PTO: A Workload-driven Predictive Table Optimizer for Lakehouse Systems

VENKATA VAMSIKRISHNA MEDURI, IBM Research-Almaden, USA
DAVID KREISMANN, IBM, Germany
RONALD BARBER, IBM Research-Almaden, USA
BERTHOLD REINWALD, IBM Research-Almaden, USA

Data lakehouse architectures manage both structured and semi-structured data, often using disaggregated storage with volumes that can reach petabyte scale of data stored in open table formats such as Apache Iceberg. Due to the size and storage structure, traditional indexes are cumbersome to maintain resulting in the need for effective table organization to enable efficient retrieval of relevant data for analytical queries. To maximize skipping of irrelevant data while scanning large tables, lakehouse systems rewrite data files according to pre-specified partitioning columns, target file sizes, row group sizes, and bin-packing or sort strategies. Optimizing these parameters can enhance the skipping of irrelevant data during table scans and improve query performance significantly. State-of-the-art lakehouse systems often require these parameters to be manually specified by the user which is impractical due to the combinatorial search space of parameter values thereby severely impeding the usability of the existing table optimization features in these systems.

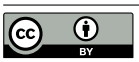
Conducting an exhaustive search to find the best combination of these parameters is impractical because these parameters are interdependent on each other, and rewriting a table with a single instantiation of all the four parameters can already take several hours at terabyte-scale. This comes with the additional complexity that optimal parameter value settings are query workload-sensitive as the filter predicates associated with the scan operators in the workload determine the skipping benefits we can get on a data layout. To overcome these challenges, we built a workload-driven predictive table optimizer for data lakehouses (PTO). It analyzes the filter predicates in the input query workload and reduces the initial search space by utilizing heuristics based on scan selectivity and query frequency in combination with the sort cluster sizes to filter out poorly performing partitioning columns and multidimensional sort column candidates. Thereafter, our PTO solution utilizes table sampling and Gradient Boosting Trees to discover the best combination of table optimization parameters. While our solution is applicable to lakehouse systems and open table formats which adopt similar parameterized layouts, we implemented PTO on Presto lakehouse engine to optimize Apache Iceberg tables. Our experiments show that PTO reduces the average workload latency by 11% on TPC-H and 36% on TPC-DS benchmarks at SF 10K while speeding up scan-intensive, long latency queries by 3.4× and 11× respectively.

CCS Concepts: • **Information systems** → **Physical data models**; • **Computing methodologies** → **Machine learning**; **Classification and regression trees**.

Additional Key Words and Phrases: Lakehouse systems;Table optimization;Data layout discovery;Apache Iceberg;Presto;Velox;Prestissimo;Apache Spark;Z-ordering;Good-Turing heuristic;Iterative sampling

ACM Reference Format:
Venkata Vamsikrishna Meduri, David Kreismann, Ronald Barber, and Berthold Reinwald. 2026. PTO: A Workload-driven Predictive Table Optimizer for Lakehouse Systems. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 67 (February 2026), 26 pages. https://doi.org/10.1145/3786681

Authors' Contact Information: Venkata Vamsikrishna Meduri, IBM Research-Almaden, San Jose, USA, vamsi.meduri@ibm.com; David Kreismann, IBM, Munich, Germany, David.Kreismann@ibm.com; Ronald Barber, IBM Research-Almaden, San Jose, USA, Ron.Barber@gmail.com; Berthold Reinwald, IBM Research-Almaden, San Jose, USA, reinwald@us.ibm.com.



ACM 2836-6573/2026/2-ART67
https://doi.org/10.1145/3786681

1 Introduction

Data lakehouses [41] manage both structured and semi-structured data and one of the primary reasons they can scale up to petabytes and exabytes of data is the decoupling of storage from the computational engine which means that query processing can happen in a separate engine from where the data is actually stored thereby enabling federated and distributed querying over multiple data sources. In particular, lakehouse engines adopt the open table format which is a metadata layer that adds structure on the top of raw data files to organize them into tables [24, 35, 36]. Examples of such open table formats are Apache Iceberg [3, 4, 27], Hive [38], Hudi [1, 2] and Delta tables [6]. Open table formats typically leverage metastores such as the Hive Metastore (HMS) [38] to store the metadata and statistical summaries about tables and schemas at various levels of granularity i.e, file, snapshot and partition-level for faster data access [40]. Data catalogs such as Unity Catalog [11] and Project Nessie [23] allow access to metastores by organizing data into schemas and adding layers of lineage, governance and discovery. Lakehouses can directly connect to semi-structured storage repositories such as Amazon S3 object store [7] and avoid Extract-Transform-Load (ETL) overheads associated with warehouses thus enhancing scalability. However, one of the current bottlenecks plaguing lakehouses is the complexity associated with the usage of traditional indexes. Building and maintaining an index over large-scale open data storage repositories is cumbersome thus requiring an alternative solution to achieve skipping of irrelevant data while running scan-intensive queries. Although open data formats like Hudi [26] support indexes, those are essentially *metadata* indexes built on column statistics and the retrieval method is skipping-based rather than direct pointers to the data used by traditional indexes. Existing works [26, 43] enable skipping of irrelevant data using *min-max* statistics and metadata indexing to accelerate scan-intensive queries. However, organizing the data optimally to maximize skipping benefits is challenging.

In this paper, we explore how to organize the tables in one such open table format, i.e., Apache Iceberg [3], and enable fast retrieval of data which is relevant to the scan filters in a given query workload. Towards this goal, we develop a lakehouse predictive table optimizer (PTO) that takes the query workload and the Iceberg tables as input to automatically detect the optimal layout parameters for each table comprising the partitioning scheme, target file size for the tables stored in a columnar (Parquet [42]) format, row group size of the Parquet files, and a multidimensional sort strategy. Thus, PTO reorganizes the underlying tables to speed up table scans in the query workload akin to what indexes are intended to achieve.

State-of-the-art lakehouses such as Dremio [12, 13], Databricks Delta Lake [17] and Amazon S3 Tables [7] support the OPTIMIZE feature which achieves compaction, i.e., combining several smaller Parquet files created during intermittent ingestion into larger Parquet files whose average file size is below the threshold of target file size. However, the target file size (TFS) of each Parquet file and row group size (RGS) indicating #row groups within a file need to be manually specified. Advanced techniques like AutoComp [25] additionally determine the granularity of compaction tasks to be at a partition or a snapshot-level and recommend identifying TFS from the query workload. Predictive optimization in Databricks [18] decides when to periodically re-optimize the Delta tables. Similarly, Snowflake automatic clustering [14] can also detect when to periodically re-cluster micropartitions but does not identify the sort columns. Databricks *liquid clustering* [20, 30] alleviates this issue by choosing clustering keys and replaces Z-ordering with alternative space-filling techniques like Hilbert curves. Amazon Redshift also automatically determines the sort order of the rows by exploiting the multidimensional filter predicates appearing in the query workload [19], and it does not assume access to underlying data. Our system, PTO, assumes access to underlying data besides the query workload but can holistically configure all the table optimization parameters comprising partitioning column, TFS, RGS and sort strategies. The reason to pick these four parameters to

be automatically configured has to do with the way Iceberg tables are organized. Iceberg tables are organized under partitions similarly to Hive tables but have additional benefits of manifest files and metadata information that support quick pruning of unnecessary partitions, files and row groups if the layout parameters are properly configured. More about the interdependence among these parameters and why PTO co-discovers them will be discussed in Section 2.

A naive exhaustive search on the parameter space is infeasible because the Iceberg tables need to be rewritten and the query workload should be run for each combination of candidate parameter values to discover the best configuration with the least runtime as the optimal table layout. Enumerating the entire search space in this manner can take days or months. Therefore, we solve this problem using a divide-and-conquer approach. We first reduce the overall search space by finding the candidate top-k partitioning columns and sort schemes with the highest filtering capability from the input query workload along with discretized target file sizes and row groups sizes. Next, we train an ML model on the skipping benefits estimated from a tiny fraction of candidate layouts as training data and predict the optimal layout yielding the highest expected skipping benefits. Following are our contributions.

(1) We build a workload-driven predictive table optimizer (PTO) which discovers optimized layout parameters and enables skipping of data irrelevant to the query scan fragments.
(2) We automatically configure four crucial parameters to accelerate scans on Iceberg tables - <partitioning column, target file size (TFS), row group size (RGS), sort strategy>.
(3) Our novel divide-and-conquer strategies detect the top-k partitioning columns and top-k sort schemes in a workload-driven manner. To this end, we rank the power set of sort columns by their filtering capability and sort-cluster sizes because large sort-clusters allow for more data skipping. We leverage sampled tables to efficiently compute the sort-cluster sizes.
(4) We train gradient boosting trees (GBT) on sampled tables to efficiently score imited #training configurations and predict the optimal configuration with the largest skipping ability.
(5) While our solution is applicable to lakehouse systems and open table formats which adopt similar parameterized layouts, we implemented PTO on the top of Presto [34, 37] lakehouse engine for Apache Iceberg tables.
(6) Our experiments show that PTO reduces the average workload latency by 11% on TPC-H [10, 15] and 36% on TPC-DS [16, 31] benchmarks at SF 10K while speeding up scan-intensive, long latency queries by 3.4× and 11× respectively.
(7) In our experiments, we show the latencies of PTO ran in an offline manner; however, to support evolving queries and data, PTO is expected to be run in the background periodically.

2 Preliminaries

2.1 Interdependence among layout parameters

In this section, we describe how the data layout parameters are interdependent which justifies their co-detection.

(1) There is a clear correlation between partitioning column and TFS. If the partitioning column leads to *too many* partitions, then we need to *appropriately* keep the file size *large enough* to ensure that we do not get too many files during table compaction. It is hard to quantify what too many or large enough means since it is dependent on the query workload and the amount of latency incurred by the scan operators during query execution. This is the reason for not relying on manual heuristics for co-discovery of parameters and opting for a predictive model to do so.
(2) We also have a dependency between the sort strategy and RGS. If Z-ordering the table results in several smaller Z-clusters, this can lead to poor skipping benefits when the row group is

relatively larger. For example, if we have 4 to 10 Z-clusters per row group, this can lead to reasonably good skipping benefits. The best case scenario is when Z-cluster size ≥ RGS.

(3) Small row groups result in long Parquet footers which hold the row group level metadata within a file. This in turn results in suboptimal scan latencies as a majority of the scan time would be spent on reading a huge amount of metadata from the Parquet file footers compared to the actual content of the file. This establishes an inherent correlation between TFS and RGS.

2.2 Problem Statement

Given a query workload Q, a set of Iceberg tables to be optimized T, a lakehouse engine L, our predictive table optimization tool PTO determines the optimal *data layout* which is a quadruple of the best combination of table parameters found for each table $t \in T$ i.e., <partitioning column $P(t)$, sort order $S(t)$, target file size $TFS(t)$, row group size $RGS(t)$> such that the average number of rows skipped collectively over T while executing Q on L is maximized.

