Appendix to 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

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

Appendix A Background

Here, we discuss some background material about lakehouse systems, open table formats which can serve as an extension of the preliminaries.

- **Lakehouse engines:** Lakehouses are loosely defined as the warehouses for data lakes [24]. As mentioned in Section 1 of the paper, the separation of storage from data is what scales lakehouses to petabytes and exabytes of data by allowing for federated query processing over distributed data sources. Some of the popular state-of-the-art lakehouse engines include Databricks Photon [3], Dremio [1], Presto [20]. While Presto is implemented in Java, Prestissimo [22] improves upon it by combining the native Java-based coordinator from Presto with the C++ implementation of workers (termed *Velox* [18] workers) to show impressive latency reduction ranging from 3×-6× on the publicly available benchmarks such as TPC-H compared to vanilla implementation of Presto. Besides, Presto and Prestissimo are open-sourced which incentivized us to design PTO on the top of Presto engine.
- **Open Table Formats:** Some of the popular open table formats are Delta Lake, Hudi and Iceberg [13] among which we choose Iceberg as the table format for this work. A common feature among all these table formats is the availability of richer metadata compared to their predecessors such as Apache Hive. Besides schema evolution, snapshot isolation and time travel which are supported by *modern* table formats such as Iceberg, the metadata is available hierarchically at partition level and file level unlike Hive and Parquet which offer metadata at the file and row group level. Additionally, manifest files, manifest lists and metadata files form a tree-like structure enabling the storage of statistics such as <min, max> values for table columns at snapshot level thus enabling time travel (AS OF) queries. Equivalently, Delta lake and Hudi support transaction logs coupled with metadata checkpoints/tables for the same purpose. Lakehouse engines such as Presto leverage the combination of metadata supplied by Iceberg and the row group level metadata offered by individual Parquet files to achieve data skipping during query execution. We

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.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/2-ART67
https://doi.org/10.1145/3786681

chose Apache Iceberg as the table format for our work because of the skipping capabilities it offers and the ease of usage through the Spark-Iceberg [2] library.

A.1 Illustrating the skipping benefits

In this section, we use examples to illustrate the skipping benefits we get by configuring each of the table optimization parameters.

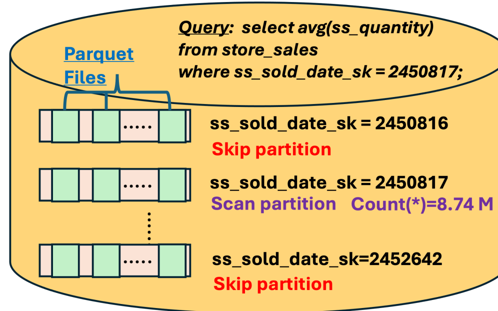


Fig. 1. Partition skipping on the TPC-DS store_sales table.

- **Partitioning Column:** By configuring the partitioning column optimally, we can skip scanning rows from irrelevant partitions in the Iceberg tables. Figure 1 shows an example aggregate query with a filter predicate on the column ss_sold_date_sk, and how partitioning a TPC-DS scale factor (SF) 10K store_sales table on ss_sold_date_sk allows skipping 28.79B rows from 1,823 irrelevant partitions and scanning only 8.7M rows.

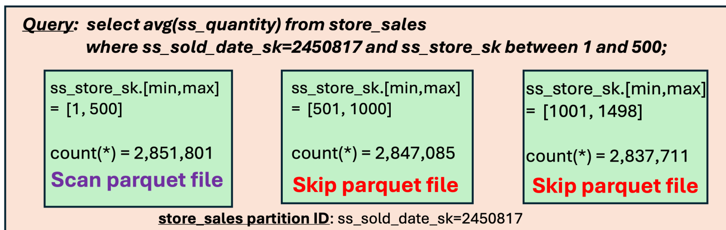


Fig. 2. File skipping in a TPC-DS store_sales partition.

