Pando: Enhanced Data Skipping with Logical Data Partitioning

Sivaprasad Sudhir
MIT, Meta
siva@csail.mit.edu

Wenbo Tao
Meta
wenbo0@meta.com

Nikolay Laptev
Meta
nlaptev@meta.com

Cyrille Habis
Meta
cyrilleh@meta.com

Michael Cafarella
MIT
michjc@csail.mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

ABSTRACT

With enormous volumes of data, quickly retrieving data that is relevant to a query is essential for achieving high performance. Modern cloud-based database systems often partition the data into blocks and employ various techniques to skip irrelevant blocks during query execution. Several algorithms, often based on historical properties of a workload of queries run over the data, have been proposed to tune the physical layout of data to reduce the number of blocks accessed. The effectiveness of these methods at skipping blocks depends on what metadata is stored and how well the physical data layout aligns with the queries. Existing work on automatic physical database design misses significant opportunities in skipping blocks because it ignores logical predicates in the workload that exhibit strongly correlated results. In this paper, we present Pando which enables significantly better block skipping than past methods by informing physical layout decisions with correlation-aware logical partitioning. Across a range of benchmark and real-world workloads, Pando attains up to 2.8X reduction in the number of blocks scanned and up to 2.3X speedup in end-to-end query execution time over the state-of-the-art techniques.

PVLDB Reference Format:
Sivaprasad Sudhir, Wenbo Tao, Nikolay Laptev, Cyrille Habis, Michael Cafarella, and Samuel Madden. Pando: Enhanced Data Skipping with Logical Data Partitioning. PVLDB, 16(9): 2316 - 2329, 2023.
doi:10.14778/3598581.3598601

1 INTRODUCTION

With increasing volumes of data, reducing the amount of data that is accessed and processed is key to achieving high performance and reducing resource consumption in data-intensive systems. Modern data warehouses use a variety of techniques like horizontal and vertical partitioning, clustered indexes, sort orders, etc. to avoid scanning unnecessary data during query processing [1, 9, 12, 36].

Typically, cloud-based database systems organize data as compressed columnar blocks in storage, each block containing thousands or millions of tuples [26]. A block is the smallest unit of I/O, i.e., a block is either skipped or scanned entirely at query time. I/O required for accessing data blocks from storage is one of the dominant costs for query processing in these systems. To minimize the amount I/O performed, DBMSs maintain metadata such as block indexes, per-block statistics, etc. along with the data to skip accessing irrelevant blocks during query execution.

The effectiveness of data skipping depends on what metadata is stored, how the data is laid out, and how well it aligns with the query patterns. Range partitioning that is used by most production systems is useful for skipping blocks for queries that filter on the partitioned columns, but does not collect enough metadata to provide any benefits for queries that filter on other columns. Commercial systems maintain per-block metadata such as min-max values of each column to skip blocks when the range of values in it does not intersect the query filter [11, 14, 30, 34]. But their effectiveness depends on the tightness of each block's min-max range which depends on how tuples are assigned to blocks.

Recent studies such as Qd-trees [45] and MTO [15] have shown that more expressive partitioning schemes can achieve higher performance by leveraging data and workload-specific information. These methods tailor the data layout for a specific application by hierarchically partitioning the physical data space using filter expressions that appear in the query workload. However, even these methods fail to exploit a common workload pattern: *correlation among query predicates.* Modern applications generate queries that involve complex predicates that are often correlated with each other, i.e., the predicates are satisfied by a similar set of tuples [23, 27]. For example, queries with filters `DISTANCE(start_point, end_point) < 3 miles` and `cost < $7` on an NYC Taxi dataset will select similar tuples. Ideally, a storage system would exploit this property by creating blocks that can supply tuples for either predicate. However, existing systems create blocks that reflect each predicate independently; for example, a split in the hierarchical partitioning based on cost is not useful for queries with filters on `DISTANCE(start_point, end_point)` and thereby forces correlated queries to touch more blocks during query processing.