2.3 Design Goals

Our design goals are motivated by the fact that an exhaustive search for optimal layouts leads to prohibitive costs at TPC-H/DS scale factor 10K (10TB). Without a predictive model, exhaustively evaluating our shortlisted space of 360 candidate layouts (generation of candidate data layouts is discussed in Section 3.2) on a single table takes 360 x (2.74 hours per rewrite + 0.27 hours for scoring) / 24 = 45 days for TPC-DS and 360 x 3.26/24 = 48.9 days for TPC-H on our cluster with 4 machines (specifications are in Section 4.1.3). On a larger experimental cluster, it would be faster but still incurs significant latency. If we score as few as ∼12 candidates, that takes 36 hours for TPC-DS and 39 hours for TPC-H but selects suboptimal layouts from such a tiny search space. Following are our design goals.

(1) In contrast to prior work that optimizes a single, most beneficial data layout parameter such as sorting or clustering, we adopt a comprehensive approach to co-discover multiple data layout parameters. We conjecture that co-discovering these parameters enhances skipping benefits.

(2) While prior work such as Ding et al. [19] does not assume access to the underlying data and purely relies on the query workload, we leverage both the data and the query workload to score layouts with respect to their skipping capabilities on data samples.

(3) To alleviate the prohibitive optimization latencies, we propose a two-step approach which first reduces the candidate search space of layouts and then trains a *lightweight* predictive model to learn skipping benefits from as few as 12 training layouts.

(4) The system is designed to operate on modest computational resources. We leverage a small experimental cluster to discover optimal data layouts for TPC-H/DS tables at scale factor 10K and selected tables from TPC-DS 30K within reasonable optimization latencies.

3 System Architecture

3.1 Overview of PTO

As shown in Figure 1, we use Hive Metastore (HMS) [38] to store all the schema information and metadata including the statistics for the Iceberg tables. We use the Spark-Iceberg library [5] to perform DDL operations such as table creation and alteration, and Spark procedures such as *rewrite_data_files* to apply the best data layout via compaction of Parquet files. We use Prestissimo comprising the Java Presto [34, 37] coordinator along with native C++ Velox [33] workers as the lakehouse engine to run the DML query workload consisting of SELECT queries. We utilize the Presto command line interface (CLI) to capture the query logs automatically generated upon executing the workload. The queries are executed on the original set of tables, termed as full tables

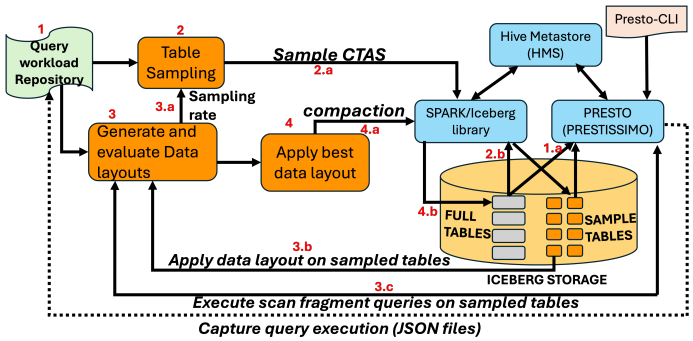


Fig. 1. System Architecture of PTO

(1.a in Figure 1). The auto-generated logs in JSON format are stored in a query workload repository (1 in the figure). PTO consists of three major steps: table sampling, generation and evaluation of data layouts, and applying the best data layout.

- *Table Sampling:* Evaluating a candidate data layout involves rewriting the corresponding table using the layout parameters (3.b in Figure 1), and running the scan fragment queries (3.c) to measure the skipping benefits. Scan fragment extraction is detailed in Section 3.2.1. This needs to be repeated on all candidate data layouts before selecting the best layout. Since this can be time-consuming, we measure the skipping benefits on sampled tables created using a Create Table As Select (CTAS) statement via the Spark-Iceberg library (2.a and 2.b in Figure 1).
- *Generate and evaluate data layouts:* To avoid an exhaustive search, we follow a two-step approach. In the first step, we generate candidate data layouts by shortlisting top-k partitioning columns and sort orders which have the highest filtering capability and can benefit the largest number of queries (termed as filter column frequency). In the case of multidimensional sort columns, we bias towards column sets which tend to generate large sort clusters and yield high skipping benefits. We leverage the Good-Turing estimator [29, 32] to determine the sampling rate (3.a in Figure 1) and efficiently estimate sort cluster sizes on sampled tables. We also discretize TFS and RGS to create a candidate search space of data layout quadruples. In the second step, we divide the search space into train and test candidates and learn a regression model (gradient boosting trees - GBT) from the skipping benefits measured on sampled tables (3.b and 3.c in Figure 1) as target labels. To achieve modest latencies for label generation, we keep the training set significantly small (~3-4% of the candidate space).
- *Apply the best data layout :* We use the trained GBT to predict the best layout with the highest skipping benefits from the overall candidate space. The original set of tables are optimized using discovered layout through compaction (4.a and 4.b in Figure 1) .

3.2 Generation of Candidate Data Layouts

In this section, we describe the first step in our two-step solution which is the generation of candidate data layouts.

3.2.1 Extraction of Scan Fragments. We parse the JSON logs of the query workload and extract the scan fragments which contain the following information for each scan operator appearing in the query - fragment ID to uniquely identify each scan fragment appearing in the query, name of the scan operator indicating whether it is a TableScan or a FilteredScan, table name, filter predicate, filter columns, table cardinality, rowsOut indicating the #rows emitted by the scan operator, and

selectivity computed as $\frac{rowsOut}{tableCardinality} \times 100.0\%$. Here, rowsIn refers to #rows going into the scan operator and it is different from tableCardinality because the skipped rows are excluded from rowsIn. If the table is partitioned, rowsIn includes the rows from relevant partitions. Furthermore, the effect of file skipping and row group skipping i.e., excluding row count from irrelevant files and row groups is also expected to be captured while computing rowsIn. An example scan fragment is listed in Table 1. Note that a scan fragment may include predicates pushed down by the optimizer from higher-level operators in the query plan.

Table 1. Sample Scan Fragment on the TPC-H lineitem table

Key	Value
fragmentID	0
scanOp	ScanFilterProject
filterPredicate	(CAST(shipdate AS timestamp)) >= (TIMESTAMP'1998-03-01 00:00:00.000')
filterColumns	['shipdate']
tableCardinality	5999989709
rowsIn	5999989709
rowsOut	538678298
selectivity	8.98%

We use the extracted scan fragments to create scan fragment queries for each table. An example query based on Table 1 for TPC-H lineitem is SELECT SUM(l_quantity) from lineitem WHERE l_shipdate >= date_parse('03/01/1998','%m/%d/%Y');. These queries are used to evaluate the candidate data layouts on sample tables. Additionally, for each table, we also record #queries in which a column appears as a filtering column in its scan fragments. We term this as *filter column frequency*. We also compute the average scan selectivity across all the scan fragments a filter column appears in. Note that we are interested in the scan fragment selectivity and not the individual filtering capability of the column. This is because there may be filter predicates on other columns and their conjunction (or disjunction) with the predicates on the current filter column of interest contributes to the overall scan selectivity. Regardless of the individual filtering capability of the column, its presence in multiple highly selective scan fragments across the entire query workload is deemed valuable.

3.2.2 Top-k Partitioning Columns. Based on the scan fragments extracted from the query workload, we shortlist the top-k categorical columns ('INT', 'VARCHAR' and 'DATE' types are allowed) with the highest filtering capability. To this end, for each candidate partitioning column, we determine the filtering capability based on its filter column frequency which is #queries in which the column appears in the scan filter predicate, and the average scan selectivity across all the scan fragments in queries where the column appears as a filtering column. We perform a <major, minor> sort of all the categorical columns in descending order upon <filter column frequency, (100 - average scan selectivity)%> and pick the top-k columns. We provide guardrails to avoid too few or too many partitions by providing configurable parameters for the expected number of minimum and maximum partitions, and the minimum cardinality of the table for it to be eligible for partitioning. Our default settings for all these parameters including top-k depend on the expected size of the candidate data layouts and are listed in Section 4.1.3. Besides the top-k partitioning columns shortlisted, we always include NULL as an extra candidate to evaluate the absence of a partitioning column in which case the table is not partitioned. PTO estimates #partitions for a partitioning column from the number of distinct values (NDV). We compute NDV from value_counts and null_value_counts in tablename$files which is an Iceberg metadata table for the current snapshot. We leverage record_count and file_size_in_bytes to estimate table cardinality and file sizes.

Algorithm 1 getCandSortSchemes(queries, table, colSet, top$_k$, sampleRate$_{min}$)

```
1: filterColFreq ← getFilterColFreq(colSet, queries)
2: avgScanSel ← getAvgScanSelectivities(colSet, queries)
3: shortListedCols ← sortColSet(colSet, filterColFreq, avgScanSel, DESC, threshold)
4: if |shortListedCols| == 0 then
5:    return "BIN-PACKING"
6: else
7:    colPowerSet ← getPowerSet(shortListedCols)
8:    avgDistSortClustSizes ← getAvgDistSortClustSizes(sampleRate_min, table, colPowerSet)
9: end if
10: scores ← scorePowerSet(colPowerSet, avgDistSortClusterSizes, filterColFreq, avgScanSel)
    {normalized equi-weight score}
11: sortColList ← arg_max_cols(scores)[:top_k]
12: candSortSchemes ← {}
13: for candCols ∈ sortColList do
14:    if DECIMAL ∈ candCols.dataTypes then
15:       candSortSchemes.append("SORT", candCols) {major-minor multidimensional sort}
16:    else
17:       candSortSchemes.append("SORT", ZORDER(candCols)) { Z-ordering}
18:    end if
19: end for
20: return candSortSchemes
```

3.2.3 Top-k Sort Schemes. Algorithm 1 describes how candidate sort schemes are obtained for a table. For a given table with a set of columns *colSet*, we first shortlist columns with high filtering capability by sorting *colSet* on its filter column frequency and average scan selectivity similarly to the partitioning columns and imposing a pre-specified *threshold* (lines 1-3 in the algorithm). The default *threshold* value is listed in Section 4.1.3. If the *shortListedCols* is empty, we return bin-packing as the candidate sort scheme (line 4).[1] Otherwise, we compute all possible $2^{|shortListedCols|}-1$ subsets of the set of shortlisted columns using the standard power-set definition (line 7). Next, we obtain the average distinct sort cluster size for each column subset in the power set (line 8) and score the subset using a weighted combination of the normalized values of average distinct sort cluster size, filter column frequency and the average scan selectivity by assigning equal weights to all the three scoring criteria (line 10). We omitted an algorithm for *scorePowerSet* in the interest of space constraints because it is fairly straightforward. Finally, we pick the top-k sort column sets with the highest score (line 11). Note that we compute average distinct sort cluster size because simply computing the average over sort cluster sizes will bring very low sort cluster sizes due to the presence of a long tail of small clusters. Skipping ability is maximized by the larger clusters and therefore, we use average distinct sort cluster sizes for scoring. We either apply major-minor ordering or use Z-ordering for multidimensional sort depending on whether a "DECIMAL" column exists among the candidate sort columns (lines 13-18) because Z-ordering on DECIMAL data type is not supported by Spark-Iceberg [5].

