Instance-Optimized Data Layouts for Cloud Analytics Workloads

Jialin Ding[†] Umar Farooq Minhas[‡] Badrish Chandramouli[‡]

Chi Wang[‡] Yinan Li[‡] Ying Li[§] Donald Kossmann[‡] Johannes Gehrke[§] Tim Kraska[†]

[†]Massachusetts Institute of Technology [‡]Microsoft Research [§]Microsoft

ABSTRACT

Today, businesses rely on efficiently running analytics on large amounts of operational and historical data to gain business insights and competitive advantage. Increasingly, such analytics are run using cloud-based data analytics services, such as Google BigQuery, Microsoft Azure Synapse, Amazon Redshift, and Snowflake. These services persist and process data in compressed, columnar formats, stored in large blocks, each of which contains thousands or millions of records. For these services, disk I/O from (remote) cloud storage is often one of the dominant costs for query processing. To reduce the amount of I/O, services often maintain per-block metadata, such as zone maps, which are used to skip blocks that are irrelevant to the query, leading to lower query execution times. However, the effectiveness of block skipping via zone maps is dependent on how the records are assigned to blocks. Recent work on *instance-optimized* data layouts aims to maximize block skipping by specializing the block assignment strategy to a specific dataset and workload. However, these existing approaches only optimize the layout for a single table.

In this paper, we propose MTO, an instance-optimized data layout framework that determines the blocking strategy for all tables in a multi-table dataset in the presence of joins, such as in a star or snowflake schema common in real-world workloads. MTO takes advantage of sideways information passing through joins to jointly optimize the layout for all tables, which results in better block skipping and hence reduced query execution times. Experiments on a commercial cloud-based analytics service show that MTO achieves up to 93% reduction in blocks accessed and 75% reduction in end-to-end query times compared to state-of-the-art blocking strategies.

ACM Reference Format:
Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457270

1 INTRODUCTION

To efficiently process increasingly larger volumes of data, modern cloud-based data analytics services use a variety of techniques to reduce data movement and data access. One standard technique is to use columnar storage to avoid accessing columns that are not relevant to a query. Data is often persisted in remote cloud storage, such as Amazon S3, and accessed by "compute nodes" during query processing. These systems group data records into large blocks, each with hundreds of thousands or millions of records in order to maximize compression ratios. To maximize throughput and minimize I/O operations per second, during query processing, a block (or a subset of columns from a block) is the smallest unit of I/O from cloud storage.

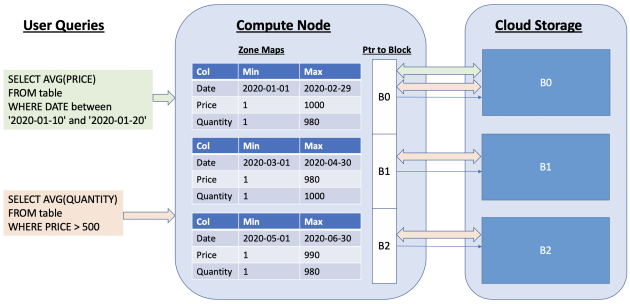

Figure 1: Zone maps over three data blocks. Using zone maps, the first query is able to skip blocks 1 and 2, whereas the second query cannot skip any blocks.

To avoid accessing blocks that are not relevant to a query, per-block metadata, which is often cached in memory, is used to skip blocks during query processing. The most common form of per-block metadata is zone maps [6, 8, 33, 39], which store the minimum and maximum value for each column in a data block. For example (Fig. 1), if a block's zone map shows that the records in the block span dates from March to April 2020, and the query filters for records with dates in January 2020, then this particular block does not have to be read from storage (i.e., can be skipped) during this query's execution.

Zone maps are cheap to maintain and potentially useful, but their effectiveness at block skipping is highly dependent on how records are assigned to blocks (i.e., the data layout). By default, most systems usually sort each table by a certain sort column (e.g., the date column), and will place contiguous chunks of records into the same block. Under this basic blocking scheme, queries that filter over the sort column will be able to skip blocks based on zone maps, but filters over other columns do not provide much skipping opportunity (Fig. 1). Z-order [34] is a multi-dimensional sorting technique, often deployed in practice for its simplicity. However, for Z-order to be effective for block skipping, the columns on which to define the Z-order and their relative order must be manually and carefully selected, and poor tuning can actually result in degraded performance.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457270

To overcome the shortcomings of existing data layout techniques, the idea of *instance-optimized data layouts* was recently proposed. These approaches "learn" a specialized blocking scheme (i.e., a sort order) that achieves high block skipping performance for a specific dataset and workload [11, 27, 35, 56]. Experiments on synthetic and real-world datasets and workloads show that instance-optimized data layouts can be orders of magnitude better in block skipping compared to simple sort-based data layouts as well as more advanced, fine-grained data skipping techniques [47, 48]. The key idea is to use the knowledge of the data distribution and query workload (e.g., the specific filter predicates) to custom-fit the data layout. However, existing instance-optimized layouts can only optimize a single table's layout, for a query workload that only queries that table. In practice, analytics workloads typically contain many tables and the queries use diverse join patterns, such as in a star or snowflake schema.

One naïve approach to optimizing the layout for a multi-table dataset is to independently optimize each table's layout using an existing instance-optimized approach. However, as we show later, this approach does not perform significantly better, as it does not exploit knowledge about the joins. In this paper, we propose MTO (Multi-Table Optimizer), the first instance-optimized data layout framework for optimizing whole datasets. Our key idea is to pass additional information about joins, which we refer to as *sideways information passing (SIP)*, through *join-induced predicates*, to jointly optimize the layout for all tables, simultaneously. This idea is inspired by prior work on SIP [21]; we discuss similarities and differences in Section 3.1. Furthermore, existing instance-optimized layout techniques [11, 35, 56] must re-optimize the entire layout in response to changes in the query workload. In contrast, MTO gracefully responds to workload changes through partial layout reorganization. We summarize our contributions as follows:

(1) We propose MTO, the first instance-optimized data layout framework for multi-table datasets. MTO aims to minimize the overall number of blocks accessed in an analytics workload with join queries, which are common in practice.
(2) We introduce join-induced predicates, used in MTO to pass information through joins. We present algorithms that exploit join-induced predicates to "learn" better data layouts.
(3) We introduce further practical techniques to ensure that MTO scales to larger datasets and query workloads and adapts to workload shift and data changes.
(4) We evaluate MTO, both in simulations and by integrating with a commercial cloud-based data analytics service to measure end-to-end gains. We compare MTO against existing instance-optimized layouts and user-tuned blocking schemes.

In the rest of this paper, we provide background (Section 2), introduce MTO's high-level design (Section 3), examine the details of MTO's algorithms (Sections 4 and 5), present experimental results (Section 6), review related work (Section 7), and conclude (Section 8).

2 CURRENT BLOCKING APPROACHES

As shown in Fig. 1, zone maps are useful for skipping irrelevant blocks during query execution, but their effectiveness depends on the physical layout of the data among blocks (i.e., the sort order). We now describe existing approaches for data layout.

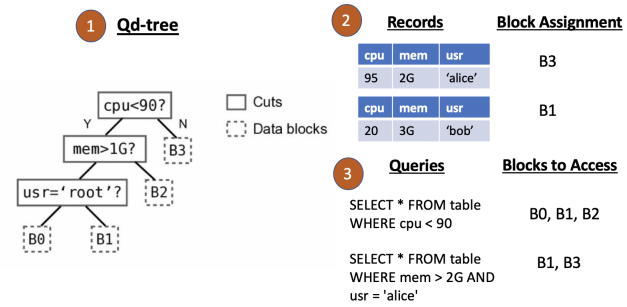

Figure 2: (1) Qd-tree defines blocks using cuts. (2) Qd-tree is used *offline* to route records to the blocks they are stored in and (3) is used *online* to determine which blocks need to be accessed during query execution.

Sort Key. A common approach used in practice is to sort each table's data by a particular column. For example, by sorting on timestamp/date, any queries that only filter over the past day of data can skip all blocks that contain data that is older than one day.

Z-ordering. One drawback to the sort key approach is that only queries that filter over the sort key column can benefit from block skipping. Z-ordering [34] "sorts" data over multiple columns simultaneously, and it is supported by several commercial systems [9, 59]. However, Z-ordering must be tuned carefully to achieve high performance. For example, a DBA must decide which columns to include in the Z-order, and whether to give more weight to certain columns over others. A poorly tuned Z-ordering can degrade block skipping performance compared to the sort key approach. Even when properly tuned, Z-ordering underperforms instance-optimized approaches.

Instance-optimized Layouts Instance-optimized layouts are specialized to perform well (e.g., achieve low overall query runtime) on a particular dataset and workload [11, 27, 35, 56]. By purposefully overfitting the layout for a specific dataset and workload, instance-optimized layouts are able to outperform existing approaches on that specific instance (dataset and workload).

Drawback of Current Approaches One major drawback of all the approaches described above is that they optimize the layout for a single table. In a dataset with multiple tables, existing approaches would optimize each table's layout independently. By not considering the layout of all tables jointly, existing approaches do not maximize block skipping performance, as we show later through experiments.