In this paper, we present Pando[1], a metadata-rich correlation-aware storage system for aggressively skipping data blocks. Pando is able to deliver substantially better block skipping than past methods by informing physical layout decisions with correlation-aware logical partitioning. Pando takes advantage of correlated predicates by creating multiple logical partitionings to minimize I/O for a variety of queries. In the NYC taxi dataset, we can create one logical partitioning that splits the data space based on `cost` attribute and one that splits based on `DISTANCE(start_point, end_point)`. Each logical partition maps to physical data blocks that intersect

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598601

[1]Pando is a huge grove of quaking Aspen that share a single root system and are considered one organism; In Latin, Pando means *I Spread.*

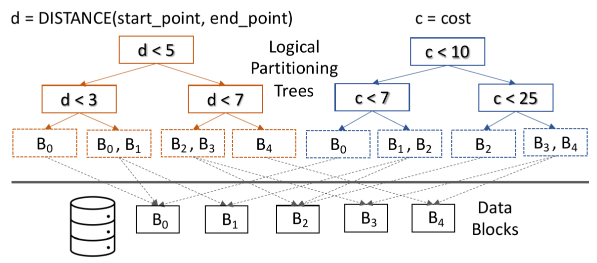


Figure 1: An example Pando layout for NYC Taxi dataset. B_0, B_1, etc. are physical data blocks. There are two logical partitioning trees, one with predicates on `DISTANCE(start_point, end_point)`, another on `cost`. Leaf nodes of the tree map to the physical blocks that intersect the logical data ranges.

it. To ensure good performance, each logical partition should map to only a few data blocks, i.e., the physical data layout has to align well with the logical partition boundaries. Pando optimizes the physical data layout to maximize the utility of these logical indexes while handling outlier tuples in separate outlier partitions. For example, the high-cost / low-distance and low-cost / high-distance rides might be stored together in an outlier partition. This layout is shown in Fig. 1.

Pando co-optimizes a set of correlation-aware logical partitionings and the physical layout of the data to maximize the overall performance of a given workload on a dataset, allowing it to specialize to data and workload distributions. Pando maintains a collection of logical partitionings of the data, each arranged as a tree that hierarchically splits the logical data space using predicate expressions that are useful for skipping blocks during query processing. The leaves of these trees point to the physical blocks that contain the logical ranges in the tree nodes. At execution time, these expressions are intersected with the query filters to skip blocks that do not intersect with the query. Pando chooses the underlying physical blocks in such a way to maximize the opportunity for block skipping, exploiting correlations between expressions to find good physical designs.

Decoupling the physical data layout and logical partitioning trees creates an extremely large search space of configurations that need to be considered during optimization. Data can be partitioned into blocks in a large number of ways (the so-called *Bell number*), which grows faster than exponential in the number of tuples. For any given blocking, an exponentially large number of index trees can be created, causing a combinatorial explosion in the space of physical design alternatives. We present a layout optimizer that efficiently navigates this space by leveraging properties of the data and workload.

To summarize, this paper makes the following contributions

(1) We present Pando, a metadata-rich correlation-aware data layout that jointly optimizes the selection of multiple logical partitioning trees and the physical data layout to enhance query performance by reducing the amount of data accessed.
(2) We provide an efficient heuristic algorithm to determine a good set of partitioning trees and the assignment of tuples to blocks for a given dataset and workload.
(3) Finally, we demonstrate that Pando outperforms other layouts across a range of real-world and benchmark datasets and workloads. Pando is up to 2.3X faster in overall query execution time and reduces the number of blocks scanned by up to 2.8X when compared to the state-of-the-art learned data layout, MTO.