- **Target File Size (TFS):** This parameter determines the upper limit of the size of each Parquet file in the Iceberg table. Assuming that the rows within the relevant partition of the store_sales table are sorted by ss_store_sk (thus benefiting the filter predicate in the query), we can see from Figure 2 that setting the TFS to ~2.8M rows results in skipping two of three Parquet files from the partition. The [min,max] statistics on the sorted column matching the filter predicate and TFS determine #files scanned.
- **Row Group Size (RGS):** This parameter specifies the row group size in a Parquet file and determines # row groups to be scanned during query processing. Figure 3 shows a query with a filter predicate on ss_store_sk. Assuming that the rows are sorted by ss_store_sk, TFS of 2.8M rows and RGS of 724K rows, we can skip two of three files from the relevant partition and three out of four row groups within the relevant file that needs to be scanned. This leads to overall skipping benefits of 99.99% on store_sales.

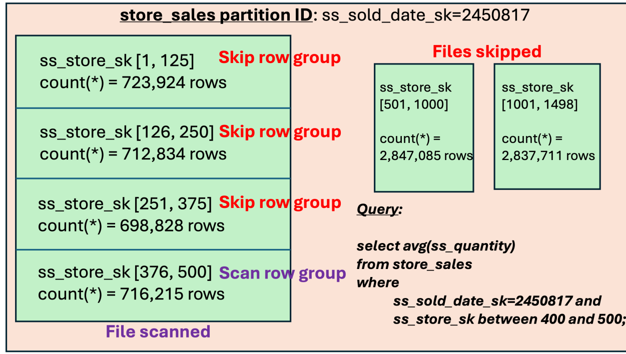


Fig. 3. Row group skipping in a TPC-DS store_sales file.

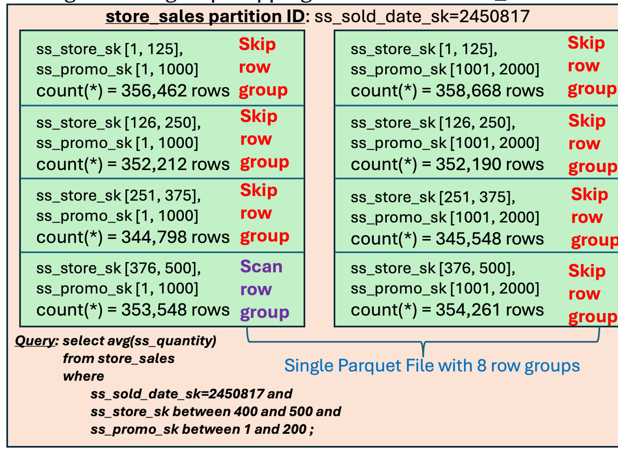


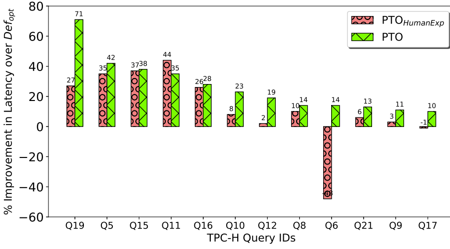
Fig. 4. Row Group skipping on Z-ordered store_sales

- **Sort Scheme:** We have an option to sort or bin-pack [2] an Iceberg table using the rewrite-data-files Spark procedure. Bin-packing is the default option which leaves the data files unsorted but packs enough rows into each file until the pre-specified RGS and TFS are met. In the example query from Figure 4, there are three filter predicates on ss_sold_date_sk, ss_store_sk and ss_promo_sk. Since ss_sold_date_sk is already used as a partition column, it achieves partition skipping. If we sort the table using Z-ordering [7, 15, 19] on ss_store_sk and ss_promo_sk, we can skip 7 out of 8 row groups using an RGS of 359K rows.