2.1 Qd-tree

We now describe qd-tree [56], an existing instance-optimized data layout framework for single tables, which we use as a fundamental building block in our work. The intuition behind qd-tree is to tailor the block assignment strategy for a given query workload to reduce the number of blocks accessed when running that workload. For example, consider a workload consisting of a single query:

```
SELECT * FROM table
WHERE X > 10 AND Y IN (1, 2, 3)
```

Let us divide the records of `table` into those that satisfy `X > 10` and those that do not. If we assign each set of records to a separate group

of blocks (e.g., records that satisfy the predicate are assigned to blocks 1 and 3, while records that do not go in blocks 2 and 4), then during query processing, we only need to access the blocks corresponding to the "satisfying" set. We can apply similar logic to divide and block the records based on whether they satisfy the other predicate, `Y IN (1, 2, 3)`. However, if we sort/block the records based on their value in some unrelated column (e.g., `Z`), the query will likely need to access all blocks. This example illustrates that by blocking based on the specific filter predicates that appear in the query workload, we can reduce the number of blocks accessed during query processing.

We now provide a high-level overview of qd-tree, which we describe in more detail in the following subsections. The qd-tree workflow is as follows (Fig. 2):

(1) The input to the workflow is a table and workload of queries that the user expects to run on the table. Using (a sample of) the table and query workload, construct a decision tree (which we call a qd-tree) that roughly evenly splits the records of the (sampled) table into data blocks (Section 2.1.3).
(2) Offline, use the qd-tree to assign the table's records to blocks. This process is called *routing* a record (Section 2.1.2).
(3) At query execution time, use the same qd-tree to determine which blocks the query needs to access (and therefore which data blocks can be skipped). This process is called *routing* a query (Section 2.1.2).

2.1.1 Qd-tree Structure. The qd-tree (Fig. 2) is a binary decision tree. Each node corresponds to some subset of records in the table. The root node corresponds to all records in the table. Each inner node contains a filter predicate, which we call a *cut*. The node's cut is used to divide its subset of records into two smaller subsets, one with records that satisfy the cut and the other with records that do not. The left child inherits the "yes" subset, and the right child inherits the "no" subset. The leaf nodes of the qd-tree correspond to data blocks. That is, the subset of records corresponding to a leaf node are assigned to the same data block.

2.1.2 Qd-tree Usage. We can use a qd-tree for both offline block assignment and online query processing. Given a qd-tree and a table, we route each record in the table through the qd-tree to assign it to the data block that it will be stored in. For example, consider the first record in Fig. 2. We route this record R through the qd-tree, from root to leaf. The root node's cut indicates that records that satisfy `cpu < 90` are inherited by the left child, while records that do not satisfy `cpu < 90` are inherited by the right child. Since R does not satisfy the root node's cut, we route R to the right child. The right child is a leaf node, and therefore we assign R to block 3. In the same manner, every record is assigned to a block.

At query execution time, we use the qd-tree to determine which blocks need to be accessed. For example, consider the first query in Fig. 2. We route this query through the qd-tree, from root to leaf. At the root node, the query only filters for records that could appear in the left child (i.e., any records that do not satisfy the cut `cpu < 90` are irrelevant), so we route the query to the left child. At this second node, the query could filter for records that appear in either child (i.e., records that satisfy and do not satisfy `mem > 1G` could both be relevant), so we route the query to both children. We continue to recurse in this manner, and at the end, we find that the query must access blocks 0, 1, and 2. Note that while records are always routed to exactly one block, queries can be routed to multiple blocks.

2.1.3 Qd-tree Construction Algorithm. We take the same greedy approach to construction as [56]: given a table and query workload, begin with all the records in a single block, i.e., the qd-tree has a single root node that contains all the records. In each iteration, we split a leaf node into two child nodes by applying a cut. Cuts are chosen from the set of *candidate cuts*, which is the set of filter predicates that appear in the query workload. For example, in the single-query workload described at the beginning of this section, there are two candidate cuts: `X > 10` and `Y IN (1, 2, 3)`. When choosing the cut for a node, we use the one amongst the candidate cuts that maximizes the number of records skipped by the resulting qd-tree over the given workload. We continue iterating until all leaf nodes have reached some desired size, measured by the number of records falling in the data block represented by that leaf node.

3 MTO OVERVIEW

In this section, we provide an overview of our approach, called MTO (Multi-Table Optimizer), that creates instance-optimized data layouts for multi-table datasets. For a multi-table dataset and a query workload, the goal of MTO is to learn an instance-optimized layout that maximizes block skipping for that specific dataset and workload. MTO consists of two parts: (1) a mapping of records to blocks, where each block has roughly the same number of records, which we call the *block size*[1]. A block can only contain records from a single table. (2) At execution time, given a query, a method to identify which blocks need to be accessed (and by proxy, which blocks can be skipped).

We first introduce the key idea that differentiates MTO from existing instance-optimized approaches: sideways information passing using join-induced predicates. We then describe MTO's end-to-end workflow. We present more details in Sections 4 and 5.

3.1 Sideways Information Passing

Existing single-table layout approaches are sub-optimal in the multi-table case because they do not take advantage of sideways information passing between tables. To provide intuition, we use the following running example (Fig. 3): let the block size be 1M records. Let our dataset have two tables: Table A with 1M records, and Table B with 8M records. All of Table A's records will fall in the same block, so the sort order for Table A does not impact block skipping. We are only interested in the blocking strategy for Table B. Consider a workload consisting of queries similar to the following:

```
SELECT COUNT(*) FROM A, B
WHERE A.KEY = B.KEY AND A.X < 100 AND B.Y > 200
```

By pushing down the two filter predicates, Table A's zone maps will be able to skip blocks based on the predicate `A.X < 100`, and Table B's zone maps will be able to skip blocks based on `B.Y > 200`. A DBA determining each table's layout independently would sort Table B's data by Y. Then the execution engine will use zone maps to skip over B's blocks where $Y \leq 200$, as shown in Fig. 3.

However, we can achieve even more block skipping using sideways information passing. We know that any records of the joined

[1] On most cloud analytics services, the block size is preset automatically by the service and cannot be changed by the user, so we do not explore variable-sized blocks.

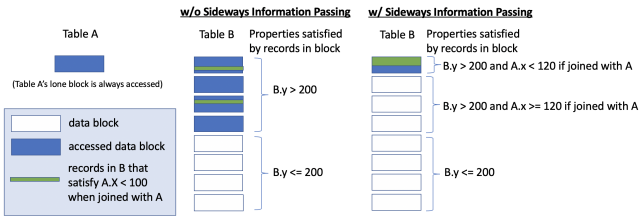

Figure 3: By taking advantage of sideways information passing to optimize the data layout, we can increase block skipping. Only blocks in the shaded regions are read.

Table 1: Join-induced predicate terminology example.

Term	Definition	Running Example
Simple predicate	Predicate over one table	`A.X < 100`
Join-induced predicate/cut	Predicate over columns in multiple tables, composed of nested semi-join subqueries	`A.BKEY IN (SELECT B.BKEY FROM B WHERE B.CKEY IN ( SELECT C.CKEY FROM C WHERE C.Z > 200))`
Literal cut	Result of evaluating subqueries in a join-induced cut	`A.BKEY IN (3, 14, 159)`
Source table	Table with the original predicate	`C`
Target table	Table whose predicate is induced	`A`
Source cut	Source table's predicate	`C.Z > 200`
Induction path	List of tables and join columns connecting source to target	C→CKEYB→BKEYA
Induction depth	Length of the induction path	2

relation that have $A.X \geq 100$ are irrelevant. Therefore, the records in B that produce irrelevant records when joined with A are themselves irrelevant. By grouping together the records of B based on whether the join with A would produce relevant records, we can further increase block skipping. Essentially, Table B's zone maps are skipping blocks based on two predicates: `B.Y > 200`, as well as a new *join-induced predicate*: `B.KEY IN (SELECT A.KEY FROM A WHERE A.X < 100)`. Table 1 defines relevant terminology through an example. We explain join-induced predicates in more detail in Section 4.1.

3.1.1 Relation to Existing Approaches. The idea of join-induced predicates is similar to some existing techniques. Semi-join reduction [3, 16] uses sideways information passing at execution time to filter rows (e.g., construct a bitmap over the build input of a hash join and use it to filter rows on the probe input before they reach the join). In contrast, join-induced predicates in MTO are used in an offline optimization stage to determine the data layout and introduce negligible overhead during query execution.

Similar to semi-join reduction, data-induced predicates (diPs) [21] pass information about the blocks selected by a predicate (e.g., the zone maps over selected blocks), through joins, which induces a predicate on the joined table's join column that can be used to skip blocks on the joined table. diPs are applied during query optimization, which avoids the overhead of performing sideways information passing at execution time and gives the optimizer extra information with which to find better plans. However, diPs are only beneficial to block skipping when certain conditions about the data layout are met, most importantly that the join column values in the blocks that satisfy a predicate contain only a small portion of all possible join column values, otherwise the induced predicate will not be selective enough to skip blocks when applied to the joining table.