2 MOTIVATION

We begin with an overview of the limitations of the existing solutions and then make the case for Pando. Qd-tree [45] is a recently proposed learned data layout framework for single tables, that specializes its data layout to a specific dataset and the workload of queries that run on it to minimize the amount of data scanned during query processing. To optimize the layout, it extracts "simple" predicates from the workload, like `X < 50`, `X < Y`, etc. and constructs a binary decision tree from them. Data is partitioned according to this tree. Each inner node in the tree is a predicate expression, called a *cut*, that cuts the physical data space into two child nodes. One of the child nodes corresponds to the tuples that satisfy the predicate expression and the other corresponds to the tuples that do not. The leaf nodes of the tree correspond to a data block. At execution time, a query traverses the tree to find the blocks that need to be scanned. At any given node, if the query intersects only one of the cut or its negation, only the corresponding child node is traversed. Otherwise, both children are traversed.

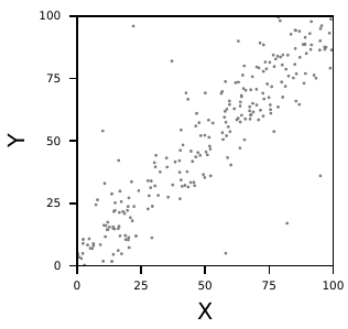
MTO [15] extends qd-trees to optimize the layout across multiple tables. MTO takes a set of tables as the input and produces one qd-tree layout per table. In addition to single table predicates in qd-trees, MTO uses join-induced predicates to cut the nodes in the partition tree. For example, if the workload contains queries like

```
SELECT * FROM T1 JOIN T2 on T1.Key = T2.Key
WHERE T1.X < 50 ANF T2.Y > 75
```

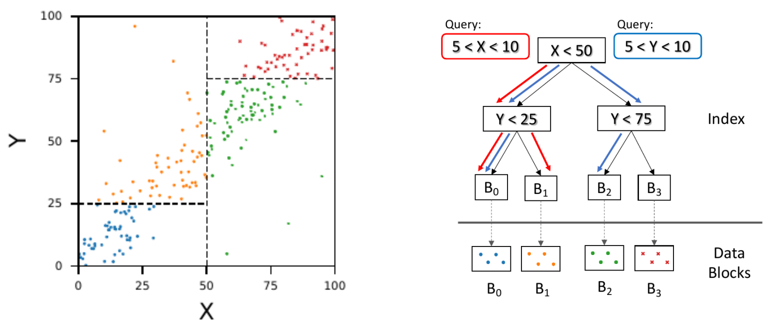
the predicate `T1.Key IN (SELECT T2.Key FROM T2 WHERE T2.Y > 75)` is considered for cutting T1's partitioning tree and `T2.Key IN (SELECT T1.Key FROM T1 WHERE T1.X < 50)` for cutting T2's data space. These cuts are useful for pre-filtering tuples that would otherwise be dropped during the join operation.

Both MTO and qd-tree construct a single tree per table which is both a partition function and an index for skipping blocks at query time. Most cloud systems have a large minimum block size (millions of tuples) to hide the latency of accessing cloud storage and to achieve high compression ratios [26]. As each leaf node of this tree corresponds to a block with a minimum size, the height of the tree is limited, so just having a single tree of expressions limits the number of expressions that can be present in the tree. This in turn reduces the utility of the tree for skipping data for a variety of queries. This can be particularly bad when there are correlated expressions, i.e., expressions over different fields that are satisfied by substantially overlapping sets of tuples.