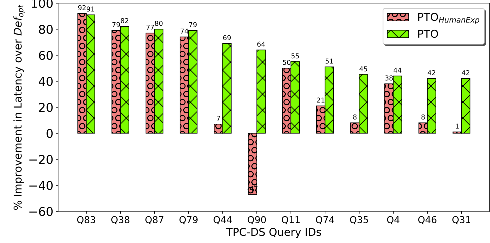
Appendix B Configuration of ML Baselines

(1) **Genetic Optimization (*GO*) :** This is a semi-supervised approach [11] in which the training labels are collected iteratively. In each iteration, the candidate data layouts are selected, scored, mutated using crossover techniques. The crossover is done between two training samples which have the highest skipping benefits (scores) by interchanging their partitioning columns, sort strategies or a combination of TFS and RGS choices to yield competent offsprings which are added to the training data. We use sampled tables to estimate the skipping benefits for training data layouts as detailed in Algorithm 3 in the paper.

(2) **Bayesian Optimization (*BO*) :** Unlike other baselines which require TFS and RGS to be discretized, *BO* can work with continuous spaces. It iteratively separates the training data layout samples using Tree Parzen Estimator (TPE) [4, 9, 10, 17, 21, 23] into good and bad regions



(a) % Benefit in query execution latency over Def_{opt} (TPC-H)



(b) % Benefit in query execution latency over Def_{opt} (TPC-DS)

Fig. 5. PTO vs. PTO$_{HumanExp}$ on queries yielding highest % improvement in query execution latency over Def_{opt}.

based on their scores reflecting the skipping ability. Next, TPE applies the Gaussian kernel density estimation (KDE) and maximizes $\frac{KDE_{good}}{KDE_{bad}}$ to find a candidate data layout which clearly separates out the good from the bad regions based on their estimated cumulative density. This new training sample chosen in each iteration is scored on the sampled table and added to the good or bad region based on its score.

(3) **Hierarchical clustering (*$Hier_{clust}$*) :** We iteratively apply K-medoids [14] clustering and BIRCH [25] on the cluster whose medoid has the most skipping ability. We return the optimal data layout for the best medoid upon exhausting the training budget.
(4) **Hybrid BO using surrogate models (*BO_{surr}*) :** A vanilla implementation of Bayesian optimization is limited by how many training data layouts can be scored on the sampled tables across a given #iterations. Instead, BO using surrogate models [6, 12] such as random forests which are used to predict the scores for training data layouts allow for more #iterations while staying under the given training budget. In each iteration, once BO finds the new sample, we can get its predicted *surrogate score* from a random forest which is re-trained periodically with actual scores.
(5) **Random search (*Rand*) :** This is an HPO baseline [5] which samples and scores random layouts iteratively as training data.

Appendix C Queries yielding highest improvement and regressions

Figure 5 shows the top-12 queries over which PTO yields the highest %improvement in execution latency over Def_{opt}. Interestingly, both in the case of TPC-H (Figure 5a) and TPC-DS (Figure 5b), PTO outperforms PTO$_{HumanExp}$ on a majority of queries yielding savings on execution latencies by 10%-71% on TPC-H and 42%-91% on TPC-DS over Def_{opt}. This is because PTO$_{HumanExp}$ is not generic enough to discover optimal layouts for all the tables and resorts to using binpacking on catalog_returns and inventory. Such suboptimal sort order choices impact PTO$_{HumanExp}$ whereas PTO on the other hand, picks optimal sort order choices which maximize the #queries which are benefitted. Several of these queries have long latencies which translates to the query benefits ranging from 1.11×-3.43× on TPC-H and 1.7×-11× on TPC-DS. Among the regressed queries in TPC-DS, we have Q23a, Q23b, Q67, Q72 as the long-running queries containing filter predicates that prioritize a few candidate sort columns over others. For example, PTO uses Z-ordering over multiple columns for catalog_sales but sorting over a single column cs_bill_customer_sk turns out to be more beneficial for Q23b. To address this problem, we prioritize long running queries in PTO using a weighted implementation evaluated in Section F. It is crucial to note that other

parameters like TFS, RGS are also tuned to benefit the long running queries. Likewise, in TPC-H, Q18 is a long-running query which can benefit from sorting the orders table on o_orderkey and o_custkey. A few regressed queries such as TPC-H Q4 are not long-running but they do not have popular filter columns that PTO chose for sorting.