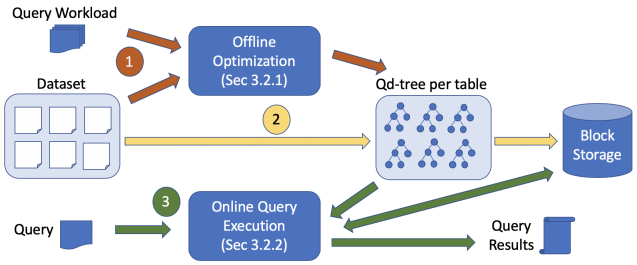

Figure 4: (1) In offline optimization, MTO produces a layout (a qd-tree per table) given a dataset and query workload. (2) MTO assigns records to blocks and stores them. (3) In online query execution, MTO skips blocks based on the layout.

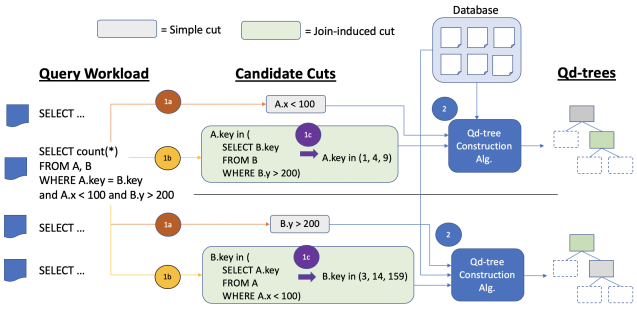

Figure 5: MTO optimization uses the query workload and dataset to create one qd-tree per table.

In contrast, MTO explicitly constructs the block layout to maximize opportunities for skipping blocks during execution. We show in our evaluation that this difference allows MTO to outperform diPs.

3.2 MTO Workflow

Our workflow (Fig. 4) has two components, corresponding to the two parts described at the beginning of this section: (1) offline optimization, and (2) online query execution.

3.2.1 Offline optimization. The MTO optimization algorithm takes a multi-table dataset and a query workload as input and creates one qd-tree per table, which will determine the data layout for that table's records. The algorithm has the following steps (Fig. 5):

(1) Do the following for each query (Fig. 5 shows the workflow for one particular query):
   (a) Extract all simple predicates (Table 1) from the query, and group them based on which table they filter. In Fig. 5, the example query contains two simple predicates: `A.x < 100` and `B.y > 200`. Therefore, `A.x < 100` is extracted for Table A and `B.y > 200` is extracted for Table B.
   (b) Pass the simple predicates extracted in Step 1a through joins to create *join-induced predicates*. In the example, the simple predicate `B.y > 200` is passed though the join to Table A, which produces the join-induced predicate `A.key IN (SELECT B.key FROM B WHERE B.y > 200)`, which filters Table A. Similarly, the simple predicate `A.x < 100` passes

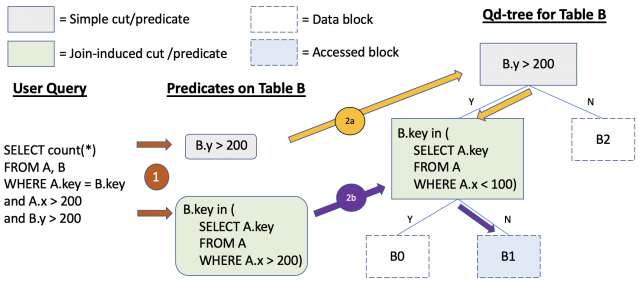

Figure 6: At query time, MTO uses the per-table qd-trees to determine which blocks to access from each table. This query only needs to read block 1 from Table B.

through the join to Table B to create a join-induced predicate on Table B. For queries with more complex join graphs (e.g., Table A joins with Table B, which joins with Table C), a simple predicate can be passed through multiple joins (e.g., a simple predicate on Table A is passed through Table B to Table C and produces a join-induced predicate on Table C).

(c) For each join-induced predicate, evaluate any subqueries to obtain the *literal cut*. In the example, running the subquery `(SELECT B.key FROM B WHERE B.y > 200)` returns the set `(1, 4, 9)`, so the literal form of the join-induced cut over Table A is `A.key IN (1, 4, 9)`.

(2) For each table independently: feed the table and overall query workload into the qd-tree construction algorithm (Section 2.1.3). The predicates over that table extracted in Step 1 (which could be either simple predicates or join-induced predicates) become the *candidate cuts* for the qd-tree. The constructed qd-tree determines that table's data layout.

Given the optimized layout, MTO assigns each table's records to data blocks using their respective qd-trees, as described in Section 2.1.

3.2.2 Online query execution. At query execution time, MTO uses the qd-tree for each table to determine which blocks need to be accessed. Following our running example, Fig. 6 shows the qd-tree that might be constructed for Table B by the MTO optimization algorithm. The qd-tree has three leaf nodes, so the records of Table B will be stored in three blocks (B0, B1, B2). The qd-tree uses two cuts: a simple cut `B.y > 200`, and a join-induced cut shaded in green. To determine which blocks of Table B need to be accessed when processing the user query in Fig. 6, MTO will do the following (a similar process would occur independently to determine blocks to access on Table A):

(1) Identify all predicates from the query on that table, including join-induced predicates, following the same procedure as Steps 1a and 1b (but not 1c) from Section 3.2.1.

(2) Use the predicates to route through the table's qd-tree to identify which blocks need to be accessed, using the process described in Section 2.1. Section 4.1.2 provides details about routing through join-induced cuts (e.g., Step 2b in the example).

Note that these steps are applied independently for each table, before execution occurs. Therefore, MTO will skip the same set of blocks regardless of the physical execution plan (e.g., the join order).

4 MTO ALGORITHMS

In this section, we provide details about join-induced predicates and also describe how MTO maintains low optimization times even when scaling to larger datasets. We use the terminology shown in Table 1.

4.1 Join-induced Predicates

4.1.1 When Can We Induce? MTO supports induction on source predicates that use $=, \neq, <, \leq, >, \geq$, `IN`, `NOT IN`, `LIKE`, and `NOT LIKE`, including predicates over multiple columns (e.g., `A.X < A.Y`), as well as any conjunctions or disjunctions of the above. We support predicate induction through equijoins over a single column, including inner, one-sided outer, semi, anti-semi, and self joins.

In the simple example in Section 3.2, we induced a predicate from Table A to Table B, and vice versa. However, in some cases we cannot induce a predicate from one table to another while maintaining a semantically equivalent query. Consider the following query:

```
SELECT AVG(A.Z) FROM A WHERE A.X < 100 AND A.Y < (
  SELECT COUNT(*) FROM B WHERE A.KEY = B.KEY AND B.Z > 200)
```

We can induce from A to B, producing the join-induced predicate `B.KEY IN (SELECT A.KEY FROM A WHERE A.X < 100)`. However, we cannot induce from B to A. To determine when predicates can be induced while maintaining a semantically equivalent query, we use the following set of rules, similar to those found in [21]:

- Predicates can be induced in both directions through inner joins; from the left to right side for a left outer join, and vice versa for right outer joins; in both directions through semi joins; and from the left to right side for a left anti-semi join, and vice versa for right anti-semi joins. Predicates cannot be induced through full outer joins.
- For self-joins, MTO logically creates two copies of the table, treats them as different tables, and applies the above rules.
- Predicates can be induced from an outer query into a correlated subquery [53] through any of the above rules. In the example query given above, a predicate from the outer query (`A.X < 100`) is induced into the correlated subquery (`SELECT COUNT(*) ...`) through an inner join (`A.KEY = B.KEY`).

Just because we *can* induce doesn't mean we *should*. In the optimization process, MTO only considers join-induced predicates whose induction paths are composed only of joins originating from columns with unique values (e.g., inducing from a dimension table into a fact table by joining on the dimension table's primary key, but not from a fact table's foreign key into a dimension table). This is not a fundamental limitation of join induction; instead, we enforce this restriction to make inserts and deletes more efficient (Section 5.2). We verified experimentally that this restriction has minimal impact on performance. Intuitively, this is because predicates induced from join columns with non-unique values tend to fall on smaller tables with fewer blocks (e.g., dimension tables), which limits the predicate's impact on the number of blocks skipped dataset-wide.

4.1.2 How Do We Use Them? Like simple cuts in the qd-tree, join-induced cuts are used to route records and queries down the tree. To route records, we use the *literal* join-induced cut in the same way as a simple cut. To route queries, the qd-tree checks for subsumption between the query and the *logical* join-induced cut: if the query's join graph does not share the join-induced cut's induction path, route the

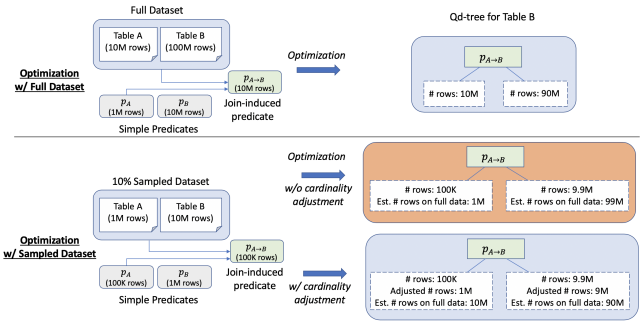

Figure 7: Cardinality adjustment allows MTO to achieve accurate block size estimates when optimizing based on a dataset sample, which improves the quality of the resulting layout.

query to both child nodes in the qd-tree. Otherwise, route to the left child if the query's filters on the source table intersect the source cut, and independently route to the right child if the query's filters on the source table intersect the *negation* of the source cut. For example, in Step 2b of Fig. 6, the query filter on the source table (`A.x > 200`) does *not* intersect the source cut (`A.x < 100`), but *does* intersect the negation of the source cut (`A.x ≥ 100`), so we only route to the right child.