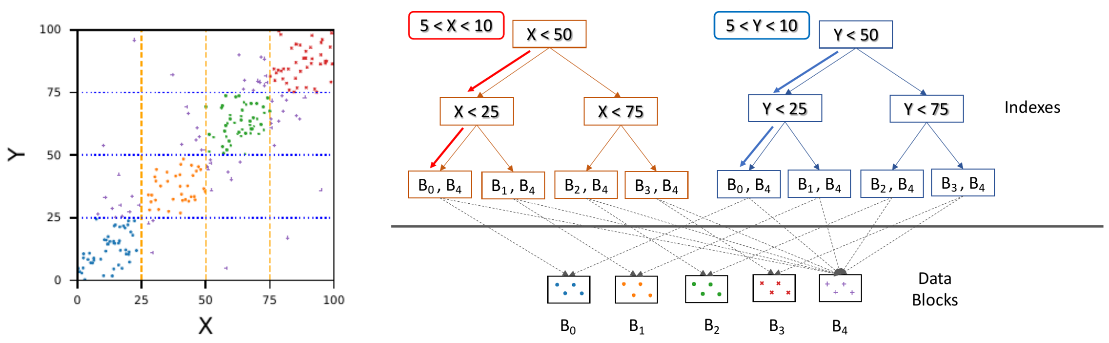
To illustrate where this overhead comes from, consider a toy example with a single two-dimensional table visualized in Fig. 2a. The X and Y axes correspond to two correlated columns in the table (e.g., cost and distance). The grey dots represent the tuples in the table. Consider a workload with 2 kinds of queries uniformly distributed in the data space. Half the queries have a range predicate on column X and the other half have a range predicate on Y



(a) Dataset



(b) Qd-tree / MTO



(c) Pando

Figure 2: Fig. 2a shows a 2-dimensional dataset with correlated columns X and Y. Consider a workload with 2 kinds of queries. Half the queries have a range predicate on column X and the other half have a range predicate on Y. Fig. 2b shows the layout that qd-tree and MTO will find. Nodes traversed during execution of queries are marked along the edges of the tree. Each query scans the blocks in the leaf nodes that intersect the query. Fig. 2c shows the Pando layout with two logical partitioning trees and data blocks that align well with the logical trees and query access patterns.

that select a narrow range of data. The table has 500 tuples and a minimum block size of 100 tuples.

Fig. 2b shows the layout for the dataset that qd-tree/MTO would find for the given workload. The root node of the expression tree has the predicate `X < 50` and the two inner nodes have predicates on Y. The tuples that satisfy `X < 50 AND Y < 25` are in block B_0, the tuples that satisfy `X < 50 AND NOT(Y < 25)` are in B_1, and so on. The black dotted lines in the grid represent the physical partition boundaries. When a query with filter `5 < Y < 10` arrives, we first intersect it with the filter in the root node expression `X < 50`. As they are not comparable, the query may intersect both child nodes. The query intersects with the left side of both children, so we only scan the blocks B_0 and B_2, scanning half the data in the table. Fig. 2b highlights the nodes traversed during query execution using this tree. Most queries in the workload will also scan half of the table. Queries with a filter on X can use the root node to skip half the data, and those with a predicate on Y can use the child nodes with an expression on Y.

An issue with the qd-tree is that correlated expressions `X < 25` and `Y < 25` cannot appear on a path from the root node to a leaf. If they do, then the child node corresponding to `X < 25 AND NOT(Y < 25)` will have very few tuples as the columns are correlated. But if we can create two different trees, then we can create one tree with predicates on X and one on Y as shown in Fig. 2c. One of the trees can be used for skipping queries with filters on X, and the other for queries that filter on Y.

A qd-tree is not only an index but also a partitioning function. Each leaf node maps to one physical data block with precisely the tuples that satisfy the leaf node's expression. But with two index trees, it is unclear what the data blocks are. If both trees have to be partitioning functions, then they have to be perfectly correlated i.e., both trees have to partition the tuples into the exact same blocks. This restricts the space of possible trees significantly as most datasets have outliers of some form.

If, as we do in Pando, we relax the precision of the leaf node expressions and allow them to point to multiple blocks containing a superset of tuples that satisfy the expressions, we can lay out the data to align well with the two trees as shown in Fig. 2c.

Here, each tree is a logical partitioning of the data space. The orange dotted line in Fig. 2c corresponds to the boundaries of the orange partitioning tree with predicates on X. Similarly, the blue dotted lines show the logical partition boundaries of the second partition tree with predicates on Y. Tuples with different colors/shapes represent the physical data blocks. The leaf nodes of the trees map to blocks that intersect its logical subspace. The leaf node of the orange tree corresponding to the expression `X < 25` points to blocks B_0 and B_4 as the blue/purple tuples intersect the subspace of data corresponding to `X < 25`.