3.2.4 Computing the Average Distinct Sort Cluster Sizes. As listed in line 8 of Algorithm 1, the average over distinct sort cluster sizes is obtained for each subset comprising the candidate sort

[1]Bin-packing is cheaper to execute than sorting but PTO does not include final compaction times while scoring the sort strategies. A possible extension to scoring mechanism could combine estimated compaction latencies with skipping benefits.

Algorithm 2 getAvgDistSortClustSizes(sampleRate$_{min}$, table, colPowerSet)

```
1:  avgSortClustSizes ← {}
2:  for sortCols ∈ colPowerSet do
3:    sampleRate_cur ← sampleRate_min
4:    while true do
5:      sample ← getBernoulliSample(sampleRate_cur, table)
6:      distClustSizes_samp ← getDistinctClustSizes(sample, sortCols)
7:      avgDistClustSize_samp ← ∑ distClustSizes_samp / |distClustSizes_samp|
8:      numSingletons ← getSingletons(distClustSizes_samp)
9:      scaleUpRate ← min(|distClustSizes_samp| / (|distClustSizes_samp| − numSingletons), |table| / |sample|) {Good-Turing Heuristic}
10:     numDistClust_orig ← min(scaleUpRate × numSingletons + |distClustSizes_samp| − numSingletons, |table|)
11:     δ ← |table| / numDistClust²_orig
12:     avgDistClustSize_orig ← avgDistClustSize_samp × (δ × |table| / |sample| + (1 − δ) × numDistClust_orig / |distClustSizes_samp|)
13:     if ( numSingletons / |distClustSizes_samp| < 0.95 OR |sample| / |table| > 0.95) then
14:       avgSortClustSizes[sortCols]← avgDistClustSize_orig
15:       break
16:     else
17:       sampleRate_cur ← 10.0 × sampleRate_cur
18:     end if
19:   end while
20: end for
21: return avgSortClustSizes, sampleRate_cur
```

columns in the power set. An example aggregate query to compute the average distinct sort cluster size on a store_sales sample for a candidate set of sort columns can be written as "SELECT AVG(DISTINCT sortClustSize) AS *avgDistClustSize$_{samp}$* FROM (SELECT COUNT(*) AS sortClustSize FROM *store_sales* TABLESAMPLE BERNOULLI(*sampleRate*) GROUP BY *ss_sold_date_sk*, *ss_store_sk*);". Sampling is required because running this query can incur a non-trivial latency on the entire table. There are two challenges here: (1) setting the sample rate, and (2) scaling up average distinct sort cluster size obtained on the sample table to the original table. A naive usage of $\frac{avgDistClustSize_{samp}}{sampleRate}$ to estimate $avgDistClustSize_{orig}$ leads to large errors. Figure 2 shows # distinct clusters on the sample ($|distClustSizes_{samp}|$) and the average distinct sort cluster size ($avgDistClustSize_{samp}$) at three different sampling rates of 0.001%, 0.01% and 0.1% in a toy example limited to 5 groups. We can see a long tail of singleton clusters at lower sampling rates (4 out of 5 clusters at 0.001%) which gets relatively shorter with larger sampling rates (1 out of 5 clusters at 0.1%). This is because most of the singleton clusters are not *truly* singletons. Those groups are just undersampled at lower sampling rates leading to smaller group sizes. We correct this by appropriately scaling up $avgDistClustSize_{samp}$ using a Good-Turing heuristic [29, 32] to iteratively estimate $avgDistClustSize_{orig}$ by increasing the sample rate until convergence.

Algorithm 2 breaks down the estimation of average distinct sort cluster size into two parts. First, the number of distinct clusters is estimated followed by the computation of average cluster size.

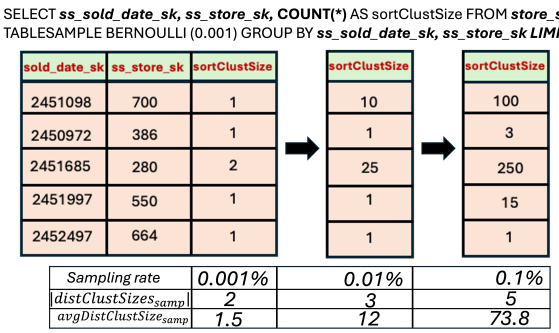


Fig. 2. Sort Cluster Sizes for sampled TPC-DS store_sales

Note that we can also use APPROX DISTINCT in Presto [21] (instead of Good-Turing) to estimate the approximate count of distinct sort clusters. In any given sampling iteration, $distClustSizes_{samp}$ and $avgDistClustSize_{samp}$ (lines 6 and 7 in Algorithm 2) are computed on a Bernoulli sample of the table created using the current sample rate sampleRate_{cur}, which is initialized at a configurable parameter sampleRate_{min}. Then, the Good-Turing heuristic [29, 32] determines the bloating factor or the $scaleUpRate$ to estimate the actual distinct count of these clusters (which are *falsely* appearing as singletons due to lower sampling rate, as shown in figure 2) in the original table (line 9). This gives us the overall count of *singleton* clusters in the entire table as $scaleUpRate \times numSingletons$ while not bloating the count of the non-singleton clusters (line 10). The reasoning here is that non-singleton clusters are sufficiently sampled compared to singletons which is also reflected in their distinct cluster count and hence, we need not overestimate their count. Once we get the number of estimated distinct clusters of all sizes on the original table ($numDistClust_{orig}$), we divide it by $|distClustSizes_{samp}|$ to get the scale-up rate and multiply it with the estimated average distinct cluster size on the sample, $avgDistClustSize_{samp}$ (line 12) to get $avgDistClustSize_{orig}$. We in fact use a weighted combination with another term $avgDistClustSize_{samp} \times \frac{|table|}{|sample|}$ to account for errors in the estimation of $numDistClust_{orig}$ (line 12). Finally, we stop refining $avgDistClustSize_{orig}$ when at least one of the two convergence criteria is met: (a) the percentage of singletons is below a threshold (95% being an acceptable long tail), or (b) the sampling size is close to the full table cardinality (> 95% of |table|). For all the enumerated candidates of column combinations in the power set of columns, we observe case (a) in practice. Case (b) is included for the sake of algorithmic completeness. The reason for observing case (a) consistently is that 95% tail is long enough for sort cluster size estimation using Good Turing heuristic to converge earlier (i.e., below 0.1% upper bound of sample rate for all the SF 10K tables).

3.2.5 Discretizing TFS and RGS. We set the lower and upper bounds of target file size (TFS) as configurable parameters in PTO, with default values listed in Section 4.1.3. We discretize $[TFS_{min}, TFS_{max}]$ by choosing all powers of 2 such that $\forall x$, if $TFS_{min} \leq 2^x \leq TFS_{max}$, we add 2^x MB to the list of candidate TFS. For each candidate TFS, we obtain a set of candidate row group sizes (RGS) as $\frac{TFS}{2 \times i}$ where i is an integer and $i \in [1, 5]$ and additional candidate RGS=TFS with one row group per file. Thus, we allow between 1 and 10 row groups per Parquet file. As mentioned earlier in Section 2.1, too many row groups will lead to long metadata footers in a Parquet file which in turn lead to poor scan latencies. While we discretized file sizes, we also compared with a Bayesian hyperparameter optimization (HPO) baseline that uses continuous TFS values.

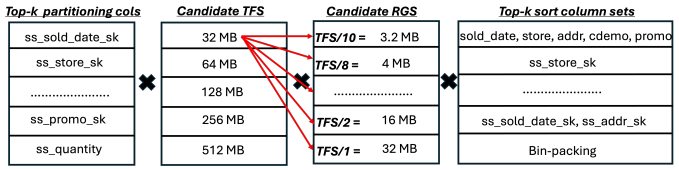


Fig. 3. Candidate Data Layouts for TPC-DS store_sales

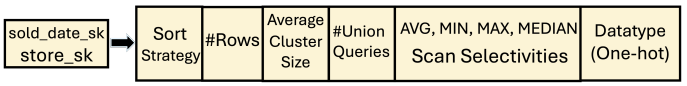


Fig. 4. Feature representation for a sample sort scheme

3.3 ML-based Detection of Best Data Layout

Figure 3 shows an illustration of the candidate space of layouts created for the TPC-DS store_sales table. Note that we included the option of not partitioning in the top-k candidate list of partitioning columns. We use 32MB and 512MB as the minimum and maximum TFS bounds while the candidate RGS are shown only for 32MB by dividing a Parquet file into 1,2,4,6,8,10 row groups. The sort strategies can either be major-minor sort or Z-ordering on a shortlisted column set or can use bin-packing. Even with a handful of shortlisted top-k candidates for each table optimization parameter and the search space containing only a few hundreds of data layouts, it can take a couple of months to perform an exhaustive search. To avoid this, we leverage ML models to find the best data layout among the shortlisted candidate layouts. This forms the second step of our two-step solution described in Section 3.1.

Algorithm 3 estimateSkippingBenefits(dataLayout, scanFragQueries, sampleTable, table)

```
samplingRate ← |sampleTable| / |table|
downscaledTFS ← samplingRate × dataLayout.TFS
downscaledRGS ← samplingRate × dataLayout.RGS
if dataLayout.partitionCol != NULL then
    sampleTable ← CTAS(sampleTable, dataLayout.partitionCol) {partitioned copy of sampleTable}
end if
sampleTable ← rewrite(sampleTable, dataLayout.sortScheme, downscaledTFS, downscaledRGS)

skippedRowCount ← getNumRowsSkipped(sampleTable, scanFragQueries)
return skippedRowCount
```

3.3.1 Feature Vector Creation. We represent each data layout as a feature vector which is a concatenation of the numerical representation for each of the four components - partitioning column, TFS, RGS, sort columns. While TFS and RGS are used as they are in the feature vectors, deriving the features for partitioning column and sort columns is non-trivial. Figure 4 shows how the feature representation is derived for a sort scheme. We first represent the sort strategy as 0/1/2 depending on whether we use bin-packing, major-minor multidimensional sort or Z-ordering. This is followed by the dimensions for table cardinality, average distinct Z-cluster size, #queries which contain

one or more sort columns in their filter predicates (#UnionQueries), and the average, minimum, maximum and median of the scan selectivities from scan fragments where at least one sort column appears as a filter, and the data types of the filter columns which are represented as a one-hot vector. The #dimensions in the one-hot vector is equal to #distinct data types of all the columns appearing in the table schema (note that a separate instance of the ML model is learned for each table). For multiple sort columns, we use AVG as the pooling technique to obtain the value of each feature dimension. In the case of bin-packing, there are no sort columns and hence, all the remaining dimensions in the sort vector are represented as 0. For the partitioning columns, we exclude the sorting technique dimension while retaining all the other dimensions shown in Figure 4 intact.