Qd-tree nodes that use join-induced cuts must store both the logical and literal cuts. The logical cut (i.e., a query of nested semi-joins) is compact, but literal cuts can incur high memory costs, because the `IN` list can grow very large, especially over high-cardinality key columns. To reduce space usage, we compress `IN` lists as Roaring Bitmaps [5], which is the state-of-the-art bitmap compression technique [52].

4.2 Scalability through Sampling

To reduce the time needed for optimizing the layout when scaling to larger datasets, MTO runs its optimization algorithm on a uniform sample of the dataset instead of the full dataset. Given a sampling rate s, MTO creates a sample by selecting s fraction of records from each table in the dataset uniformly at random. For especially small tables (e.g., under 1K records), MTO simply uses the entire table, because sampling small tables does not meaningfully decrease optimization time. If the desired block size on the full dataset is b, MTO uses the adjusted block size $b \times s$ when optimizing based on the sampled dataset.[2]

It is well-known that the join of two uniform samples has quadratically fewer tuples than a sample of the original join [18]. MTO must account for this effect when evaluating the quality of join-induced cuts when optimizing on a sampled dataset. For example, Fig. 7 shows a dataset with 10M records in Table A and 100M records in Table B. The simple predicate p_B on Table B and the join-induced predicate $p_{A\to B}$ on Table B both select 10M records. However, on a sampled dataset with $s = 0.1$, p_B selects (10M)s=1M records, whereas $p_{A\to B}$ selects (10M)s^2=100K records. If MTO estimates block sizes on the full dataset as $1/s$ of the block sizes on the sample, then it produces inaccurate estimates of block size (e.g., the qd-tree shaded in orange in Fig. 7). Optimizing without taking this discrepancy into account may degrade the quality of the resulting layout.

To account for this effect, MTO attaches a value called the *cardinality adjustment (CA)* to every join-induced cut, defined as s^d, where d is the induction depth (e.g., the CA for $p_{X\to Y\to Z}$ is s^2). Therefore, in Fig. 7 the CA for $p_{A\to B}$ is s. The left block in the bottom qd-tree (which is constructed over a sample) has block size 100K records, but the cardinality-adjusted block size is (100K)/s = 1M records. This is then used to produce an accurate estimate of the block size on the full dataset.

Formally, let qd-tree node N cover r records of the sampled table, so that the estimated cardinality of N on the full dataset is r/s. Let N use join-induced cut p, so that the left child N_L covers r_L records and the right child N_R covers $r_R = r - r_L$ records. If p has a CA of k, then the estimated cardinality of N_L on the full dataset is not r_L/s. Instead, it is r_L/sk. Accordingly, the estimated cardinality of N_R on the full dataset is not r_R/s, but instead $r/s - r_L/sk$. Simple cuts have a CA of 1.

The CA for a block (i.e., a leaf node) is the product of CAs for all cuts on the traversal route from root to leaf. Adjustments caused by a particular join are not double-counted if multiple intersecting cuts along the traversal route have induction paths that contain that join.

5 WORKLOAD SHIFT AND DATA CHANGES

In this section, we describe how MTO can adapt to changes in the query workload and data.

5.1 Dynamic Workloads

MTO's layout is optimized for a given query workload. However, workload characteristics (e.g., join patterns, frequently filtered columns) often change over time, which may cause query performance on MTO's layout to degrade. In response, MTO can re-optimize its layout and physically reorganize blocks to specialize for the new workload. However, fully reorganizing a large dataset can require significant time and computational resources. Therefore, MTO has the ability to *partially* reorganize its layout. Intuitively, MTO only reorganizes qd-tree subtrees that result in the most overall performance gain. For example, if only the workload over Europe has changed, and the qd-tree root node has the cut `REGION = 'EUROPE'`, MTO would only reorganize the left subtree. Next, we describe a reward function for determining the value (i.e., benefit minus cost) of reorganizing a qd-tree subtree, and then we describe how MTO uses this reward function to determine the best reorganization strategy.

5.1.1 Minimizing Impact of Reorganization. To minimize impact on query performance, MTO spins up a separate process that performs (partial) reorganization using a (partial) copy of the data. During reorganization, queries are still executed on the existing data/layout, so query serving is unaffected. After reorganization completes, the new data/layout is swapped with the existing layout with minimal impact on the workload.

5.1.2 Reward Function. Assume that workload shift has occurred and we have already observed some queries, denoted Q, from this new workload (e.g., a sample of recently-run queries); assume we expect to run q more queries from the same distribution as Q before the next workload shift. The reward of reorganizing a subtree T (i.e.,

[2] We also experimented with ways to sample at different rates for different tables while maintaining an overall sample rate of s (e.g., sample more from smaller tables, sample less from larger tables). However, we found through evaluation that more complex schemes did not meaningfully impact the optimized layout's performance.

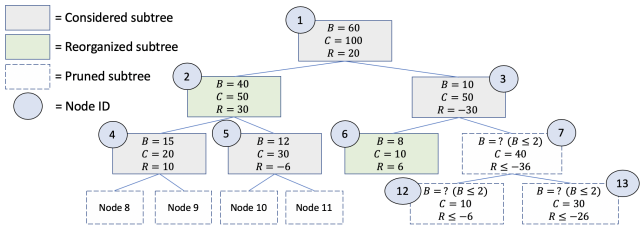

Figure 8: Using $q/w = 2$ in the reward, MTO chooses to reorganize the subtrees of nodes 2 and 6, achieving total reward 36.

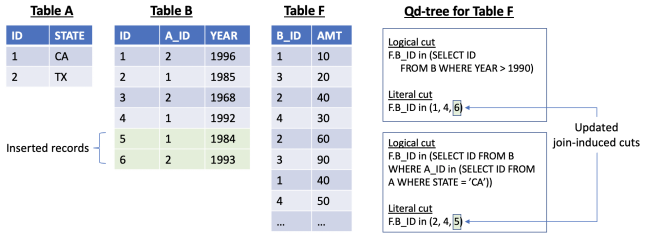

Figure 9: Inserting two records into table B causes updates to join-induced cuts in table F's qd-tree.

replacing T with a new qd-tree T' over the records in T's blocks) is defined as $R(T,Q) = (q/w) \cdot B(T,Q) - C(T)$, where:

- $B(T,Q)$ is the average number of block accesses that can be reduced for a query in Q if we fully reorganize the subtree T. It represents the *benefit* of reorganizing T in terms of expected impact on each of the next q queries. To compute $B(T,Q)$, we take all records in T's blocks and re-run offline optimization (Section 3.2.1) to construct a new qd-tree T', then take the difference in block accesses over Q between T and T'.
- $C(T)$ is the total number of blocks in T. It represents the *cost* of fully reorganizing T's blocks. Note that $C(T) \geq B(T,Q)$.
- w represents the *relative* overhead of reorganizing (i.e., re-compressing and re-writing blocks) vs. accessing blocks in the underlying storage system. For example, in our evaluation system (Section 6.1.2), compressing and writing a block is ~100× slower on average than reading a block, so $w = 100$.

A negative reward implies that it is not worth fully reorganizing T (however, a subtree of T may still have positive reward). Computing reward does not require us to actually perform any physical reorganization. Reorganizing T's blocks does not impact any other blocks in the layout. A higher q (meaning we expect the next workload shift will occur later in the future) encourages MTO to reorganize a larger portion of the dataset. $q \leq w$ leads to no reorganization, because reward can never be positive. Currently, a user must manually set q; we leave automatic setting of q based on predictions of future workload changes as future work.

5.1.3 Finding the Optimal Reorganization Strategy. For each table in the dataset, we compute $R(T,Q)$ for each subtree T of the table's qd-tree. We want to find the set of non-overlapping subtrees that has the maximum combined reward; this *optimal set* can be empty, in which case overall reward is 0. We find the qd-tree's optimal set via dynamic programming: we visit all nodes, starting from the leaves and working towards the root. At each node, we determine the optimal set over its subtree: the optimal set for a leaf L is $\{L\}$ if $R(L,Q) > 0$ and empty otherwise. The optimal set for a non-leaf T is either $\{T\}$ or the union of the optimal sets of its two children, whichever one has higher combined reward. The root node's optimal set is the qd-tree's optimal set. Fig. 8 shows a qd-tree in which the optimal set has two subtrees.

MTO runs this re-optimization workflow periodically according to some user-defined interval, such as every n hours or every n queries. If the overall reward is positive, MTO physically performs the reorganization by replacing each subtree T in the optimal set with its re-optimized subtree T' and re-writing T's blocks accordingly. If reward is non-positive, implying minimal workload shift during the interval, MTO will not reorganize.