Each leaf node in Fig. 2c points to $2/5^{th}$ of the data, more tuples than earlier. But with more expressions stored in the trees, queries scan fewer leaf nodes. The query `5 < Y < 10` can use the second partitioning tree and only intersects the first leaf node that maps to B_0 and B_4. In our workload, most queries intersect only one leaf

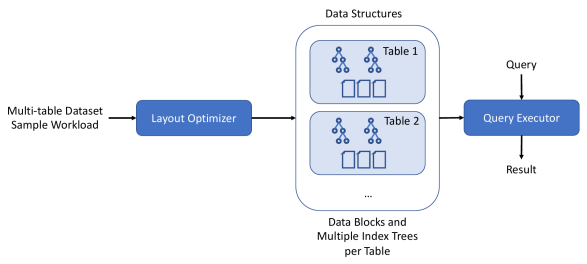


Figure 3: Overview of Pando architecture.

node and scan only $2/5^{th}$ of the data compared to half the data when using a qd-tree. Intuitively, the layout in Fig. 2c reduces the amount of data scanned by splitting correlated expressions like `X < 25` and `Y < 25` across two logical partitioning trees, and assigning tuples to blocks to maximize the utility of the trees.

This is a toy example and a simple correlation between two columns like this can also be addressed by existing techniques like Correlation Maps [22], zone maps [19], etc. But our method goes beyond that and captures the correlation between expressions that exist in the workload. For example, in the NYC taxi dataset, the trees may contain complex predicates such as `DISTANCE(start_point, end_point) < ?` and `cost < ?` that are correlated but where `DISTANCE()` does not exist as an attribute in the database. Pando builds two trees for expressions on `X` and `Y` that are correlated. The X and Y axes in the Fig. 2c correspond to the sort order of the data dictated by the two trees, and it visualizes the correlation between the expressions in the two trees. Our method can also take advantage of correlations across tables. For example, in TPC-H, predicate `l_shipdate > ?` and join-induced predicate `l_orderkey IN (SELECT o_orderkey FROM orders WHERE o_orderdate > ?)` are correlated. Pando leverages this to construct multiple expression trees that work in conjunction and organizes the data to maximize their effectiveness.

Even if the expressions are not correlated, storing multiple trees can perform well and in fact, save space. A single query can combine information from multiple indexes to scan fewer blocks. For example, if columns `X` and `Y` were not correlated and uniformly distributed, the layout corresponding to the 16 grid cells in Fig. 2c can be optimal for a minimum block size of one-sixteenth of the table size. A single tree that can index the 16 blocks will need 15 expression nodes. This single tree can be visualized by appending the second partitioning tree to every leaf node of the first tree. The two trees shown in Fig. 2c can index the 16 blocks with equal expressive power, but with just 6 expressions. A query with a predicate on both `X` and `Y` can combine information from both trees to only access the data blocks that intersect the query. We incur additional overhead in maintaining the pointers to blocks, but it is outweighed by the space benefits from fewer nodes. So we focus on creating multiple partitioning trees instead of one large tree.

3 OVERVIEW

As shown in Fig. 3, Pando is comprised of two components: a layout optimizer which determines what indexes to create and how to lay out the data, and a query executor which executes queries on this layout. Pando organizes data as blocks, disjoint sets of rows, with a minimum size in storage and builds one or more binary partitioning trees that index these blocks (§4). When a query arrives, the query executor gets the data blocks to retrieve using each partitioning tree and combine them smartly to access as few blocks as possible from storage (§5). Given a multi-table dataset, a representative workload, a minimum block size, and a number of indexes to create per table, the layout optimizer finds the data layout and associated indexes for each table to maximize overall performance (§6). Now, we describe these components in more detail.