Appendix D Evaluating Sort Cluster Size Estimation

Table 1. Optimal Sort Schemes detected by varying Sort Cluster Size Estimators (TPC-DS SF10K)

Table Name	APPROX DISTINCT		GOOD TURING		FIXED SAMPLING	
	Optimal Sort Order	#Queries benefited	Optimal Sort Order	#Queries benefited	Optimal Sort Order	#Queries benefited
catalog_returns	cr_item_sk	4	cr_item_sk, cr_return_amount	11	cr_item_sk, cr_return_amount	5
catalog_sales	z-order(cs_bill_cdemo_sk, cs_bill_addr_sk, cs_sold_time_sk)	8	z-order(cs_bill_cdemo_sk, cs_bill_addr_sk, cs_sold_time_sk, cs_bill_customer_sk)	9	cs_item_sk, cs_promo_sk, cs_bill_addr_sk	13
inventory	inv_quantity_on_hand	2	inv_quantity_on_hand	2	inv_item_sk	3
store_returns	sr_reason_sk	1	sr_reason_sk	1	sr_item_sk	3
store_sales	ss_store_sk	17	ss_store_sk	17	z-order(ss_store_sk, ss_promo_sk, ss_item_sk)	30
web_returns	wr_item_sk, wr_refunded_cdemo_sk, wr_return_amt	4	wr_refunded_addr_sk	1	z-order(wr_refunded_addr_sk, wr_refunded_cdemo_sk)	1
web_sales	z-order(ws_ship_date_sk, ws_web_site_sk)	3	z-order(ws_sold_time_sk, ws_bill_addr_sk, ws_ship_addr_sk, ws_ship_hdemo_sk)	9	z-order(ws_sold_time_sk, ws_ship_date_sk, ws_ship_hdemo_sk, ws_web_page_sk)	8

Table 1 shows the optimal sort orders discovered by varying the sort cluster size estimator. Note that #queries benefited is computed from the set union of queries in which the sort columns act as filtering columns. The total number of queries benefited eventually also depends on target file size (TFS), row group size (RGS) and partitioning column which we are excluding from this discussion. This is to exclusively understand the impact of sort strategies and their corresponding filter column frequency which are affected by varying the sort cluster size estimators. We can observe that the sort strategies discovered by PTO while using fixed sampling as the sort cluster size estimator benefit the highest #queries especially on the larger fact tables such as store_sales, catalog_sales and inventory. This is followed by Good-Turing estimator which benefits the highest #queries on catalog_returns and web_sales. APPROX DISTINCT benefits the highest #queries only on the web_returns table. This establishes that sort cluster size estimation should be balanced with #queries benefited to ensure that the detected layout brings significant latency savings on the overall query workload.

Appendix E Comparison with MDDL [8]

Algorithm 1 describes the discovery of the multidimensional data layout (MDDL) [8]. We pass the table, the scan fragment filter predicates and a latency threshold as input parameters to the algorithm. The maximum number of predicates used to represent the MDDL bit vector is set to 31 (which is the maximum bits supported by a 4-byte integer) as mentioned in the original work [8] (line 1 of the algorithm). Lines 2 and 3 shortlist the top-k (≤ MAX_BITS) scan fragment predicates which are highly selective.

Subsequently, the shortlisted filter predicates are evaluated on the entire table to create a dictionary with the row identifier as the key and a list of predicate identifiers as the value for all the rows in the table (line 4). This is a step that incurs high latency and therefore, we have a latency threshold that is set to 3 hours per table exclusively for this step of predicate evaluation. This limits

Table 2. Top-k Sort Predicates detected by MDDL to create multidimensional sort key for TPC-DS SF10K tables.