Computing the reward for every subtree can be expensive for large qd-trees. The main bottleneck is computing $B(T,Q)$ for every T by re-running optimization on T's records to obtain a new qd-tree T'. The following properties help MTO prune nodes (i.e., avoid computing B on that node's subtree) that provably cannot be part of the optimal set:

(1) $B(T,Q)$ is upper bounded by the number of block accesses for the average query in Q using T's layout. This is because a new qd-tree T' cannot reduce the block accesses to less than zero.
(2) $B(T,Q) \geq B(T_L,Q) + B(T_R,Q)$, where T_L and T_R are the left and right subtrees of T. This is because any reorganizations of T_L and T_R independently can also be achieved by reorganizing T.
(3) If $R(T,Q) \geq B(T_L,Q) + B(T_R,Q)$, then no set of T's subtrees can have combined reward larger than $R(T,Q)$. This follows from property 2 and the fact that $C(T) \geq 0$.

To take advantage of these properties, we first use property 1 on every subtree to prune out any subtrees whose maximum possible reward is non-positive. For each non-pruned subtree T, we cache the upper bound for $B(T,Q)$. We then compute the reward for subtrees starting from the root node and continuing in breadth-first order (e.g., in node ID order in Fig. 8). When it comes time to compute the reward for T, we first check the cached upper bound for $B(T,Q)$. If the bound is low enough that $R(T,Q)$ cannot be positive, then we prune T.

Otherwise, we compute the true value of $B(T,Q)$ and update the cache to help prune later subtrees: (1) Benefits for T's subtrees are upper bounded by $B(T,Q)$. Let T's sibling and parent be S and P. Benefits for S's subtrees are upper bounded by $B(P,Q) - B(T,Q)$. This is possible through property 2. In Fig. 8, this helps us prune nodes 7, 12, and 13. (2) Once we compute the reward for T and its two children, we use property 3 to possibly prune out all further subtrees of T. In Fig. 8, this helps us prune nodes 8-11.

5.2 Dynamic Data

Inserted, deleted, or updated records are routed to the relevant data blocks using MTO's qd-trees. The physical change itself is handled transparently by the data analytics service. For example, many services buffer data changes in delta stores, then periodically merge delta stores into the main data store, which often requires re-writing blocks. This merging overhead must be paid for any data layout strategy that maintains some sort order over records, including simple strategies such as sorting by a user-selected column.

Data changes pose one unique challenge for MTO: join-induced cuts must be updated to reflect the new data. After an insert into a table, MTO must update all join-induced cuts in *other tables'* qd-trees that have the changed table on its induction path. In Fig. 9, inserting two records into table B results in updates to two join-induced cuts in table F's qd-tree. Any join-induced cuts in table B's qd-tree are unaffected. We perform the update by evaluating the relevant cut only on the inserted records, not all the records of table B. Similarly, a delete results in updates to join-induced cuts in other tables' qd-trees, performed by evaluating cuts only on the deleted records. A data update is handled as a delete followed by an insert.

A subtle but important ramification of updating join-induced cuts is that it shifts the "boundaries" between blocks. Will this force existing records to change blocks? Assuming referential integrity [54], and due to induced predicates only originating from join columns with unique values, like primary key columns (Section 4.1.1), inserts and deletes in MTO will never cause unchanged records to change blocks, because there cannot be records in the "boundary shift" region. For example, the inserted records in Fig. 9 do not join with any existing records in table F, so table F's join-induced cuts will select the same set of records before and after updating. However, data updates might cause updates to join-induced cuts that force existing records to change blocks.

6 EVALUATION

We present the results of an in-depth experimental study that compares MTO with other data layout strategies on a variety of multi-table datasets and workloads. Overall, this evaluation shows that:

- On a commercial cloud-based analytics service, MTO achieves up to 93% reduction in blocks accessed and up to 75% reduction in overall query time compared to alternative methods (Section 6.2). Queries with selective filters over joined tables benefit most from MTO (Section 6.3).
- MTO achieves low optimization times through sampling, resulting in faster end-to-end performance compared to alternatives (Section 6.4).
- MTO adapts its layout in response to workload shift and data changes (Section 6.5) and scales to larger query workload sizes and data sizes (Section 6.6).

6.1 Setup

6.1.1 Datasets and Workloads. We evaluate on three datasets: Star Schema Benchmark (SSB) [37], TPC-H [50], and TPC-DS [49], each by default with scale factor 100. This corresponds to around 60GB of data for SSB and 100GB of data for TPC-H and TPC-DS. For SSB, we use all 13 queries in the workload. For TPC-H, we support all 22 templates, and by default we use 8 randomly generated queries per template, resulting in a workload of 176 queries. For TPC-DS, we use 46 templates that vary in complexity[3], with one query per template.

6.1.2 Implementation and Systems. We implement MTO's offline optimization and simulation of blocks accessed during query execution in Python. We evaluate offline optimization and simulated performance on an Arch Linux machine with Intel Xeon Gold 6230 2.1GHz CPU and 256GB RAM. We also test the impact on query execution times on a commercial cloud-based analytics service, which we refer to as Cloud DW, which performs block skipping via per-block zone maps and semi-join reduction during query execution. Cloud DW aims to store 1M records in each of its data blocks, but blocks can have less than that target size (as low as around 100K records) due to various internal factors, including the efficiency of compression. Therefore, the block size in Cloud DW is not uniform. In simulation, we use a block size of 500K records.

We performed a shallow integration of MTO into Cloud DW: each block across the multi-table layout is assigned a unique block ID (BID). For each table, we materialize a new column that contains the BID for each record. In storage, we sort each table by its BID column. The per-block zone maps will now contain the min/max BIDs for records in the block. Before feeding each query into Cloud DW, MTO will rewrite the query by adding extra predicates which are used transparently by Cloud DW's zone maps to skip unnecessary blocks. For example, if routing a query through Table A's qd-tree tells us that processing the query only requires records from blocks 2 and 4 of Table A, we add the predicate `A.BID IN (2, 4)` to the query.

6.1.3 Comparisons. We compare MTO against two alternatives: (1) Baseline, which sorts each table by a user-tuned column[4]. (2) STO, which is an instance-optimized layout approach that follows MTO's algorithms without using join-induced predicates. That is, STO constructs a qd-tree per table, using only simple predicates. Note that for all methods, we create one layout for all queries in the workload.

In simulation, we also use data-induced predicates [21] (diPs, described in Section 3.1.1) to enhance the performance of STO and Baseline, using range-sets of size 20. Since diPs are meant to be incorporated into the query optimizer, we were not able to show the performance of diPs in Cloud DW as part of our shallow integration.

6.1.4 Metrics. We evaluate on three metrics: (1) number of blocks accessed in simulation, where each block is exactly 500K records. (2) Fraction of blocks accessed on Cloud DW. Because blocks are not equally sized on Cloud DW, it is unfair to compare the raw number of blocks accessed. Therefore, we use the fraction of blocks accessed out of the total number of blocks in the accessed base tables. (3) End-to-end query runtime on Cloud DW.

6.2 Overall Results

For each metric, we compare MTO to the alternatives using the overall metric across the entire query workload. We normalize to the metric achieved by Baseline.

6.2.1 Simulated Block Skipping. Fig. 10a shows that across datasets, MTO achieves between 43%–96% reduction in simulated block accesses compared to Baseline and between 32%–94% reduction in simulated block accesses compared to the best alternative method.

Data-induced predicates (diPs) help reduce blocks accessed by Baseline on SSB and TPC-DS. diPs do not provide any improvements on TPC-H because the diP is usually not selective enough to make an impact when pushed to other tables. Similarly, diPs provide only

[3]We use templates 1-50, except for 14, 23, 24, and 39, which are each composed of multiple queries.

[4]For SSB, we sort lineorders by orderdate and all other tables by primary key. For TPC-H, we sort lineitem by shipdate, orders by orderdate, and all other tables by primary key. For TPC-DS, we sort all fact tables by date (sold_date for sales tables and returned_date for returns tables) and all dimension tables by primary key.

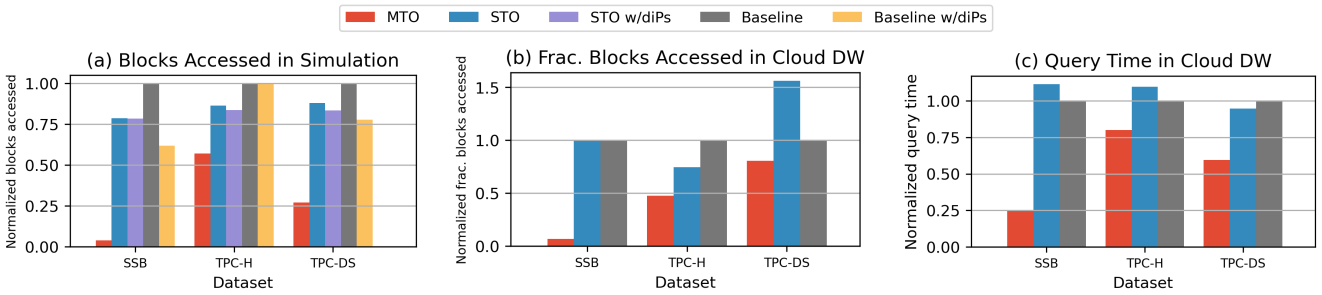

Figure 10: MTO achieves better overall workload performance than alternatives across datasets and metrics. Note that the y-axes are normalized to the metric achieved by Baseline.