4 DATA STRUCTURE

Pando organizes data for each table in a database as compressed columnar blocks in storage. A block only contains tuples from one table. Blocks are mutually exclusive, i.e., tuples are not replicated across blocks. There is only one copy of the data for each table.

For each table, we create k logical partitionings. Each logical partitioning is an expression tree that indexes a table's blocks. Each node in the tree is a predicate expression that filters that table's tuples, called cut. For example, the cuts in table `T1`'s partitioning tree can contain single-table expressions like `T1.Col < 5` or join-induced predicates like `T1.Key IN (SELECT T2.Key from T2 WHERE T2.Z > 2)`. The former is useful for queries that have a predicate on `T1.Col`. The latter is useful for skipping blocks for queries that join `T1` with `T2` and have a filter on `T2.Z`.

Unlike qd-trees that cut the physical data space into blocks, Pando's partitioning trees cut the logical data space. The left child of the node corresponds to the logical data subspace that satisfies the expression, and the right child corresponds to the subspace that do not satisfy the predicate. For example, the predicate `T1. Key IN (SELECT T2.Key from T2 WHERE T2.Z > 2)` splits table `T1`'s data space into two subspaces: tuples that when joined with `T2` on `Key` will satisfy `T2.Z > 2` and tuples that will not find a match during join or when joined will not satisfy `T2.Z > 2`.

Each tree partitions the logical data space into mutually exclusive subspaces. Each leaf node corresponds to a predicate based on a conjunction of the expressions (or their negations) in the tree as we traverse the tree from root to leaf. For the rest of the paper, we call this *leaf node expression.* In the first partition tree in Fig. 2c, the first leaf node from left (the one that points blocks B_0, and B_4) corresponds to the expression `X < 50 AND X < 25`. Similarly, the second one has `X < 50 AND NOT(X < 25)`.

Each leaf node contains a set of pointers to the physical blocks that intersect with this logical space. In other words, the leaf node maintains a list of blocks with at least one tuple satisfying the leaf node expression. Note that each leaf node expression is complete, i.e., it points to every block that contains a tuple that satisfies the leaf node expression. However, it is not precise, as there may be tuples in some of the pointer blocks that do not satisfy the expression.

5 QUERY EXECUTION

When a query arrives, we leverage information from all expression trees to minimize the number of blocks scanned. Each index in Pando is a logical partitioning of the data space. We use each index to identify the leaf nodes of the tree that needs to be scanned to answer the query. We traverse down each expression tree to find the leaf nodes that intersect the query filters. At each node, if the

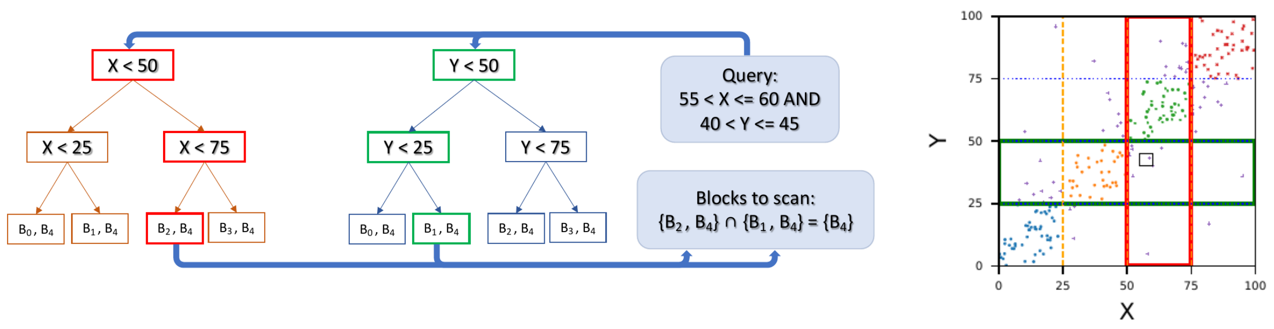


Figure 4: Query execution in Pando for the layout in Fig. 2

query does not intersect the cut in the node, we skip the left child. If it does not satisfy the negation of the cut, we skip the right child. For e.g., consider a node with cut `X < 50`. For query `X > 70`, we skip scanning the left child. Similarly, for the query `X < 10`, we skip the right child. For queries like `X < 60` that intersect both the cut and its negation, or for queries like `Y < 10` where intersection cannot be determined from the expressions, we scan both child nodes.