Table	Top-k Predicates used for bitwise sort key	% Selectivity
catalog_returns	cr_returned_date_sk BETWEEN 2452126 AND 2452140	0.56
	cr_return_amount > 10000	0.86
	cr_returned_date_sk BETWEEN 2452133 AND 2452163	1.09
	cr_returned_date_sk BETWEEN 2452580 AND 2452609	1.34
	cr_returned_date_sk BETWEEN 2450815 AND 2451179	8.49
	cr_item_sk BETWEEN 13 AND 299982	74.6
catalog_sales	(cs_bill_addr_sk BETWEEN 2 AND 5999993) AND (cs_sold_date_sk BETWEEN 2451270 AND 2451299)	0.18
	(cs_bill_addr_sk BETWEEN 2 AND 5999993) AND (cs_sold_date_sk BETWEEN 2450935 AND 2450965)	0.18
	(cs_bill_addr_sk BETWEEN 1 AND 6000000) AND (cs_sold_date_sk BETWEEN 2452184 AND 2452214)	0.41
	(cs_bill_customer_sk) IS NULL	0.5
	(cs_item_sk BETWEEN 19 AND 299946) AND (cs_sold_date_sk BETWEEN 2450905 AND 2450934)	0.71
	(cs_sold_date_sk BETWEEN 2452366 AND 2452396) AND (cs_item_sk BETWEEN 4 AND 299999)	0.73
inventory	(inv_item_sk BETWEEN 3389 AND 296474) AND (inv_quantity_on_hand BETWEEN 100 AND 500)) AND (inv_date_sk BETWEEN 2450971 AND 2451031)	0.85
	(inv_item_sk BETWEEN 35446 AND 278356) AND (inv_quantity_on_hand BETWEEN 100 AND 500)) AND (inv_date_sk BETWEEN 2452070 AND 2452160)	1.15
	(inv_item_sk BETWEEN 13 AND 299982) AND (inv_date_sk BETWEEN 2451654 AND 2451714)	2.57
	inv_date_sk BETWEEN 2451911 AND 2451969	3.07
	inv_date_sk BETWEEN 2451576 AND 2451941	19.9
store_returns	(sr_return_amt) > 10000	0.16
	sr_returned_date_sk BETWEEN 2452126 AND 2452140	0.47
	sr_returned_date_sk BETWEEN 2452488 AND 2452518	1.01
	sr_returned_date_sk BETWEEN 2452133 AND 2452163	1.08
	(sr_reason_sk) = 3	1.38
	sr_returned_date_sk BETWEEN 2451270 AND 2451391	5.43
	sr_returned_date_sk BETWEEN 2452366 AND 2452579	9.06
store_sales	(ss_net_profit BETWEEN 50.00 AND 300.00) AND (ss_sales_price BETWEEN 50.00 AND 200.00) AND ((not(ss_store_sk IS NULL)) AND (ss_addr_sk BETWEEN 1 AND 5999996) AND (ss_sold_date_sk BETWEEN 2451911 AND 2452275) AND (ss_hdemo_sk BETWEEN 120 AND 6479) AND (ss_cdemo_sk BETWEEN 33 AND 1920800)	0.09
	(ss_addr_sk BETWEEN 2 AND 5999993) AND (ss_sold_date_sk BETWEEN 2451270 AND 2451299)	0.16
	(ss_addr_sk BETWEEN 2 AND 5999993) AND (ss_sold_date_sk BETWEEN 2450935 AND 2450965)	0.17
	(ss_addr_sk BETWEEN 1 AND 6000000) AND (ss_sold_date_sk BETWEEN 2452184 AND 2452214)	0.39
	(ss_net_profit BETWEEN 0.00 AND 25000.00) AND (ss_sales_price BETWEEN 50.00 AND 200.00)) AND (not(ss_store_sk IS NULL)) AND (ss_addr_sk BETWEEN 3 AND 6000000) AND (ss_sold_date_sk BETWEEN 2451911 AND 2452275) AND (ss_cdemo_sk BETWEEN 3 AND 1920788)	0.54
	(ss_sold_date_sk BETWEEN 2451549 AND 2451579) AND (ss_item_sk BETWEEN 1 AND 299998)	0.7
web_returns	(wr_return_amt) > 10000	0.84
	wr_returned_date_sk BETWEEN 2452133 AND 2452163	1.48
	(wr_refunded_addr_sk BETWEEN 8 AND 5999999) AND (wr_returning_cdemo_sk BETWEEN 1 AND 1920800) AND (wr_refunded_cdemo_sk BETWEEN 1 AND 1920800)	17.09
	wr_returned_date_sk BETWEEN 2451545 AND 2451910	19.17
	wr_item_sk BETWEEN 3 AND 299977	74.62
web_sales	(ws_ship_hdemo_sk IS NULL)	0.25
	(ws_ship_date_sk BETWEEN 2452396 AND 2452456) AND (ws_ship_addr_sk BETWEEN 508 AND 5999932) AND (ws_web_site_sk BETWEEN 2 AND 36)	1.55
	(ws_bill_addr_sk BETWEEN 2 AND 5999993) AND (ws_sold_date_sk BETWEEN 2451270 AND 2451299)	1.76
	(ws_bill_addr_sk BETWEEN 2 AND 5999993) AND (ws_sold_date_sk BETWEEN 2450935 AND 2450965)	1.82
	(ws_ship_date_sk BETWEEN 2451270 AND 2451330) AND (ws_ship_addr_sk BETWEEN 27 AND 5999958) AND (ws_web_site_sk BETWEEN 2 AND 36)	1.83
	(ws_bill_addr_sk BETWEEN 1 AND 6000000) AND (ws_sold_date_sk BETWEEN 2452184 AND 2452214)	4.14
	(ws_net_profit BETWEEN 50.00 AND 300.00) AND (ws_sales_price BETWEEN 50.00 AND 200.00) AND (ws_web_page_sk IS NOT NULL) AND (ws_sold_date_sk BETWEEN 2451911 AND 2452275)	7.07
	(ws_item_sk BETWEEN 19 AND 299946) AND (ws_sold_date_sk BETWEEN 2450905 AND 2450934)	7.09
	(ws_sold_date_sk BETWEEN 2452356 AND 2452386) AND (ws_item_sk BETWEEN 8 AND 300000)	7.32
	(ws_promo_sk BETWEEN 1 AND 1500) AND (ws_item_sk BETWEEN 3 AND 299977) AND (ws_sold_date_sk BETWEEN 2452491 AND 2452521)	12.5
	(ws_net_profit) > 1.00) AND ((ws_net_paid) > 0.00)) AND ((ws_quantity) > 0) AND (ws_sold_date_sk BETWEEN 2451880 AND 2451910)	13.14

