankfully acknowledge the contributions of the many members of the Amazon Redshift team. We extend our thanks to Jialin Ding, Stefan Gromoll, Adam Hartman, Andre Hernich, Balakrishnan (Murali) Narayanaswamy, Davide Pagano, Pascal Pfeil, Orestis Polychroniou, and Alexander van Renen, all of whom helped realizing this project.

REFERENCES

[1] 2019. *Amazon Redshift introduces Automatic Table Sort, an automated alternative to Vacuum Sort.* https://aws.amazon.com/about-aws/whats-new/2019/11/amazon-redshift-introduces-automatic-table-sort-alternative-vacuum-sort/

[2] 2019. *Automated materialized views.* https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-auto-mv.html

[3] 2021. *Skewed TPC-H.* https://www.microsoft.com/en-us/download/details.aspx?id=52430

[4] 2023. *Automatic Clustering.* https://docs.snowflake.com/en/user-guide/tables-auto-reclustering

[5] 2023. *Improve performance of workloads containing repetitive scan filters with multidimensional data layout sort keys in Amazon Redshift.* https://aws.amazon.com/blogs/big-data/improve-performance-of-workloads-containing-repetitive-scan-filters-with-multidimensional-data-layout-sort-keys-in-amazon-redshift/

[6] 2023. *Snowflake Search Optimization Service.* https://docs.snowflake.com/en/user-guide/search-optimization-service

[7] 2024. *Apache Iceberg.* https://iceberg.apache.org/

[8] 2024. *Apache ORC.* https://orc.apache.org/

[9] 2024. *Apache Parquet.* https://parquet.apache.org/

[10] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[11] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2205–2217. https://doi.org/10.1145/3514221.3526045

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

[13] Peter A. Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Caetano Sauer, Knut Stolze, and Marcin Zukowski. 2023. SPA: Economical and Workload-Driven Indexing for Data Analytics in the Cloud. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 3740–3746. https://doi.org/10.1109/ICDE55515.2023.00302

[14] Peter A. Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Caetano Sauer, Knut Stolze, and Marcin Zukowski. 2023. SPA: Economical and Workload-Driven Indexing for Data Analytics in the Cloud. In *ICDE*. IEEE, 3740–3746.

[15] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.

[16] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. 2018. Integrated Querying of SQL database data and S3 data in Amazon Redshift. *IEEE Data Eng. Bull.* 41, 2 (2018), 82–90.

[17] 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 *SIGMOD Conference*. ACM, 215–226.

[18] 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 Conference*. ACM, 418–431.

[19] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

[20] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.* 14, 11 (2021), 2432–2444.

[21] Goetz Graefe. 2009. Fast Loads and Fast Queries. In *DaWaK (Lecture Notes in Computer Science, Vol. 5691)*. Springer, 111–124.

[22] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD Conference*. ACM, 1917–1923.

[23] Brian Hentschel, Michael S. Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *SIGMOD Conference*. ACM, 857–872.

[24] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[25] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[26] 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.

[27] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD Conference*. ACM, 311–326.

[28] Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, and X. Sean Wang. 2020. BinDex: A Two-Layered Index for Fast and Robust Scans. In *SIGMOD Conference*. ACM, 909–923.

[29] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. Morgan Kaufmann, 476–487.

[30] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC (Lecture Notes in Computer Science, Vol. 5895)*. Springer, 237–252.

[31] Lefteris Sidirourgos and Martin L. Kersten. 2013. Column imprints: a secondary index structure. In *SIGMOD Conference*. ACM, 893–904.

[32] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD Conference*. ACM, 1115–1126.

[33] 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 *SIGMOD Conference*. ACM, 193–208.