The set of leaf nodes returned by a tree corresponds to a logical data subspace that subsumes the query i.e., all tuples that satisfy the query are contained in this space. The physical blocks pointed to by these leaf nodes are a superset of the logical space represented by the expressions. Scanning these blocks is sufficient to answer the queries. We obtain the set of blocks to scan from each tree independently and then intersect all the sets to get the set of blocks to retrieve from storage.

Fig. 4 shows the execution of an example query with filter `55 < X <= 60 AND 40 < Y <= 45`. The black rectangle in the grid visualization corresponds to the subspace of data that corresponds to the query filter. The two indexes are used to identify the set of leaf nodes that intersect the query filter. The path traversed is highlighted in each tree. The index on `X` finds the leaf node corresponding to `50 < X <= 75`, and the index on `Y` finds the leaf node corresponding to `25 < X <= 50`. The logical subspace of data that satisfies these leaf node expressions is highlighted by the red/green rectangle in the grid that subsumes the black query rectangle. The leaf node `50 < X <= 75` maps to physical data blocks B_2 and B_4 which correspond to all green and purple tuples. These blocks subsume the red rectangle in the grid which subsumes the query. Similarly, blocks B_1 and B_4 subsume the query. As physical data blocks are mutually exclusive, only the blocks that are in the intersection of the two sets need to be retrieved. So Pando only retrieves block B_4 to answer the query.

6 LAYOUT OPTIMIZATION

The input to our layout optimizer is a multi-table dataset D, a representative workload W, a minimum block size b, and a number of expression trees to create per tree k. The output is a blocked layout for each table such that each block has greater than b tuples and k expression trees per table to index the blocks in the table. We note that our algorithm works for different values of k for each table. For simplicity of presentation, we use one k for all tables in the database. Our algorithm works in two phases.

Top-Down tree construction: We first extract relevant single-table and join-induced predicates from the representative workload and use them to construct k trees per table in a top-down fashion. Trees are constructed to capture different expressions that are useful for accelerating queries in the workload. As we will describe below, correlated expressions that exist in the workload are captured in different trees if they are useful for skipping queries.

Bottom-Up blocking assignment: We then group the tuples in a bottom-up fashion to maximize the utility of these indexes until each block has at least b tuples. This bottom-up clustering of tuples is done to minimize the I/O cost of the queries when executed using the indexes constructed in the previous phase. The expression trees created in phase 1 are updated if required.

We now discuss each step in detail.

6.1 Top-Down Tree Construction

The goal of this stage is to construct k partitioning trees per table, each with predicates that are useful for skipping blocks for queries in the workload. The top-down stage works as follows:

(1) Extract useful predicates from the workload and create a set of *candidate cuts* for each table.
(2) For each table independently: construct k partitioning trees using the cuts from its candidate set to minimize the number of tuples scanned from the table.