3.3.2 Creation of Target Labels for Training set. Section 3.3.1 details how the feature vectors are created but not the target labels. In this section, we discuss how we label the training set of feature vectors. Even a training set of layouts as small as ~3-4% of the entire candidate space incurs huge latencies to be evaluated on the original set of tables. Therefore, we run scan fragment queries on sampled tables to compute the skipping benefits of a data layout. Algorithm 3 details how the skipping benefits are estimated for an input data layout on its corresponding sample table. We downscale TFS and RGS by multiplying them with the convergence sampling rate from Algorithm 2 in Section 3.2.4 (which we use to create sampled tables) to obtain downscaledTFS and downscaledRGS (lines 1-3). If the partitioning column is non-null, we create a partitioned copy of the sample table (lines 4-6) using Create Table As Select (CTAS) on the original sample table. Next, we rewrite the sampled table using the sort scheme, downscaledTFS and downscaledRGS (line 7). Finally, we examine the EXPLAIN ANALYZE plan output from each of the scan fragment query workload to measure the skipping benefit as $(1.0 - \frac{scanFragment.rowsIn}{scanFragment.tableCardinality}) \times 100.0\%$ and it is used as the target label for the feature vector of the corresponding input data layout. Next, we train a gradient boosting tree (GBT) as a regression model on the labeled set of data layouts and use it as the prediction model to predict the skipping capability on the ~96-97% unlabeled set of data layouts. Finally, the best layout is applied on the original set of input tables to optimize them and yield the best possible skipping benefits on the input query workload. We also evaluated PTO on alternative ML baselines based on hyperparameter optimization (HPO) (Refer to Section 4.1.2).

4 Experiments

4.1 Experimental Setup

4.1.1 Datasets. We evaluate PTO on TPC-H [10, 15] and TPC-DS [16, 31] benchmarks and the corresponding query workloads at scale factor 10,000 (SF 10K)[2]. We feed the 103 queries from TPC-DS (Q14, Q23, Q24 and Q39 are two-part queries among the 99 TPC-DS queries) and 22 queries from TPC-H as the input workload for which the layouts are optimized for the tables in these benchmarks.

4.1.2 Baselines. For the end-to-end evaluation of PTO (Section 4.2), we use the following baselines:
1. Default (Def_{opt}) : This is an implementation where the tables are not automatically optimized. The partitioning columns are borrowed from most common choices made by database engines for which configurations are posted on the public benchmark repositories [15, 16]. Among the TPC-DS tables, we partitioned *store_sales, catalog_sales* and *web_sales* on *ss_sold_date_sk, cs_sold _date_sk* and *ws_sold_date_sk* respectively. On TPC-H, we partitioned the *lineitem, orders, customer* and *part* tables on *l_shipdate, o_orderdate, c_mktsegment* and *p_brand* respectively. For TFS and RGS, we used the default sizes of the pre-created Parquet files from the dataset which we ingested into

[2]The table sizes shown in Table 2 may not add up to 10TB as expected for an SF 10K dataset due to compressed Parquet files in the Iceberg format. Same applies to the TPC-DS SF 30K table sizes in Table 6.

Iceberg tables. We did not specify any sort columns because that would require the suggestions from an automatic tool.

2. PTO using a human expert ($PTO_{HumanExp}$) : Since PTO co-optimizes the layout parameters using an ML model, we compare it with this baseline which instead uses a human expert. To this end, we re-use the top-k candidate generation mechanisms that we discussed for partitioning columns and sort schemes in Sections 3.2.2 and 3.2.3. We model the co-dependencies between parameters using a human expert who empirically comes up with heuristics detailed in Algorithm 4 while feeding each of the top-k candidate partitioning columns, $partitionCol$, as input.

Algorithm 4 estimateFileSizes($table$, $partitionCol$, TFS_{min}, TFS_{max}, RGS_{min}, RGS_{max}, $files_{min}$, $files_{max}$, $partFiles_{min}$)

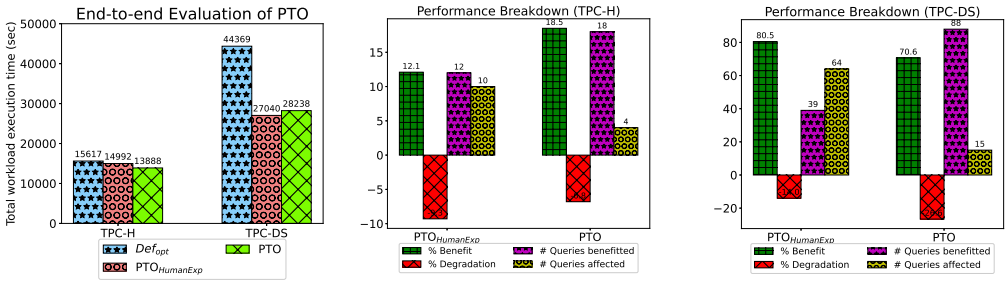
1: $parts \leftarrow partitionTable(table, partitionCol)$
2: $TFS \leftarrow \frac{table.size}{min(max(parts, files_{max}), max(partFiles_{min} \times parts, files_{min}))}$
3: $TFS \leftarrow \min(TFS_{max}, \max(TFS_{min}, TFS))$
4: $RGS \leftarrow min(RGS_{max}, \frac{TFS}{4}, max(RGS_{min}, 100K \times \frac{table.size}{|table|}))$
5: **return** (TFS, RGS)

(1) This baseline, $PTO_{HumanExp}$, models the dependency between the partitioning column and TFS by ensuring that there are at least $partFiles_{min}$ files per partition if the table is partitioned or at least $files_{min}$ files if non-partitioned (line 2 in Algorithm 4).
(2) The number of files is capped at $files_{max}$ if the total number of partitions, $parts \leq files_{max}$ (line 2).
(3) The target file size TFS is computed by dividing the size of the table in MB by the number of expected files (line 2).
(4) We ensure that $TFS_{min} \leq TFS \leq TFS_{max}$ (line 3).
(5) We fit at least 100K rows in a row group or have 4 row groups in a file (line 4).
(6) We empirically set $files_{min}$, $files_{max}$ and $partFiles_{min}$ to 30, 10K and 10 respectively. TFS_{min}, TFS_{max}, RGS_{min} and RGS_{max} are set to 32 MB, 512 MB, 8 MB and 128 MB respectively.
(7) Finally, it scores the pre-identified $partitionCol$, TFS and RGS in conjunction with the top-k candidate sort schemes (binpacking is included when #candidates < k) on the sampled tables by computing the skipping benefits on each candidate layout to pick the best possible layout. It measures the skipping benefits using the downscaling method described in Section 3.3.2.

The baselines for the ML model, sort cluster size detection and candidate sort scheme discovery are discussed in Sections 4.3.1, 4.3.2 and 4.3.3. A weighted PTO baseline is evaluated in Section 4.3.4.

4.1.3 Configurations and Settings. We conducted our experiments on a cluster of four machines with an aggregate 0.7 TB RAM and a total of 320 CPUs (80 physical cores with a hyperthreading factor of 4) of 2 GHz Intel(R) Xeon(R) Gold 6138 running Red Hat Enterprise Linux 9.5. We used MinIO for S3 storage and implemented PTO using Python 3.7.3 and Java with openJDK 17.0.13. We used scikit-learn 0.24.2 and HyperOpt [8, 9] to implement K-medoids and HPO algorithms like genetic optimization and Bayesian optimization using Tree Parzen Estimators (TPE) and random search. The parameters for PTO are discussed below.

We set a conservative value of k=3 for the generation of top-k candidate partitioning columns and sort schemes. The purpose of candidate generation is to create a compact search space of data layouts for quick evaluation. Even with k=3, an exhaustive evaluation takes several days which makes larger values of k impractical. To generate candidate partitioning columns, we ensure that



(a) Quality of layouts (b) Quality breakdown (TPC-H) (c) Quality breakdown (TPC-DS)

Fig. 5. PTO vs. PTO$_{HumanExp}$ and Def_{opt} on TPC-H/-DS workloads along with the performance breakdown.

the table has at least 10K rows (which is again a conservative estimate that leads to evaluating partitioning in almost all the cases) and only those columns which lead to at least 10 partitions and a maximum of 4,000 partitions are treated as candidate columns. Note that we also include "No Partitioning" as an additional candidate (k=4) to evaluate the effect of not partitioning the table. While generating the top-k candidate sort schemes using Algorithm 1, we set the query frequency threshold to 5% to ensure that any column appearing in very few queries is pruned away. We set TFS$_{min}$ to 32 MB and TFS$_{max}$ to 512 MB leading to [32, 64, 128, 256, 512] as the discrete candidate set of TFS following the discussion from Section 3.2.5. For each candidate TFS, we have an RGS candidate set of $\frac{TFS}{2\times i}$ with i∈[1,5] and an additional candidate of RGS = TFS i.e., a single row group per file. Top-4 partitioning columns, top-3 sort schemes, five candidates for TFS and six RGS candidates result in a test set of 360 candidate data layouts. The training set for the ML models is created from the top-k candidate partitioning columns and sort schemes along with random choices for TFS and RGS leading to 12 data layouts (~3.2% of candidate space). We use them as fixed training and test sets for all our experiments.

4.2 End-to-end Evaluation

Here, we evaluate PTO end-to-end w.r.t. the entire query workload of TPC-H/DS at SF 10K.

4.2.1 Quality of the layouts discovered by PTO. In order to evaluate the quality of PTO, we run the query workload on the full set of tables in TPC-DS and TPC-H SF 10K benchmarks which are rewritten using the optimized layouts discovered by PTO. Note that we list the optimized layouts for all the 8 tables in TPC-H but in TPC-DS, we list the optimized layouts for the fact tables which are relatively large, i.e., sales, returns and inventory tables. Compared to the baseline of default layouts, Def_{opt}, PTO achieves 11% savings in latency over the end-to-end execution of the 22 TPC-H queries and 36% savings upon the 103 TPC-DS queries. This can be noticed in Figure 5a (13.8K seconds with PTO vs. 15.6K seconds using Def_{opt} on TPC-H and 28.2K seconds vs. 44.37 seconds on TPC-H). The reason for this improvement is evident from the optimized layouts shown in Table 2. All the tables are unsorted (use binpacking) in Def_{opt} and only a few large tables are partitioned such as the sales tables in TPC-DS and lineitem, customer, orders and part in TPC-H. In contrast, PTO uses candidate layouts in combination with gradient boosting trees (GBT) to co-discover optimal partitioning columns, TFS, RGS and the sort scheme. Z-ordering is predominantly used to sort on columns with large sort cluster sizes while benefitting maximum #queries.

Table 2. Optimized layouts configured by Def_{opt}, $\text{PTO}_{HumanExp}$ and PTO for the tables in TPC-H and TPC-DS.