Algorithm 1 createMDDL($table$, $scanFragPreds$, $latencyThres$)

1: MAX_BITS $\leftarrow$ 31
2: $sel \leftarrow getSelectivities(table, scanFragPreds)$
3: $shortlistedPreds \leftarrow shortlistPreds(table, sel,$ MAX_BITS$)$
4: $rowToPreds \leftarrow eval(table, latencyThres, shortlistedPreds)$
5: $mddlSortKey \leftarrow createBitVector(table, rowToPreds)$
6: **return** $mddlSortKey$

the # predicates to being highly selective and avoids the evaluation of predicates which evaluate to true for most of the rows in the table thereby saving discovery time while also discovering important predicates. Finally, the top-k evaluated predicates are converted into a bit vector which is used to sort the table (line 5).

Table 2 shows the top-k MDDL predicates for each of the 7 fact tables. Note that for the remaining 14 dimension tables, we use the same optimized layouts as discovered by PTO for a fair comparison. Interestingly, we can observe that for the three largest fact tables, store_sales, catalog_sales and web_sales, the highest selectivities are 0.7%, 0.73% and 13.14% respectively. Even with such highly selective predicates, MDDL+PTO manages to show decent savings in workload latencies, when combined with the other optimal parameter choices discovered by PTO as shown earlier in Section 4.3.3 of the paper [16]. Only in the case of catalog_returns and web_returns, we encounter top-k predicates which exceed a selectivity of 74%. Thus, if we want to detect best data layouts with MDDL under reasonable latency limits, we need to ensure that we pass it candidate selection predicates from our pre-discovered scan fragments which are highly selective instead of atomic predicates which incur high latency and may not be selective in the first place.