	SSB	TPC-H	TPC-DS
Total cuts	272	4253	662
Total join-induced cuts	201	3397	517
Avg induction depth	1	1.44	1.07
Max induction depth	1	4	2
Memory size	2.05MB	3.67GB	4.58MB

Table 2: Statistics of MTO's qd-trees.

minor improvements for STO, because STO creates blocks based only on columns that are filtered by simple predicates, which are independent of join columns. Therefore, diPs created from STO's layout are not selective enough to make an impact.

Table 2 shows that for MTO, the total number of cuts over all qd-trees varies across datasets. This trend is primarily due to the size of the query workload that MTO optimizes on: (1) workloads with more queries produce more candidate cuts, and (2) larger workloads require a finer-grained blocking strategy in order to maximize block skipping across all queries. For all datasets, MTO's qd-trees are composed mostly of join-induced cuts. On SSB, the induction depth of join-induced cuts is always 1, because all dimension tables are joined directly to the fact table. On TPC-H, the maximum induction depth is 4 (e.g., a join-induced cut with region as the source table and joining through nation, customer, and orders to reach the lineitem table).

Table 2 also shows that the memory overhead of MTO, which comes from its per-table qd-trees, is at most a few GB, which is small compared to the size of the data (~100GB). MTO's size on TPC-H is much higher than on SSB and TPC-DS due to having more join-induced cuts with higher-cardinality literal cuts (e.g., many join-induced cuts on TPC-H originate from the orders table, which produces literal cuts with as many as 150M values).

6.2.2 Block Skipping on Cloud DW. Fig. 10b shows that MTO's simulated reduction in block accesses roughly translates to actual reduction of block accesses on Cloud DW. Across datasets, MTO achieves between 19%–93% reduction in fraction of blocks accessed compared to the best alternative method.

There are a couple reasons for the difference between simulated and actual block accesses: (1) Block sizes in simulation are fixed at 500K records, whereas actual block sizes in Cloud DW vary between a maximum of 1M records and a low of around 100K records. (2) The execution engine of Cloud DW may perform extra optimizations that we do not consider in simulation, such as semi-join reductions, which lead to additional block skipping. These extra optimizations may affect each method differently. In particular, the second reason explains why, for TPC-DS, MTO and STO have higher normalized block accesses on Cloud DW than in simulation: most queries in the workload filter on date, which allows Baseline to heavily take advantage of semi-join reductions, because Baseline sorts fact tables by date. Therefore, Baseline accesses significantly fewer blocks on Cloud DW than in simulation. MTO and STO can also take advantage of semi-join reductions, but to a lesser extent because their fact tables are not completely sorted on date.

6.2.3 End-to-end Runtimes on Cloud DW. Fig. 10c shows that across datasets, MTO achieves between 20%–75% reduction in end-to-end query runtimes compared to the best alternative method. The reduction in query time is generally not as dramatic as the reduction in block accesses because block access is only one part of the total time spent on query execution (e.g., time to compute joins is not reduced by block skipping). However, on TPC-DS, MTO and STO actually improve their normalized performance compared to Fig. 10b. This is because Baseline incurs heavier runtime costs of using semi-join reductions, as explained in Section 6.2.2.

6.3 Performance Breakdown by Query

Fig. 11 shows the fraction of queries that achieve a certain reduction in query time on Cloud DW compared to the alternative methods. Reduction in query times from MTO is achieved by all queries in SSB, around 50% of query templates in TPC-H, and around 75% of queries in TPC-DS. For a few queries, performance regresses when using MTO's layout. This is because MTO optimizes to achieve best overall block skipping across all queries in a workload. Therefore, MTO may allow performance to regress for certain queries in order for overall performance to improve.

6.3.1 When Does MTO Pay Off? Fig. 10 shows that the performance benefits of MTO varies depending on the dataset. To better understand the conditions under which MTO offers performance benefit over STO and Baseline, we select five query templates with different filter/join characteristics from the TPC-H workload, all of which touch the fact table (lineitem): Q1 has no joins and scans most of the fact table, Q14 has a filter over the fact table on the sort column (`L_SHIPDATE`) but no filter over joined dimension tables, Q6 has filters over the fact table on non-sort columns but no joins, Q4 has selective filters over joined dimension tables only on columns that

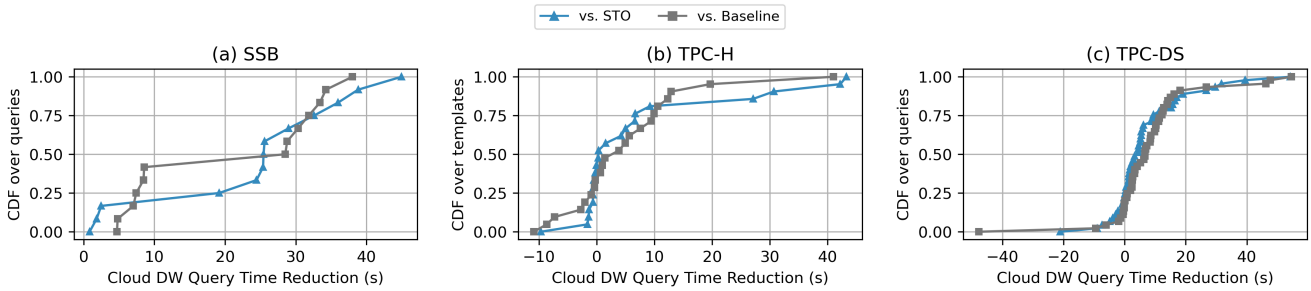

Figure 11: Reduction in query runtimes achieved by MTO, relative to STO and Baseline. Different queries achieve different performance gains.

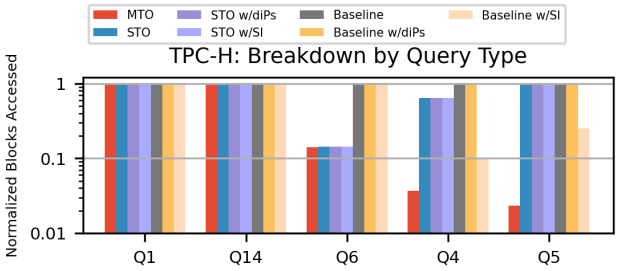

Figure 12: MTO has the most performance advantage over STO and Baseline on queries with selective filters over joined tables, like Q4 and Q5.

	SSB	TPC-H	TPC-DS
MTO			
Optimization time (min)	0.195	3.67	0.619
Data sample rate used for opt.	0.01	0.03	0.01
Routing time (min)	1.54	5.80	3.50
Total offline time (min)	1.73	9.47	4.12
STO			
Optimization time (min)	0.0213	0.697	0.0611
Data sample rate used for opt.	0.003	0.0003	0.01
Routing time (min)	0.360	0.978	0.771
Total offline time (min)	0.381	1.68	0.832

Table 3: Offline optimization times for Fig. 10.

	SSB	TPC-H	TPC-DS
X=total queries run, **Y**=STO	4	33	29
X=minutes from start, **Y**=STO	2.18	26.9	9.72
X=total queries run, **Y**=Baseline	6	56	32
X=minutes from start, **Y**=Baseline	2.41	39.1	10.3

Table 4: How many X until MTO runs more queries than Y?

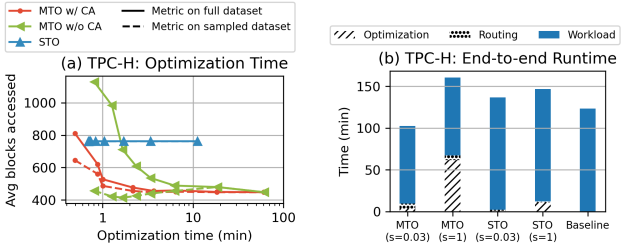

Figure 13: (a) MTO and STO can decrease optimization time by using sampling. Cardinality adjustment (CA) helps MTO mitigate performance degradation. (b) MTO achieves the lowest end-to-end runtime when optimized with a 3% data sample.

are correlated to the fact table's sort column, and Q5 has selective filters over joined dimension tables on columns that are not correlated to the fact table's sort column. We create five different layouts using MTO and STO, specialized for each query template individually.

Fig. 12 shows simulated block skipping. For STO and Baseline, we also evaluate using data-induced predicates (diPs), as well as creating a secondary index (SI) on the fact table's join column (L_ORDERKEY) so that at runtime, we push a join-induced predicate from the dimension table to the fact table and use the secondary index to prune blocks.

Fig. 12 provides several insights: (1) On queries that have non-selective filters (like Q1) or selective filters only over the sort column (like Q14), MTO and STO have little or no advantage over Baseline because Baseline already prunes most irrelevant blocks. (2) On queries that have selective filters over non-sort columns and no joins or only non-selective filters over joined tables (like Q6), MTO and STO perform equally well, because MTO cannot take advantage of join-induced predicates, but they both outperform Baseline. (3) On queries with selective filters over joined tables that are correlated with the fact table's sort column (like Q4), MTO performs better than STO and Baseline, but using a secondary index allows Baseline to take advantage of correlations to filter out some blocks in the fact table at runtime. (4) On queries with selective filters over joined tables that are not correlated with the fact table's sort column (like Q5), MTO outperforms all alternatives by a large margin.