Dataset	Table	Total Size	Def_{opt}				$\text{PTO}_{HumanExp}$				PTO			
			Partitioning Column	TFS (MB)	RGS (MB)	Sort Scheme	Partitioning Column	TFS (MB)	RGS (MB)	Sort Scheme	Partitioning Column	TFS (MB)	RGS (MB)	Sort Scheme
TPC-H (SF 10K)	customer	74.5 GB	c_mktsegment	150	75	binpack	c_nationkey	305	51	(c_mktsegment,c_acctbal)	c_nationkey	256	64	(c_mktsegment,c_acctbal)
	lineitem	1.6 TB	l_shipdate	343	114	binpack	l_commitdate	165	28	(l_shipdate,l_returnflag, l_shipmode,l_shipinstruct, l_discount)	l_shipdate	512	128	z-order(l_returnflag, l_shipmode,l_shipinstruct)
	nation	2.4 KB	-	32	8	binpack	-	32	8	n_name	-	32	8	n_name
	orders	412.4 GB	o_orderdate	14	14	binpack	o_orderdate	42	11	o_orderstatus	o_orderdate	512	128	o_orderstatus
	part	41 GB	p_brand	97	97	binpack	p_size	84	21	z-order(p_brand,p_type, p_container)	p_size	512	128	z-order(p_brand,p_type, p_container)
	partsupp	251 GB	-	29	29	binpack	-	512	128	binpack	-	512	128	binpack
	region	1.3 KB	-	32	8	binpack	-	32	8	r_name	-	32	8	r_name
	supplier	4.8 GB	-	43	43	binpack	s_nationkey	32	8	binpack	s_nationkey	512	128	binpack
TPC-DS (SF 10K)	catalog_returns	83.8 GB	-	97	97	binpack	cr_returned_date_sk	32	8	binpack	cr_returned_date_sk	128	32	(cr_item_sk,cr_return_amount)
	catalog_sales	763 GB	cs_sold_date_sk	128	128	binpack	cs_sold_date_sk	86	22	z-order(cs_bill_cdemo_sk, cs_bill_addr_sk, cs_sold_time_sk, cs_bill_customer_sk)	cs_sold_date_sk	512	128	z-order(cs_bill_cdemo_sk, cs_bill_addr_sk, cs_sold_time_sk, cs_bill_customer_sk)
	inventory	5.9 GB	-	128	128	binpack	inv_date_sk	32	4	binpack	inv_date_sk	512	128	inv_quantity_on_hand
	store_returns	111 GB	-	150	75	binpack	sr_returned_date_sk	32	8	(sr_reason_sk, sr_return_amt,sr_cdemo_sk)	sr_returned_date_sk	256	64	sr_reason_sk
	store_sales	893.8 GB	ss_sold_date_sk	323	108	binpack	ss_sold_date_sk	123	31	z-order(ss_store_sk, ss_addr_sk,ss_cdemo_sk, ss_promo_sk)	ss_sold_date_sk	512	128	ss_store_sk
	web_returns	39.2 GB	-	28	28	binpack	wr_returned_date_sk	32	8	wr_refunded_addr_sk	wr_returned_date_sk	512	128	wr_refunded_addr_sk
	web_sales	535.5 GB	ws_sold_date_sk	43	43	binpack	ws_sold_date_sk	37	9	z-order(ws_sold_time_sk, ws_bill_addr_sk, ws_ship_addr_sk, ws_ship_hdemo_sk)	ws_sold_date_sk	512	128	z-order(ws_sold_time_sk, ws_bill_addr_sk, ws_ship_addr_sk, ws_ship_hdemo_sk)

Human expert heuristics can be treated as gold-standard ground truth for a workload. They were discovered upon running the workload in an ad hoc, randomized way extensively with various parameter configurations. Although the human expert heuristics themselves are not parameterized by information from the query workload, the values of $files_{min}$, $files_{max}$, $partFiles_{min}$ and 100K rows per row group in Algorithm 4 of our paper are specific to SF 10K TPC-H/DS. Since $\text{PTO}_{HumanExp}$ forgoes the usage of GBT and instead replaces ML models with manually/empirically discovered heuristics to capture the co-dependencies among layout parameters, it is expected to discover an optimal layout. We can notice how PTO generically pushes TFS and RGS to be as large as possible while $\text{PTO}_{HumanExp}$ finds tailored TFS and precise RGS for each table. However, the drawback of $\text{PTO}_{HumanExp}$ is that the discovery of these heuristics takes several weeks and a plethora of experimental runs without a principled approach. Additionally, when the dataset and workload changes, re-discovering the optimal layout needs a new set of ad hoc runs. Figures 5b and 5c show that PTO benefits more queries (18 vs. 12 on TPC-H and 88 vs. 39 on TPC-DS) than $\text{PTO}_{HumanExp}$ despite being fully automatic and achieves almost comparable quality in terms of the latency benefits on the end-to-end query workload. This is because PTO shortlists candidate sort columns based on their scan selectivity and #queries they appear in as discussed in Algorithm 1.

Table 3. Latency breakdown for offline execution of PTO.

(a) Component-wise latencies.

Component	TPC-H (#Queries:22)	TPC-DS (#Queries:103)
Scan fragment extraction	1.43 sec	21.9 seconds
Sort cluster size estimation	1.89 hours	3.29 hours
Top-k candidate layout generation	0.006 seconds	33 seconds
Feature vector creation	3,763 seconds	1,581 seconds
Target label (data skipping) estimation	6.93 hours	29.6 hours
Applying the optimized layout	3.26 hours	2.74 hours

(b) Table-wise breakdown of latency-intensive components.

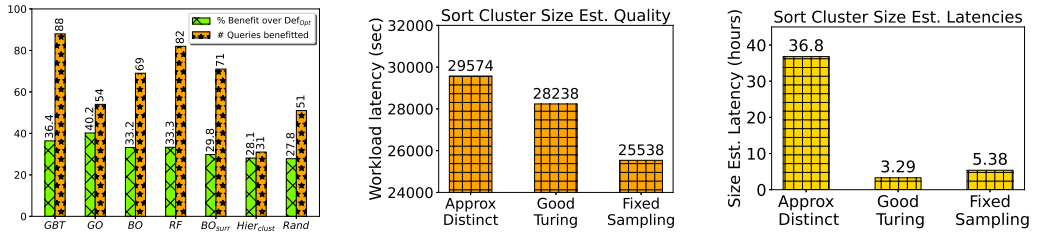
Dataset	Table	# Scan Fragment Queries	Sort cluster size estimation	Data skipping est (sample)			Apply optimized layout
				# Labeling iterations	Rewrite time/iter	Scoring time/iter	
TPC-DS	store_sales	66	1,416 sec	12	341 sec	4,753 sec	3,600 sec
	store_returns	15	1,233 sec		176 sec	117 sec	420 sec
	catalog_sales	37	884 sec		344 sec	1,360 sec	3,300 sec
	catalog_returns	10	273 sec		143 sec	41 sec	246 sec
	web_sales	35	7,007 sec		353 sec	539 sec	2,160 sec
	web_returns	10	963 sec		201 sec	38 sec	120 sec
	inventory	6	89 sec		116 sec	17 sec	28 sec
TPC-H	lineitem	5	4,088 sec	12	197 sec	214 sec	7,401 sec
	customer	4	1,235 sec		103 sec	16 sec	1,860 sec
	orders	3	221 sec		325 sec	132 sec	1,380 sec
	part	7	1,230 sec		110 sec	49 sec	480 sec
	partsupp	0	-		-	-	480 sec
	supplier	5	15 sec		92 sec	10 sec	113 sec
	nation	4	4 sec		-	-	17 sec
	region	3	3 sec		-	-	13 sec

4.2.2 Latency of PTO to generate the optimized layout. Table 3 shows the latencies incurred by PTO to discover the optimized layout. As we can notice from the component-wise latencies in Table 3a, the most expensive step is the generation of target labels for training feature vectors using the estimation of data skipping benefits detailed in Algorithm 3. Despite using sampling, it is latency inducing because each of the 12 training layouts needs to be evaluated over the scan fragment queries. Table 3b shows the time taken per labeling iteration and we see that evaluating one layout

takes 4,753 seconds for the TPC-DS store_sales table alone. One way to optimize this latency is to reduce the number of scan fragment queries used to estimate skipping benefits. We empirically show how we can achieve this without sacrificing layout quality in Section 4.5. Sort cluster size estimation is another such latency-intensive step for which we use an optimized approach of iterative sampling with the Good-Turing estimator. Since layout discovery is an offline step, it can be run in the background asynchronously.

4.3 Evaluating the components in PTO

In this section, we evaluate the components of PTO such as the predictive model, sort cluster size estimation, candidate sort scheme discovery. We also evaluate the impact of prioritizing long-running queries using query weighting on the layout quality.



(a) Quality of Predictive Models (b) Sort cluster size est. (Quality) (c) Sort cluster size est. (Latency)

Fig. 6. Evaluating the quality of predictive models and sort cluster size estimation (TPC-DS SF 10K).

4.3.1 Evaluation of the Predictive Model. Here, we evaluate ML-based parameter co-detection models in PTO. Figure 6a evaluates the quality of the predictive models used in PTO and Table 4 shows the time taken to label the training samples. Since GBT and random forest (RF) are supervised models, they are trained in a single iteration using the 12 labeled examples (~3-4% of the candidate search space). Note that the actual training time of the model is insignificant but as listed in Table 3, label generation is the most expensive step. In fact, hyperparameter optimization (HPO) models such as genetic optimization (GO), Bayesian optimization (BO), hierarchical clustering (Hier_{clust}), and random selection using Bayesian models need 20 more training labels (an extra ~5-6% of the candidate space). All the HPO baselines including random search score one data layout per iteration thus labeling 20 extra training samples in 20 iterations. This is because they use 12 pre-trained layouts as bootstrapped data and need more iterations to detect the actual layout. That leads to a whopping overall labeling latency of 75.89 hours for the 32 labeling iterations in total for such alternative models which is why we use GBT as the default predictive model.

Figures 6a shows that the only HPO model benefiting from this additional training is GO which shows a 40.2% savings in latency over Def_{opt} at the expense of an additional 46 hours of labeling time. Interestingly, GO benefits 54 queries while 88 queries are benefited by GBT. Hybrid baseline, BO_{surr}, using surrogate RF model runs for 380 iterations but scores only 20 training samples. In the remaining 360 iterations, it uses surrogate scores from RF. BO_{surr} performs worse than BO showing the futility of surrogate labels.

4.3.2 Evaluating Sort Cluster Size Estimation. Here, we show how replacing the default sort cluster size estimator, i.e., Good-Turing using iterative sampling (Algorithm 2) with alternative estimators based on APPROX DISTINCT and fixed sampling affects the quality and estimation latency.

Table 4. Labeling latencies for model training (TPC-DS).