Appendix F Impact of Query Weighting

Table 3 shows the optimized layouts discovered by Weighted PTO and PTO for all the 24 TPC-DS tables. Among the 7 fact tables (i.e., sales, returns and inventory), the sort scheme differs for catalog_sales. While Weighted PTO selects cs_bill_customer_sk as the sort column that benefits high latency queries, PTO z-orders the table upon four columns while subsuming cs_bill_customer_sk. This difference clearly explains how PTO in general, ends up benefiting more queries than its weighted counterpart. Another interesting difference is that Weighted PTO mostly pushes the TFS and RGS to be close to the upper limit of 512 MB and 128 MB more aggressively than PTO. A possible explanation is that the candidate sort columns shortlisted by Weighted PTO happen to score better w.r.t. skipping capabilities on the sampled tables with larger TFS and RGS thereby suggesting the GBT model to pick such candidates. Usually, the bias towards larger file and row group sizes happens when a decisive difference in skipping abilities among various layouts cannot be seen with smaller candidates i.e., the data distribution of the underlying samples is such that we can only see higher percentage of data skipping only when we increase the target file sizes.

Another interesting observation is that Weighted PTO partitions and z-orders the smaller dimension tables but PTO does not. This is because when the table sizes are so small that at least a 0.1% sample cannot be drawn (sampling extremely small tables leads to empty row sets), PTO automatically defaults to using binpacking and avoids partitioning the table. For TFS and RGS, it chooses reasonably low values. On the other hand, we allowed Weighted PTO to score the candidates to show the distinction between using the default layout choices for small tables and actually trying to apply the GBT model on them. Since sampling leads to empty sets for tables such as income_band, reason, ship_mode etc., all the training candidates get the same scores. Not so surprisingly, that results in partitioning the table and choosing a z-ordering strategy for it even

Table 3. Optimized layouts configured by weighted PTO and vanilla PTO for TPC-DS SF 10K.

Table Name	Total Size	Weighted PTO: Partitioning Column	Weighted PTO: TFS (MB)	Weighted PTO: RGS (MB)	Weighted PTO: Sort Scheme	PTO: Partitioning Column	PTO: TFS (MB)	PTO: RGS (MB)	PTO: Sort Scheme
catalog_returns	83.8 GB	cr_returned_date_sk	128	128	(cr_item_sk,cr_return_amount)	cr_returned_date_sk	128	32	(cr_item_sk,cr_return_amount)
catalog_sales	763 GB	cs_sold_date_sk	512	128	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	inv_date_sk	512	128	inv_quantity_on_hand	inv_date_sk	512	128	inv_quantity_on_hand
store_returns	111 GB	sr_returned_date_sk	512	128	sr_reason_sk	sr_returned_date_sk	256	64	sr_reason_sk
store_sales	893.8 GB	ss_sold_date_sk	512	128	ss_store_sk	ss_sold_date_sk	512	128	ss_store_sk
web_returns	39.2 GB	wr_returned_date_sk	512	128	wr_refunded_addr_sk	wr_returned_date_sk	512	128	wr_refunded_addr_sk
web_sales	535.5 GB	ws_sold_date_sk	512	128	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)
customer	2.42 GB	c_birth_month	512	128	c_current_cdemo_sk	-	256	64	z-order(c_current_cdemo_sk, c_preferred_cust_flag, c_birth_month)
call_center	15.7 KB	-	32	32	binpack	-	32	32	binpack
catalog_page	1.44 MB	-	32	32	binpack	-	32	32	binpack
customer_address	490 MB	ca_state	512	128	ca_country,ca_gmt_offset	-	256	64	ca_state,ca_gmt_offset
customer_demographics	8.4 MB	cd_purchase_estimate	32	8	z-order(cd_marital_status, cd_gender)	-	32	8	zorder(cd_education_status, cd_marital_status, cd_gender)
date_dim	1.3 MB	d_year	32	8	zorder(d_moy,d_qoy, d_month_seq)	-	32	8	d_moy
household_demographics	30 KB	-	32	8	z-order(hd_dep_count, hd_vehicle_count, hd_buy_potential)	-	32	32	binpack
income_band	1.13 KB	-	32	8	ib_income_band_sk	-	32	8	binpack
item	28.3 MB	i_manufact_id	32	8	z-order(i_category, i_color, i_class)	-	64	64	binpack
promotion	66.1 KB	-	32	8	z-order(p_channel_email, p_channel_tv)	-	32	32	binpack
reason	1.72 KB	-	32	8	r_reason_sk	-	32	32	binpack
ship_mode	2.38 KB	-	32	8	sm_carrier	-	32	32	binpack
store	97.7 KB	-	32	8	z-order(s_store_name, s_gmt_offset)	-	32	32	binpack
time_dim	958 KB	t_hour	32	32	t_mealtime	-	32	32	binpack
warehouse	5.68 KB	-	32	32	binpack	-	32	32	binpack
web_page	58.36 KB	-	32	8	wp_char_count	-	32	32	binpack
web_site	16.6 KB	-	32	32	binpack	-	32	32	binpack