Therefore, in workloads with a significant portion of queries that satisfy the third and fourth conditions above, MTO will show the largest gains. This is especially true for the SSB workload, in which most queries include a selective filter over a joined dimension table, and no one sort column on the fact table is correlated to filtered columns in all dimension tables. In contrast, the TPC-H workload contains many queries that satisfy the first and second conditions, and therefore MTO's performance gains are not as significant.

6.4 End-to-end Performance

We examine the impact of offline steps (layout optimization and assigning records to blocks) on end-to-end performance.

6.4.1 Optimization Time. Table 3 shows the time that MTO and STO take to find the optimal layout for each dataset, when optimized using the data sample rates shown in the table. These sample rates were chosen so that the simulated performance of the layout optimized on the sample has less than 1% difference with the layout optimized on the full data. Smaller query workloads and simpler join patterns generate fewer candidate cuts, which leads to lower optimization times. Therefore, MTO optimization finishes quickly for SSB, which has 13 queries and a maximum induction depth of 1, while optimization takes longer on TPC-H, whose workload has 176 queries and a maximum induction depth of 4 (Table 2). Similarly, data routing (i.e., assigning each record to a block) is slowest on TPC-H because the qd-trees optimized on TPC-H are larger and the paths from root to leaf are deeper (Table 2). Optimization and routing times are lower for STO than MTO because STO does not need to consider join-induced cuts during optimization and does not include join-induced cuts in its qd-trees, which makes both steps computationally simpler than for MTO.

Fig. 13a shows the impact of varying the sample rate between 1 (i.e., no sampling) and 0.0003 on optimization time for TPC-H. The solid lines show blocks accessed in simulation when evaluated on the full dataset, whereas the dotted lines show blocks accessed when evaluated on the sample. With cardinality adjustment (CA) (Section 4.2), the metric computed on the sample is close to the true metric on the full dataset, whereas without CA, the sampled metric is inaccurate. By using CA, MTO can reduce its optimization time from nearly an hour without sampling to under 4 minutes with a 3% sample, while achieving nearly the same layout quality. STO's layout quality is negligibly impacted by sampling because it does not consider join-induced cuts, which are most affected by sampling.

6.4.2 End-to-end Time. Fig. 13b shows the end-to-end time for the TPC-H workload with 176 queries, including the offline optimization and routing times. By optimizing on a 3% sample of the data, optimization time is a small fraction of overall runtime for both MTO and STO, and therefore the faster query times achieved by MTO allow it to complete the end-to-end workload quickest. Query routing latency (i.e., determining which blocks a query must read) is on the order of milliseconds per query, which is negligible compared to total query time, which is on the order of seconds per query (Fig. 11).

Since MTO pays the upfront time cost to perform offline optimization and data routing, how long does it take for MTO to catch up to STO and Baseline? Table 4 shows the total number of queries MTO runs before surpassing STO and Baseline, as well as the time it takes for MTO to complete that many queries (including time for offline steps). For example, on SSB, MTO runs more queries than STO after ~2 minutes. In all cases, MTO reaches this crossover point before the workload completes.

6.5 Dynamic Workloads and Data

6.5.1 Dynamic Workloads. To show how MTO adapts to workload shift, we use MTO to optimize the layout based on templates 1-11 of the TPC-H workload using 8 queries per template, then actually run queries drawn from templates 12-22 of TPC-H. This simulates a scenario in which the user completely and suddenly changes their query workload; in reality, workload shift is likely not so abrupt.

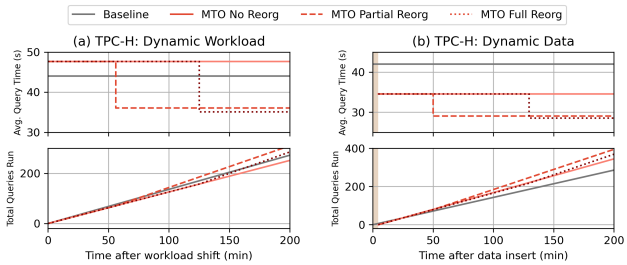

Figure 14: (a) MTO initially performs worse than Baseline after workload shift but performs better in the long run after reorganization. (b) MTO maintains its advantage over Baseline after data insertion.

	Frac. Data Reorganized	Re-opt. Time (min)	Frac. Subtrees Considered in Re-opt.
$q = 100$	0	0	0
$q = 200$	0.370	9.81	0.031
$q = 500$	0.841	25.0	0.163
$q = 1000$	0.893	17.3	0.084
$q = \infty$	1.0	2.48	N/A

Table 5: MTO behavior after workload shift.

Fig. 14a shows that immediately after the workload shift, queries on MTO have higher execution times than queries on Baseline, because MTO's layout is not optimized for the observed workload (i.e., templates 12-22). Re-optimizing and physically reorganizing the entire layout based on the observed workload (MTO Full Reorg) takes more than two hours; during reorganization, queries are still executed on the old layout (Section 5.1.1). However, MTO is able to use partial re-optimization (Section 5.1, using $q = 200$ and $w = 100$) to physically reorganize only a subset of existing blocks. Partial re-optimization and reorganization completes in under an hour (MTO Partial Reorg), while achieving nearly the same resulting performance benefit as a full reorganization, because it only reorganizes the blocks that have the most impact on performance. The bottom half of Fig. 14a shows the impact of reorganization on the total number of queries executed over time. Even though MTO initially executes fewer queries than Baseline after the workload shift, it is able to quickly recoup the lost time after reorganization.

Table 5 shows that as we increase q in the reward function while fixing $w = 100$ (Section 5.1.2), MTO will choose to reorganize a larger fraction of the data. Therefore, a user can adjust q to trade off between decreased execution time of future queries and computation costs of reorganization. The time for physically performing reorganization (i.e., writing/compressing blocks) is roughly proportional to the fraction of data reorganized, and reorganizing all TPC-H data takes around 2 hours in our setup. Furthermore, the time to perform re-optimization is kept relatively low (compared to time for performing reorganization) because we use the properties from Section 5.1.3 to avoid unnecessary computation, and therefore only consider a small fraction of all subtrees during the re-optimization process (Table 5).

6.5.2 Dynamic Data. To show how MTO adapts to dynamic data, we use the TPC-H dataset. We first remove all records from the orders table with orderdate after Jan 1, 1996, and all records in the lineitem

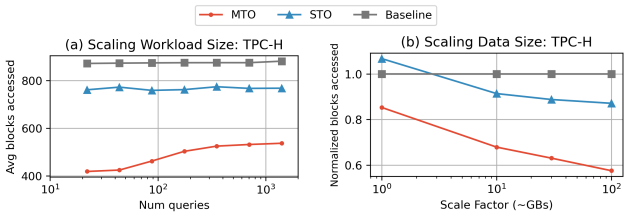

Figure 15: MTO scales to larger query workload sizes and improves its relative performance at larger data sizes.

table that join with the removed order records. This leaves around 61% of the records in both the orders and lineitem tables. We perform offline optimization using this partial dataset, then insert the records we had removed. This represents the common use case in which new records are inserted into the fact tables (in this case, new records from 3 years of orders).

Fig. 14b shows that MTO takes around 4 minutes to update join-induced cuts (represented by the shaded region). MTO is unable to assign inserted lineitem records to blocks while cuts are updating, so queries accessing lineitem executed in the first 4 minutes must read all inserted records, which is slow. After cuts have updated and inserted records are assigned to blocks, MTO without reorganization already achieves lower average query time than Baseline. This implies that MTO performance is impacted less by data changes than by workload shift, because the structure/cuts of the existing qd-trees still conform to observed query patterns; *this is important for practical reasons* because data changes happen frequently but we do not expect workload shift to occur as often. MTO can optionally perform reorganization to boost performance further (Fig. 14b). In all cases, MTO is able to quickly recoup the time spent updating join-induced cuts and outpace Baseline in number of executed queries.

6.6 Scalability

6.6.1 Workload Size. On TPC-H, we vary the number of queries per template in the workload from 1 to 64. Since there are 22 templates, this results in workloads ranging from 22 to 1408 queries. Fig. 15a shows that as workload size increases, average blocks accessed by MTO increases slightly, because the fixed number of data blocks does not provide MTO enough degrees of freedom to optimize for all queries in a larger workload with more unique predicates. Nevertheless, MTO maintains a relative performance advantage over alternative methods, with 30% and 39% fewer blocks accessed than STO and Baseline, respectively, for a workload with 1408 queries.

6.6.2 Data Size. On TPC-H, we vary the scale factor from 1 to 100, while maintaining the workload of 176 queries and a block size of 500K records. Fig. 15b shows that MTO (and STO) achieves greater reduction in block accesses over Baseline as data size increases; larger data size leads to more total data blocks, which allows MTO (and STO) to take advantage of finer-grained blocking strategies.

7 RELATED WORK

Physical Data Layouts & Partitioning. Cloud-based analytics services typically distribute data across multiple nodes or partitions, in order to scale out computation and load balance among computational resources. Data is often distributed either based on ingestion time, or using range, hash, or round-robin distribution schemes [26]. Automatic design advisors use what-if analyses and data mining to auto-tune the physical design and partitioning scheme [1, 36, 44]. Certain automated approaches are specialized for transactional workloads [7, 42, 43] or analytic workloads [13, 28]. MTO may be applied within each node or partition created by these schemes.