Model Name	GBT	RF	GO	BO	Hier_{clust}	Rand	BO_{surr}
# Labeling Iterations	12		32				
# Training Iterations	1		20				380
Overall Latency	**29.6 hours**		**75.89 hours**				

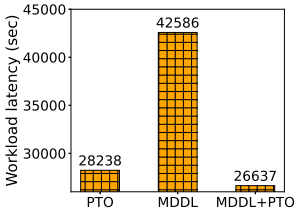


Table Name	PTO	MDDL		
	Candidate sort scheme detection	Shortlist selective predicates	Selective predicate evaluation	Sort key (bit vector) creation
store_sales	0.41 hr	474 sec	1.5 hr	1.56 hr
store_returns	0.36 hr	33 sec	0.34 hr	1.4 hr
catalog_sales	934 sec	111 sec	1.01 hr	1.1 hr
catalog_returns	328 sec	21 sec	2.73 hr	2.97 hr
web_sales	1.96 hr	94 sec	1.15 hr	1.42 hr
web_returns	0.28 hr	19 sec	1.7 hr	1.61 hr
inventory	144 sec	11 sec	0.75 hr	0.79 hr

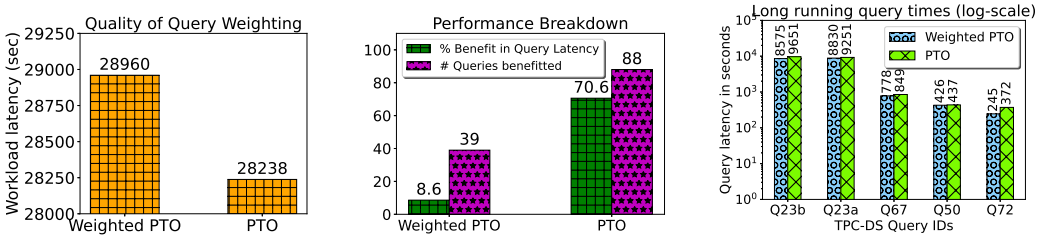
Fig. 7. Evaluating data layouts from PTO vs. MDDL [19]

(1) **APPROX DISTINCT:** We run APPROX DISTINCT [21] on the full table to compute # sort clusters and the average distinct sort cluster size is estimated as $\frac{tableCardinality}{\#SortClusters}$.
(2) **FIXED SAMPLING :** We take the upper bound of the sampling rate (0.1%) used in our Good-Turing implementation and repurpose it as fixed sampling rate on each table to measure the number of sort clusters and average distinct sort cluster size.

As mentioned in Section 3.2.4, Good-Turing can be replaced with any decent alternative estimator in PTO. APPROX DISTINCT is directly applied on the full set of tables without sampling and is expected to be a more accurate estimator than Good-Turing and fixed sampling-based estimators. However, Figure 6b shows that APPROX DISTINCT fails to fetch better query latency savings than Good-Turing despite spending 6.8× more time estimating the cluster sizes. On the contrary, fixed sampling estimator at 0.1% sample rate yields the best latency savings. Upon inspecting the detected layouts, we found that the #queries benefited from each discovered sort strategy seem to impact the layout quality. This emphasizes the importance of co-optimizing both sort cluster size and the expected #queries benefited from each layout. Note that fixed sampling takes 1.6× longer than Good-Turing as shown in Figure 6c to estimate the sort cluster sizes because iterative sampling using Good-Turing converges much sooner than 0.1% sample rate used by fixed sampling.

4.3.3 Comparison with MDDL [19]. In this section, we compare PTO against multidimensional data layout (MDDL) [19] form Amazon Redshift to evaluate candidate sort scheme discovery (discussed in Section 3.2.3). Ding et al. [19] characterize a data layout only based on the sort scheme and do not discover other parameters unlike PTO. Instead of using columns to sort the table, the top-k most selective filter predicates from the query workload are represented as a multidimensional bit vector which serves as the sort key. If a tuple satisfies a predicate, the corresponding bit is set. As we can see in Figure 7, the table layouts detected by MDDL result in 1.5× longer workload execution latencies (42.5K seconds vs. 28.2K seconds) compared to PTO. Additionally, MDDL takes up to 62.6× longer than PTO to detect the sort scheme. To give MDDL added advantage, we implement a variant MDDL+PTO which uses the optimal partitioning column, TFS and RGS discovered by PTO in combination with the sort strategy discovered by MDDL to get 6% better latency savings (26.6K seconds vs. 28.2K seconds) than PTO. In the original implementation of MDDL [19], atomic predicates are used to represent each bit in the sort key which we found to be more time-consuming to evaluate over large tables and can lead to scalability issues. Hence, we give MDDL an additional

advantage of directly using the filter predicates that we extract from the query workload as sort dimensions unlike Z-ordering which uses the filter columns as sort dimensions.



(a) Quality (b) Performance breakdown (c) Impact on long running queries

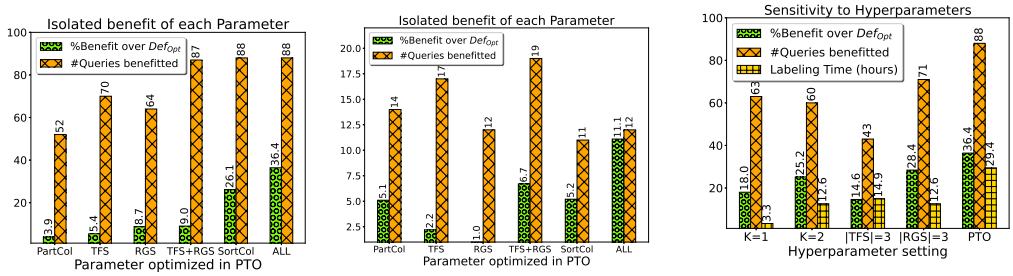
Fig. 8. Evaluation of query weighting (TPC-DS SF 10K).

4.3.4 Impact of Query Weighting. Here, we evaluate the impact of weighting queries from the workload during candidate sort scheme selection on the overall quality of the discovered data layouts. In real query workloads, end users may implicitly prioritize some queries over others by running them more frequently and it make sense to bias the layouts to benefit such queries. In the case of TPC-DS, there is no notion of how frequently each query is run. Therefore, we use the normalized latency of each query i.e., $\frac{queryLatency - minLatency}{maxLatency}$ (where min and max latencies are computed across all queries) $\in [0, 1]$ as its weight. Figures 8a, 8b show that Weighted PTO performs 2.5% worse than PTO w.r.t. overall savings in workload latency and benefits only 39 queries by 8.6% over Def_{opt}. This is because query weighting is inherently biased towards high latency queries. Interestingly, the top-5 long running queries benefited by Weighted PTO (Figure 8c) consist of TPC-DS Q23a and Q23b which run for over 2.5 hours. Upon these two queries, Weighted PTO yields 4% and 11% latency savings compared to PTO.

4.4 Ablation Study and Sensitivity Evaluation

In this section, we report the results of an ablation study evaluating the isolated benefit of optimizing each data layout parameter one at a time. The second evaluation studies the sensitivity of PTO to hyperparameters such as top-k and candidate TFS, RGS values.

4.4.1 Isolated Benefit of each Layout Parameter. Figure 9a shows % improvement over Def_{opt} as well as the #queries benefitted by optimizing each parameter in isolation for TPC-DS. As we can notice, optimizing the partition column (*PartCol*) alone shows the least improvement of 3.9% and over fewest queries (52 out of 103 TPC-DS queries). This is because the largest of the fact tables (store_sales, web_sales, catalog_sales) are already partitioned by Def_{opt}. PTO additionally partitions store_returns, web_returns, catalog_returns and inventory which benefits fewer queries such as Q83 involving filtered scans on one or more of these tables. As expected, optimizing the sort schemes (*SortCol*) results in the most dominant latency reduction among all parameters by 26% while speeding up 88 queries. However, sorting alone falls short of the maximum possible benefit (36%) PTO can achieve by optimizing ALL parameters as shown in the figure. This establishes the importance of co-optimizing data layout parameters in PTO. Likewise, optimizing TFS alone gains only 5% because larger file sizes (512 MB) with smaller row groups (28 MB, 43 MB) created several (12 to 18) row groups for tables like web_sales and web_returns which was counterproductive. Since row groups were parameterized based on file sizes (by having 1 to 10 row groups per file), we



(a) Ablation Study (TPC-DS) (b) Ablation Study (TPC-H) (c) Parameter Sensitivity (TPC-DS)

Fig. 9. Isolated Benefit of Each Parameter (Ablation Study) and Sensitivity to Hyperparameters

evaluated optimizing both TFS and RGS. Interestingly, TFS+RGS benefitted more queries but showed comparable improvement (~9%) to optimizing RGS alone. This explains that tuning TFS:RGS ratio is more important than tuning TFS alone as the overhead spent in reading metadata for several row groups negates the compaction gains from large TFS. Tuning RGS sets optimal #row groups for store_sales and catalog_returns thus benefitting 64 queries.

Figure 9b also shows that optimizing ALL parameters is crucial to get the highest savings of 11% over Def_{Opt} upon the TPC-H benchmark. Interestingly, optimizing TFS or RGS alone did not yield enough savings but co-optimizing TFS and RGS achieves the highest standalone benefit of 6.7%. This explains why increasing TFS alone with smaller row groups is not beneficial consistent with our observation on TPC-DS. Increasing RGS alone benefitted TPC-DS but did not make a difference for most TPC-H tables as it did not change #row groups per file. For instance, increasing row group size from 29 MB to 128 MB does not make a difference on a file sized 29 MB because the file has a single row group in both the settings. TFS+RGS also surpassed individually optimizing sort columns or partitioning column thus establishing the need to co-discover parameters. This is because Def_{opt} uses lower TFS, RGS values and compacting them with suitable larger sizes fetches more benefit than isolated sorting or partitioning which is logically equivalent to a single-column sort. Isolated optimization sometimes benefitted more queries (14 to 19) compared to co-optimization benefiting 12 out of 22 TPC-H queries but the overall latency reduction is better with co-optimization. We treat the findings on TPC-DS more representative than TPC-H due to its comprehensive workload.

4.4.2 Sensitivity to Hyperparameters. PTO is most sensitive to the size of the shortlisted search space which is determined by top-k partitioning columns, top-k sort candidates and the discrete set of target file sizes (TFS). Therefore, in this section, we will study the impact of the size of the search space on PTO. More specifically, the goal of this experiment is to evaluate the quality of the data layouts PTO can detect from hyperparameter settings that create a smaller search space. Enlarging the shortlisted search space will further increase the optimization time of PTO. Instead of enlarging search space (more particularly for TFS), we can alter it for larger scale factors (scalability experiment in Section 4.5). Varying ML model hyperparameters did not have a significant impact due to an already compact search space. Instead of top-k=3 which is the default setting in PTO, we used top-k={1, 2} for candidate partitioning columns and sort strategies. We also included a comparison while reducing #TFS choices from 5 to 3 candidates (32 MB, 128 MB, 512 MB) and #RGS to 3 candidates resulting in 1, 4 and 10 row groups per file to evaluate the impact of shortlisting a smaller candidate space (fewer than 3 candidates is too tiny to get meaningful insights).