though that does not lead to any benefit. This is why, we present most of the results on the 7 large, fact tables in the main body of the paper [16].

References

[1] 2015. Dremio Unified Lakehouse Platform. https://www.dremio.com/platform/.

[2] 2017. Apache Iceberg: Spark procedures. https://iceberg.apache.org/docs/nightly/spark-procedures/.

[3] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2326–2339. doi:10.1145/3514221.3526054

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger (Eds.). 2546–2554. https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html

[5] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13 (2012), 281–305. doi:10.5555/2503308.2188395

[6] Eric Brochu, Vlad M. Cora, and Nando de Freitas. 2010. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *CoRR* abs/1012.2599 (2010). arXiv:1012.2599 http://arxiv.org/abs/1012.2599

[7] Databricks. 2024. Data skipping for Delta Lake. https://docs.databricks.com/en/delta/data-skipping.html.

[8] 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

[9] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 1436–1445. http://proceedings.mlr.press/v80/falkner18a.html

[10] Matthias Feurer and Frank Hutter. 2019. Hyperparameter Optimization. In *Automated Machine Learning - Methods, Systems, Challenges*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). Springer, 3–33. doi:10.1007/978-3-030-05318-5_1

[11] David E. Goldberg. 1989. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley.

[12] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers (Lecture Notes in Computer Science, Vol. 6683)*, Carlos A. Coello Coello (Ed.). Springer, 507–523. doi:10.1007/978-3-642-25566-3_40

[13] 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

[14] Leonard Kaufman and Peter J. Rousseeuw. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley. doi:10.1002/9780470316801

[15] Dipankar Mazumdar. 2022. How Z-Ordering in Apache Iceberg Helps Improve Performance. https://www.dremio.com/blog/how-z-ordering-in-apache-iceberg-helps-improve-performance/.

[16] Venkata Vamsikrishna Meduri, David Kreismann, Ronald Barber, and Berthold Reinwald. 2026. PTO: A Workload-driven Predictive Table Optimizer for Lakehouse Systems.

[17] Yoshihiko Ozaki, Yuki Tanigaki, Shuhei Watanabe, and Masaki Onishi. 2020. Multiobjective tree-structured parzen estimator for computationally expensive optimization problems. In *GECCO '20: Genetic and Evolutionary Computation Conference, Cancún Mexico, July 8-12, 2020*, Carlos Artemio Coello Coello (Ed.). ACM, 533–541. doi:10.1145/3377930.3389817

[18] 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

[19] Matthew Powers. 2023. Delta Lake Z Order. https://delta.io/blog/2023-06-03-delta-lake-z-order/.

[20] 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

[21] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 2960–2968. https://proceedings.neurips.cc/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html

[22] 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

[23] Shuhei Watanabe. 2023. Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *CoRR* abs/2304.11127 (2023). arXiv:2304.11127 doi:10.48550/ARXIV.2304.11127

[24] 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

[25] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 103–114. doi:10.1145/233269.233324

Received July 2025; revised October 2025; accepted November 2025