Qd-tree [56] (Section 2.1) and Sun et al. [47, 48] propose physical data layouts that maximize block skipping. Amoeba [46] adapts its partitioning to ad-hoc workloads. These approaches optimize the layout for a single table, whereas MTO jointly optimizes the layout for multiple tables.

Instance-Optimized Databases. There has been a recent research trend towards instance-optimized database systems and components. Whereas design decisions in traditional systems are often made though manual tuning or heuristics, the goal of instance-optimized systems is to automatically specialize database components and algorithms to a particular use case, sometimes using machine learning. MTO and qd-tree [56] are frameworks for instance-optimized data layouts. [2] introduces layouts for hybrid read-write workloads tailored to the data and query workload. Recent works have proposed instance-optimized, or learned, approaches for partition advising [17], tuning [51], data structures and indexes [10, 11, 15, 19, 24, 27, 35, 55], query optimization [25, 31, 32, 40], cardinality estimation [12, 22, 57, 58], job scheduling [30], workload forecasting [29], and complete database systems [23].

Sideways Information Passing. Similar to MTO's join-induced predicates, data-induced predicates [21] (Section 3.1.1), column equivalence [14], and magic-set rewriting [45] can also be used to push predicate information through joins. The performance benefit of these techniques depends on the data layout (e.g., pushed predicates cannot help skip blocks if every block contains records satisfying the predicate). MTO uses join-induced predicates to explicitly construct a layout that maximizes opportunities for block skipping during execution.

During query execution, sideways information passing between two joined tables or subexpressions, often in the form of semi-join reduction [3], can be used to skip blocks and speed up joins [4, 20, 41]. In contrast, MTO performs sideways information during offline optimization in order to produce a better joint layout for multiple tables.

Some auxiliary data structures cache useful information about joining tables. These include materialized views, join indexes [38], and join zone maps [39]. These data structures use extra storage space and incur maintenance overhead. In contrast, MTO does not duplicate any of the base data.

8 CONCLUSION

One of the dominant costs for query processing in cloud-based data analytics services is the I/O for accessing large data blocks from cloud storage. Per-block zone maps are a commonly-employed technique for reducing I/O by skipping blocks, but their effectiveness is dependent on how the records are assigned to blocks, i.e., the data layout. Existing approaches for optimizing data layouts only target a single table, and their performance suffers in the presence of join-based queries. In this paper, we propose MTO, a data layout framework that

automatically and jointly optimizes the blocking strategy for all tables in a multi-table dataset for a given query workload. We show that by taking advantage of sideways information passing through joins during the optimization process, MTO produces layouts that achieve up to 93% reduction in blocks accessed and 75% reduction in end-to-end query times on a commercial cloud-based data analytics service.

Acknowledgements. This research is supported by the MIT Data Systems and AI Lab (DSAIL) and by NSF IIS 1900933.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database Tuning Advisor for Microsoft SQL Server 2005: Demo. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 930âĂŞ932. https://doi.org/10.1145/1066157.1066292

[2] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *Proc. VLDB Endow.* 12, 13 (Sept. 2019), 2393âĂŞ2407. https://doi.org/10.14778/3358701.3358707

[3] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (Jan. 1981), 25âĂŞ40. https://doi.org/10.1145/322234.322238

[4] Vivek Bharathan, Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Chuck Bear, and Ariel Cary. 2013. Materialization Strategies in the Vertica Analytic Database: Lessons Learned. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 1196âĂŞ1207. https://doi.org/10.1109/ICDE.2013.6544909

[5] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* 46, 5 (2016), 709–719. https://doi.org/10.1002/spe.2325 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2325

[6] Zach Christopherson. 2016. Amazon Redshift EngineeringâĂŹs Advanced Table Design Playbook: Compound and Interleaved Sort Keys. https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/

[7] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. 3, 1âĂŞ2 (Sept. 2010), 48âĂŞ57. https://doi.org/10.14778/1920841.1920853

[8] Databricks. 2020. Data skipping index. https://docs.databricks.com/spark/latest/spark-sql/dataskipping-index.html

[9] Databricks Delta Engine. 2020. Z-Ordering (multi-dimensional clustering). https://docs.databricks.com/delta/optimizations/file-mgmt.html#z-ordering-multi-dimensional-clustering

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969âĂŞ984. https://doi.org/10.1145/3318464.3389711

[11] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. arXiv:cs.DB/2006.13282

[12] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates Using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (May 2019), 1044âĂŞ1057. https://doi.org/10.14778/3329772.3329780

[13] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. 2008. Supporting Table Partitioning by Reference in Oracle. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1111âĂŞ1122. https://doi.org/10.1145/1376616.1376727

[14] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind M. Joshi. 2007. Execution Strategies for SQL Subqueries. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 993âĂŞ1004. https://doi.org/10.1145/1247480.1247598

[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, 1189âĂŞ1206. https://doi.org/10.1145/3299869.3319860

[16] Cesar A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. 2008. Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, USA, 1190âĂŞ1199. https://doi.org/10.1109/ICDE.2008.4497528

[17] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 143âĂŞ157. https://doi.org/10.1145/3318464.3389704

[18] Dawei Huang, Dong Young Yoon, Seth Pettie, and Barzan Mozafari. 2019. Joins on Samples: A Theoretical Guide for Practitioners. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 547âĂŞ560. https://doi.org/10.14778/3372716.3372726

[19] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 535âĂŞ550. https://doi.org/10.1145/3183713.3199671

[20] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, USA, 774âĂŞ783. https://doi.org/10.1109/ICDE.2008.4497486

[21] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 252âĂŞ265. https://doi.org/10.14778/3368289.3368292

[22] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv preprint arXiv:1809.00677* (2018).

[23] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[24] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD âĂŹ18)*. Association for Computing Machinery, New York, NY, USA, 489âĂŞ504. https://doi.org/10.1145/3183713.3196909

[25] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[26] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. 2013. Enhancements to SQL Server Column Stores *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1159âĂŞ1168. https://doi.org/10.1145/2463676.2463708

[27] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119âĂŞ2133. https://doi.org/10.1145/3318464.3389703

[28] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 589âĂŞ600. https://doi.org/10.14778/3055540.3055551

[29] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD âĂŹ18)*. Association for Computing Machinery, New York, NY, USA, 631âĂŞ645. https://doi.org/10.1145/3183713.3196908

[30] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270âĂŞ288. https://doi.org/10.1145/3341302.3342080

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705âĂŞ1718. https://doi.org/10.14778/3342263.3342644

[32] Ryan Marcus and Olga Papaemmanouil. 2019. Towards a Hands-Free Query Optimizer through Deep Learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p96-marcus-cidr19.pdf

[33] Microsoft. 2019. Columnstore indexes - Query performance. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-query-performance

[34] G. M. Morton. 1966. *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing (PDF)*. Technical Report. IBM.

[35] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD âĂŹ20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3318464.3380579

[36] Rimma Nehme and Nicolas Bruno. 2011. Automated Partitioning Design in Parallel Database Systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1137âĂŞ1148. https://doi.org/10.1145/1989323.1989444

[37] Pat O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2020. Star Schema Benchmark. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[38] Oracle. 2020. Bitmap Join Indexes. https://docs.oracle.com/cd/B10500_01/server.920/a96520/indexes.htm

[39] Oracle. 2020. Database Data Warehousing Guide: Using Zone Maps. https://docs.oracle.com/database/121/DWHSG/zone_maps.htm

[40] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. ACM, 4.

[41] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-up Approach. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 663âĂŞ676. https://doi.org/10.14778/3184470.3184471

[42] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61âĂŞ72. https://doi.org/10.1145/2213836.2213844

[43] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 430âĂŞ441. https://doi.org/10.1145/2452376.2452427

[44] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating Physical Database Design in a Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 558âĂŞ569. https://doi.org/10.1145/564691.564757

[45] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 435âĂŞ446. https://doi.org/10.1145/233269.233360

[46] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. 2017. A Robust Partitioning Scheme for Ad-Hoc Query Workloads. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 229âĂŞ241. https://doi.org/10.1145/3127479.3131613

[47] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-Grained Partitioning for Aggressive Data Skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1115âĂŞ1126. https://doi.org/10.1145/2588555.2610515

[48] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-Oriented Partitioning for Columnar Layouts. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 421âĂŞ432. https://doi.org/10.14778/3025111.3025123

[49] TPC. 2020. TPC-DS. http://www.tpc.org/tpcds/.

[50] TPC. 2020. TPC-H. http://www.tpc.org/tpch/.

[51] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024. https://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf

[52] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 993âĂŞ1008. https://doi.org/10.1145/3035918.3064007

[53] Wikipedia. 2020. Correlated Subquery. https://en.wikipedia.org/wiki/Correlated_subquery

[54] Wikipedia. 2021. Referential Integrity. https://en.wikipedia.org/wiki/Referential_integrity

[55] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, 1223âĂŞ1240. https://doi.org/10.1145/3299869.3319861

[56] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 193âĂŞ208. https://doi.org/10.1145/3318464.3389770

[57] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. arXiv:cs.DB/2006.08109

[58] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 279âĂŞ292. https://doi.org/10.14778/3368289.3368294

[59] Zack Slayton. 2017. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB. https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/.