Table 5. Search space sizes for Hyperparameter Settings.

Setting	Top-k=1	Top-k=2	|TFS|=3	|RGS|=3	PTO
# Candidates Per Table	60	180	216	180	360

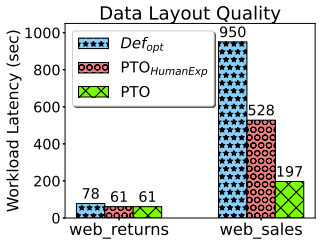


Fig. 10. Scalability Evaluation (TPC-DS SF 30K)

Table 5 shows the candidate space size per table for each evaluated hyperparameter setting. including the default search space sized 360 for PTO. Setting top-k=1 produces a candidate space sized (top-1 sort strategy × 5 TFS × 6 RGS per TFS × (top-1 partitioning + No Partitioning as an additional candidate)) thus yielding 60 candidates in the search space. Figure 9c and Table 5 show that top-k=2 yields a 25% reduction in workload execution latency over Def_{opt} with a shortlisted search space of 180 candidates which is better than 18% gain at top-k=1. Setting |RGS| to 3 fetches a higher benefit of 28% despite equally shortlisting 180 candidates. This is because |RGS|=3 still preserves the three most important RGS choices without pruning any of the top-k sort schemes or partitioning columns. Thus, PTO is highly sensitive to top-k and also to TFS because |TFS| = 3 reduced the latency gain to 14% despite shortlisting 216 candidates. Upon closer examination, we found that the lack of diverse TFS candidates in the training data did not give enough signal to PTO about optimal file sizes and row groups to select because RGS values are also pivoted on TFS. Since we choose ~3% of the candidate space for training, lower hyperparameters led to better labeling (skipping benefit estimation) times than vanilla PTO.

4.5 Scalability Evaluation w.r.t. Table Sizes

In this section, we show that PTO can scale to larger tables. To enable this evaluation on our cluster of 4 machines, we chose TPC-DS web_sales and web_returns at SF 30K as they are a representative pair of <sales, returns> fact tables. Since we cannot run the end-to-end TPC-DS queries on a controlled schema with two fact tables, we used the scan fragment queries on corresponding tables as the query workload to compare PTO with Def_{Opt}. To accommodate growing table sizes and keep search space compact, we altered candidate TFS space from [32, 64, 128, 256, 512] MB to [64, 128, 256, 512, 1024] MB. We applied SF 10K $PTO_{HumanExp}$ heuristics at 30K and set $files_{min}$ and $files_{max}$ to 64 MB and 1024 MB in Algorithm 4. Table 6 shows that PTO found larger TFS, RGS for web_sales but smaller TFS, RGS for web_returns at SF 30K than 10K due to the larger 0.1% sample drawn at SF 30K. This is because PTO estimates skipping benefits of layouts on sample tables which cannot be computed effectively unless the downscaled TFS and RGS can pack sufficient #rows in each rewritten row group or Parquet file in the sample. This phenomenon led to several downscaled candidate TFS and RGS configurations not resulting in enough row groups or target files to produce meaningful skipping benefits. Since samples are tiny at SF 10K for the relatively smaller web_returns table, we found that TFS and RGS were artificially pushed to larger values to

Table 6. Optimized layouts configured by Def_{opt}, $PTO_{HumanExp}$ and PTO for web_returns and web_sales in TPC-DS SF 30K.

Table	Total Size	Def_{opt}				$PTO_{HumanExp}$				PTO			
		Partitioning Column	TFS (MB)	RGS (MB)	Sort Scheme	Partitioning Column	TFS (MB)	RGS (MB)	Sort Scheme	Partitioning Column	TFS (MB)	RGS (MB)	Sort Scheme
web_returns	**130.6 GB**	-	28	28	binpack	wr_returned_date_sk	64	16	wr_refunded_addr_sk	wr_returned_date_sk	64	16	wr_refunded_addr_sk
web_sales	**1.15 TB**	ws_sold_date_sk	43	43	binpack	ws_sold_date_sk	103	26	z-order(ws_ship_date_sk, ws_ship_hdemo_sk)	ws_sold_date_sk	1024	256	z-order(ws_ship_date_sk, ws_ship_hdemo_sk)

Table 7. Latency breakdown for TPC-DS SF 30K.

Table	# Scan Fragment Queries	Sort cluster size estimation	Target label estimation (sample)		Apply optimized layout
			Vanilla PTO	Lightweight PTO (#Training queries)	
web_returns	10	4,857 sec	5,593 sec	2,858 sec (4 queries)	300 sec
web_sales	35	7,119 sec	18,725 sec	9,569 sec (16 queries)	3,360 sec

Table 8. Sample Scan Fragment Query Latencies (TPC-DS SF 30K)

Table	Scan Fragment Query	Def_{Opt}	$PTO_{HumanExp}$	PTO
web_returns	SELECT count(*),sum(wr_return_quantity) FROM WEB_RETURNS WHERE wr_returned_date_sk BETWEEN 2452133 AND 2452163;	6.7 sec	2.5 sec	2.5 sec
web_sales	SELECT count(*),sum(ws_quantity) FROM WEB_SALES WHERE ((ws_web_page_sk BETWEEN 6 AND 2966) AND (ws_ship_hdemo_sk BETWEEN 240 AND 6359) AND (ws_sold_time_sk BETWEEN 32400 AND 39599));	109.2 sec	60 sec	14.5 sec

counter the downscaling effect. This specific anomaly was rectified at SF 30K where the absolute sizes of the samples were larger and TFS and RGS were pushed to higher values only if necessary and not because samples were too small to yield substantial #row groups or files for a particular layout configuration. Another interesting observation is that the sort scheme changed at SF 30K from what was earlier discovered at SF 10K. This can be explained by the fact that the cardinality of unique keys and concentration of data skews do not grow linearly between scale factors although the row counts scale linearly for fact tables. Additionally, the absolute sample sizes changed at SF 30K along with the estimated skipping benefits for the training layouts which led to GBT predicting different skipping benefits (regression scores) for the candidate layouts compared to SF 10K.

Besides altering the TFS candidate space, we implemented a Lightweight PTO that used only a subset of scan fragment queries to estimate skipping benefits (training labels). We chose scan fragment queries having distinct combinations of columns in their filter predicates and this led to training on 4 out of 10 scan fragment queries for web_returns and 16 out of 35 scan fragment queries for web_sales. This optimization reduced the training label estimation latency by ~50% as shown in Table 7 without sacrificing layout quality. Figure 10 shows that PTO reduced workload latency by 21% on web_returns and 79% on web_sales over Def_{Opt}. Interestingly, there were no regressions on web_sales despite training only on a fraction of the queries and all the scan fragment queries were benefitted by 40%-93%. On web_returns, 5 out of 10 queries were benefitted by 26%-62% and the regressed queries did not benefit from the sort scheme due to a non-overlapping filter column. $\text{PTO}_{HumanExp}$ found the same TFS, RGS as PTO on web_returns and benefitted same #queries but detected lower TFS, RGS for web_sales thereby incurring longer workload latencies and establishing the need to re-discover human expert heuristics when scale factors (and datasets) vary. Table 8 shows latencies for sample queries.

5 Discussion

In this section, we clearly state the scope of PTO and the scenarios where it will be applicable in its current implementation while discussing its possible extensions.

1. Generalizability: PTO is applicable to lakehouse systems and open table formats with similar data layouts (parameterized by file sizes, partitioning, sort scheme) and can be adapted to other table formats as long as they have tunable parameters. Micropartitions in Snowflake [14] eliminate the need to detect a partition column but require periodical re-clustering with manually specified columns and MAX_SIZE to maintain a desired system clustering ratio. Automatic clustering [22] in Snowflake decides when to run re-clustering but does not identify the sort columns. PTO is still relevant in such cases to minimize manual intervention. Likewise, Databricks liquid clustering [20, 30] does not partition tables but configures clustering columns, MIN_CUBE_SIZE and TARGET_CUBE_SIZE. Although a separate implementation for each lakehouse system is beyond the scope of this paper, ideas from PTO like reducing search space of candidate layouts and training ML models on skipping benefits learned from sampled tables can be applied to other systems.

2. Scalability w.r.t. #Tables: Since there are multiple tables in a lakehouse, it may not be scalable for PTO to run table optimization for each table. We currently prioritize tables to be optimized based on their corresponding sample sizes. If we end up with empty samples for a table, it is left unoptimized. A possible extension is that PTO can prioritize the tables in a lakehouse w.r.t. their cardinalities, schema width in terms of #frequently queried columns a table contains, #queries in which a table appears and their frequency. We can let PTO run either until a pre-specified latency budget is exhausted or all the tables are optimized. It is likely that the budget is exhausted well before all the tables are optimized and the remaining tables which are left unoptimized are hopefully less important to the workload. Additionally, PTO can be applied in parallel to multiple tables given a larger computational cluster with enough storage as tables can be optimized independent of each

other. All the steps in PTO such as table sampling, estimating the skipping benefits for shortlisted candidate layouts on sampled tables, ML model training and inference can be run in parallel across tables. We apply PTO sequentially to each table due to resource constraints.

3. Optimization granularity: PTO currently detects optimal layouts at a table level and can be extended to support partition-level optimization. This is cheaper than rewriting the entire table when data ingests affect a subset of the partitions but this also assumes that the partitioning scheme cannot change after data ingests and is hence not re-discovered. PTO can achieve partition-level rewrites by leveraging the support for WHERE clauses during rewrites offered by the Spark-Iceberg library [5]. We can specify the partitions to be optimized using the rewrite_data_files procedure in a selection predicate on the partitioning column.

4. Data Ingests and Workload Evolution: We will discuss the following scenarios separately - catering to data ingests and query workload drift in isolation vs. handling workload drift while data ingests are concurrently happening. Data ingestion and workload drift are tackled separately in PTO. Data ingests can change the underlying data distribution and result in stale table samples which necessitate re-sampling and re-learning the skipping benefits. To support data ingests, the current implementation of PTO is re-run periodically in a batched manner on the entire table. This can be further optimized by applying PTO only on the affected partitions as mentioned earlier thereby isolating the rewrites from concurrent data ingests. A tradeoff here is that simultaneous data ingests can create new candidates for layout parameters such as a larger TFS and corresponding RGS. We do not expect new candidate sort schemes unless new queries arrive with filter predicates on columns outside the candidate list or schema-level changes are done introducing new columns. Since we apply PTO in a batched manner, we can support offline partition-level optimization but supporting concurrent data ingests in real time requires careful re-design.

On the other hand, supporting workload evolution in isolation is more straightforward. Although automatic workload drift detection is not supported, we can invoke PTO explicitly when new queries arrive. PTO can be retrained on an updated workload containing all the latest queries and a tiny subset of older queries having the highest weights (Section 4.3.4 discusses query weighting).

Supporting concurrent data ingests while query workload evolves is a challenging problem because data ingestion necessitates re-learning of skipping benefits even for historical (repeating) candidate layouts. That is possible when the workload is considered static and allows estimating the target labels on a *frozen* set of scan fragments. If the scan fragments are evolving concurrently as ingests happen, we pivot the learning to the most recent workload.

6 Related Work

As data gets ingested into the Parquet (or ORC or any lakehouse-compatible columnar formats) files within the Iceberg format, multiple small files are created. This coupled with updates via Copy-on-Write (CoW) and Merge-on-Read (MoR) techniques [28] entails that periodic compaction is applied on small files to get larger Parquet files meeting the requirement of a pre-specified target file size (TFS). The compacted files are sized such that the individual file size is close to TFS. AutoComp [25] analyzes the cost-benefit tradeoff in compacting candidate small files based on multiple heuristics such as the rewrite costs and the computational resources spent in reaching a reduced file count vs. the table utility in a given workload. AutoComp further enables parallel rewrites by scoping the rewrite granularity at partition-level or snapshot-level. AutoComp also suggests that TFS could be tuned based on the query workload. While the OPTIMIZE feature in Dremio [12, 13], Delta Lake [17], and S3 Tables [7] achieve compaction, they require TFS to be manually specified or a default value of TFS is applied in all the cases. Similarly, a default row group size (RGS) is used to create the row groups within each Parquet file. Predictive optimization

in Databricks Delta Lake [18] determines when to periodically re-run the OPTIMIZE command but does not discover the table optimization parameters.

At the time of querying, skipping of irrelevant data is maximized when the query predicates match the partitioning scheme and the sort strategy applied on the tables. Zimmerer et al. [43] extend the scope of pruning techniques for Snowflake micro-partitions beyond filtered scan predicates by deriving additional min/max ranges for complex expressions involving LIMIT, top-k and JOIN operators. Such advanced pruning techniques can be augmented with PTO to further enhance the skipping capability. Amazon Redshift [19] exploits the multidimensional predicates (MDDL) appearing in the query workload to determine the sort order but does not automatically detect the partitioning columns, TFS or RGS. Another limitation of MDDL is that the predicates used to sort the columns require an exact match with the query predicates to yield any skipping benefits. Also, the predicates containing non-deterministic functions such as RANDOM() are not used for sort ordering. PTO, on the other hand, does not make any such assumptions about the query predicates as the sorting is done on columns. Similarly to MDDL, we re-run PTO periodically to support workload evolution and data ingests. On the flipside, PTO assumes access to both data and query workload whereas MDDL does not need access to underlying data thereby preserving data privacy.

HP Vertica DBDesigner [39] discovers sort strategies by including candidate sort columns from join predicates, aggregates and GROUP BY clauses besides scan predicates which enriches the candidate search space. However, it does not apply Z-ordering and sort orders are dicovered for materialized views. Databricks liquid clustering [20, 30] discovers optimal layouts and differs from PTO w.r.t. the space filling curve (Hilbert vs. Z-ordering). It is amenable to data updates because it avoids the need to specify a partitioning column but configures clustering columns, MIN_CUBE_SIZE and TARGET_CUBE_SIZE. Liquid clustering is not open-sourced for Iceberg tables that precludes a comparison with PTO. Micropartitions in Snowflake [14] require clustering columns and MAX_SIZE to be configured which can benefit from PTO.

7 Conclusion

In this work, we built a workload-driven predictive table optimizer, PTO for Apache Iceberg tables on the top of Presto lakehouse engine. PTO leverages the scan fragments in the workload to co-discover the optimal partitioning column, target file size, row group size and the multidimensional sort strategy. We propose heuristics to significantly reduce the candidate search space based on sort cluster size, selectivity and filter column frequency of the columns appearing in the filter predicates of the scan fragments. Subsequently, we learn a predictive model (GBT) utilizing as few as ~3-4% of the candidate space to reduce the layout discovery latency significantly and estimate the data skipping benefits of the candidate layouts. The optimal layout predicted by the ML model is applied on the full set of tables. Our experimental results on TPC-H and TPC-DS at SF 10K show that PTO yields up to 11%-36% average savings in workload execution latencies while speeding up scan-intensive, long latency queries by 3.4× - 11×. Our evaluation against multiple baselines validates the design choices for each component in PTO w.r.t. balancing the optimality of layouts and #queries benefited with the layout discovery latency. Currently, we support evolving query workload and data by periodically re-running PTO and a more principled solution for these cases is left for future work.

References

[1] 2017. Apache Hudi. https://hudi.apache.org/docs/overview.
[2] 2017. Apache Hudi Github Repository. https://github.com/apache/hudi.
[3] 2017. Apache Iceberg Documentation. https://iceberg.apache.org/docs/nightly/.
[4] 2017. Apache Iceberg Github Repository. https://github.com/apache/iceberg.

[5] 2017. Apache Iceberg: Spark procedures. https://iceberg.apache.org/docs/nightly/spark-procedures/.

[6] 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. doi:10.14778/3415478.3415560

[7] Jeff Barr. 2024. New Amazon S3 Tables: Storage optimized for analytics workloads. https://aws.amazon.com/blogs/aws/new-amazon-s3-tables-storage-optimized-for-analytics-workloads/.

[8] James Bergstra, Daniel Yamins, and David D. Cox. 2013. Hyperopt: Distributed Hyperparameter Optimization. https://github.com/hyperopt/hyperopt/.

[9] James Bergstra, Daniel Yamins, and David D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 115–123. http://proceedings.mlr.press/v28/bergstra13.html

[10] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8391)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer, 61–76. doi:10.1007/978-3-319-04936-6_5

[11] Ramesh Chandra, Haogang Chen, Ray Matharu, Sarah Cai, Jeff Chen, Priyam Dutta, Bogdan Ghita, Todd Greenstein, Gopal Holla, Peng Huang, Yuchen Huo, Adrian Ionescu, Adriana Ispas, Tim Januschowski, Vihang Karajgaonkar, Stefania Leone, David Lewis, Andrew Li, Nong Li, Cheng Lian, Stephen Link, Qing Lu, Yesheng Ma, Chris Pettitt, Vijayan Prabhakaran, Bogdan Raducanu, Kyle Rong, Paul Roome, Samarth Shetty, Sean Smith, Xiaotong Sun, Yuyuan Tang, Weitao Wen, Lei Xia, Junlin Zeng, Ben Zhang, Reynold Xin, and Matei Zaharia. 2025. Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 310–322. doi:10.1145/3722212.3724459

[12] Dremio Cloud. 2024. Optimize Table. https://docs.dremio.com/cloud/reference/sql/commands/optimize-table/.

[13] Dremio Cloud. 2024. Optimizing Tables. https://docs.dremio.com/cloud/sonar/query-manage/managing-tables/optimizing/.

[14] Snowflake Community. 2022. Understanding Micro-partitions and Data Clustering. https://community.snowflake.com/s/article/understanding-micro-partitions-and-data-clustering.

[15] Transaction Processing Council. 1999. TPC-H Version 2 and Version 3. https://www.tpc.org/tpch/.

[16] Transaction Processing Council. 2006. TPC-DS Version 2 and Version 3. https://www.tpc.org/tpcds/.

[17] Databricks. 2024. Optimize data file layout. https://docs.databricks.com/en/delta/optimize.html.

[18] Databricks. 2024. Predictive optimization for Unity Catalog managed tables. https://docs.databricks.com/en/optimizations/predictive-optimization.html.

[19] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. 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, 55–67. doi:10.1145/3626246.3653379

[20] Databricks Documentation. 2024. Use liquid clustering for Delta tables . https://docs.databricks.com/en/delta/clustering.html.

[21] Presto 0.290 Documentation. 2024. Approximate Aggregate Functions. https://prestodb.io/docs/current/functions/aggregate.html.

[22] Snowflake Documentation. 2022. Automatic Clustering. https://docs.snowflake.com/en/user-guide/tables-auto-reclustering.

[23] Dremio. 2020. Project Nessie: Transactional Catalog for Data Lakes with Git-like semantics. https://projectnessie.org.

[24] Dremio. 2025. The Architect's Guide to Open Table Formats and Object Storage. https://www.dremio.com/newsroom/the-architects-guide-to-open-table-formats-and-object-storage/.

[25] Anja Gruenheid, Jesús Camacho-Rodríguez, Carlo Curino, Raghu Ramakrishnan, Stanislav Pak, Sumedh Sakdeo, Lenisha Gandhi, Sandeep K. Singhal, Pooja Nilangekar, and Daniel J. Abadi. 2025. AutoComp: Automated Data Compaction for Log-Structured Tables in Data Lakes. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 404–417. doi:10.1145/3722212.3724430

[26] Apache Hudi. 2017. Indexing. https://hudi.apache.org/docs/metadata_indexing/.

[27] Jason Hughes. 2021. Apache Iceberg: An Architectural Look Under the Covers. https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/.

[28] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf

[29] David A. McAllester and Robert E. Schapire. 2000. On the Convergence Rate of Good-Turing Estimators. In *Proceedings of the Thirteenth Annual Conference on Computational Learning Theory (COLT 2000), June 28 - July 1, 2000, Palo Alto, California, USA*, Nicolò Cesa-Bianchi and Sally A. Goldman (Eds.). Morgan Kaufmann, 1–6.

[30] jaseidman mssaperla, irinaskaya. 2024. Use liquid clustering for delta tables. https://learn.microsoft.com/en-us/azure/databricks/delta/clustering.

[31] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1049–1058. http://dl.acm.org/citation.cfm?id=1164217

[32] Alon Orlitsky and Ananda Theertha Suresh. 2015. Competitive Distribution Estimation: Why is Good-Turing Good. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 2143–2151. https://proceedings.neurips.cc/paper/2015/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html

[33] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384. doi:10.14778/3554821.3554829

[34] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1802–1813. doi:10.1109/ICDE.2019.00196

[35] Evan Smith. 2024. What are Open Table Formats? Why data lakehouses rely on open table formats like Iceberg. https://www.starburst.io/blog/open-table-formats/.

[36] Simon Spati. 2023. The Open Lakehouse Stack: DuckDB and the Rise of Table Formats. https://motherduck.com/blog/open-lakehouse-stack-duckdb-table-formats/.

[37] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2 (2023), 189:1–189:25. doi:10.1145/3589769

[38] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proc. VLDB Endow.* 2, 2 (2009), 1626–1629. doi:10.14778/1687553.1687609

[39] Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary, Jaimin Dave, and Sreenath Bodagala. 2014. DBDesigner: A customizable physical design tool for Vertica Analytic Database. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 1084–1095. doi:10.1109/ICDE.2014.6816725

[40] Kyle Weller. 2024. Comprehensive Data Catalog Comparison. https://www.onehouse.ai/blog/comprehensive-data-catalog-comparison.

[41] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[42] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161. https://www.vldb.org/pvldb/vol17/p148-zeng.pdf

[43] 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/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 757–770. doi:10.1145/3722212.3724447

Received July 2025; revised October 2025; accepted November 2025