First, we extract candidate expressions from the workload for cutting the data space in the tree nodes. For each query in the workload, we extract predicates on each table and decompose them into simple expressions without any conjunctions or disjunctions and add them to a set of candidate cuts for that table. For example, consider the query: `SELECT ... FROM T1 JOIN T2 on T1.Key = T2.Key WHERE T1.X < 100 AND T1.Y > 72 AND T2.Z > 2`. Single-table candidate cuts extracted for table `T1` are `T1.X < 100`, and `T1.Y > 72`. Similarly, `T2.Z > 2` is added to the candidate cuts set for table `T2`. These single table predicates are then propagated through the join graph to create join-induced candidate cuts. The predicate on `T2`, `T2.Z > 2` is propagated to table `T1` through the join `T1.Key = T2.Key` to create the join-induced candidate `T1.Key IN (SELECT T2.Key from T2 WHERE T2.Z > 2)`. This is added to `T1`'s set of candidate cuts. Similarly, `T2.Key IN (SELECT T1.Key from T1 WHERE T1.X < 100)` and `T2.Key IN (SELECT T1.Key from T1 WHERE T1.Y > 72)` are added to `T2`'s candidate cuts. We then construct the k trees independently for each table using expressions from its candidate set.

We give a top-down greedy algorithm, which extends the algorithm for the construction of trees from qd-tree and MTO, to construct k trees, one cut at a time. Throughout the first stage of the algorithm, we assume that each tuple is in a separate block. The iterative algorithm works as follows:

(1) Initialize k trees, each with exactly one node that does not have any cuts and points to every block in the table.

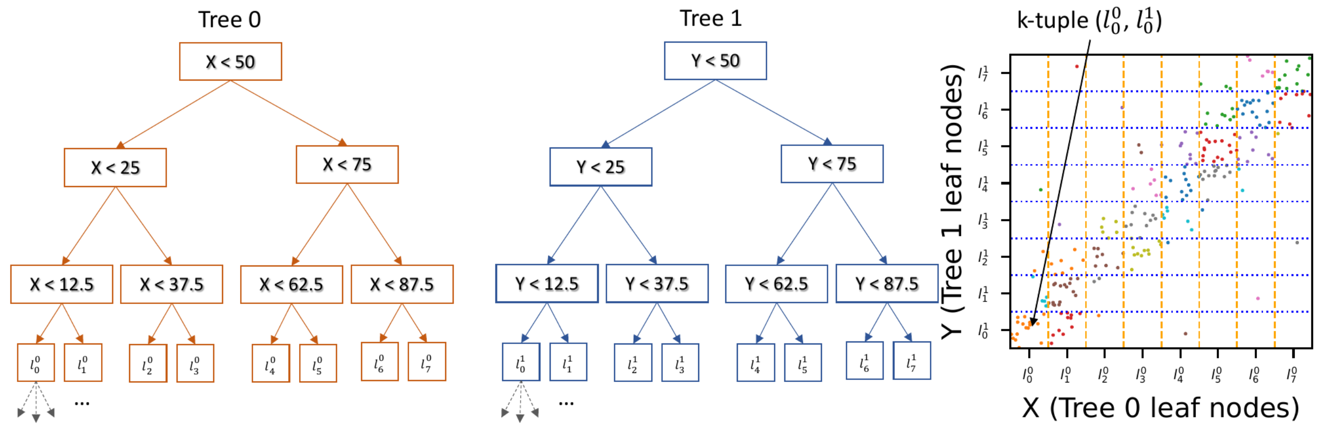


Figure 5: The trees constructed during the top-down phase of Pando's layout optimizer. The grid visualizes the correlation between the leaf nodes of the trees.

(2) For each tree, for each leaf node, for each candidate cut, find the cost of executing the workload if the leaf node is cut using the candidate.
(3) Let (t, l, c) be the tree, leaf node, and candidate cut triple with the lowest cost in step 2. Cut the leaf node l of the tree t using the cut c.
(4) Repeat steps 2 and 3 until one of the following conditions is satisfied:
    (a) Cutting leaf nodes no longer yield any more benefit.
    (b) Cutting leaf nodes result in child nodes that point to fewer than a threshold number of tuples.

We start with k trees with no cuts in any of the trees, i.e., the root node of each tree is also a leaf node. Each leaf node corresponds to the entire logical dataspace and maps to every block in the table (each block is a tuple).

In each iteration of the algorithm, we pick a leaf node of one of the k trees and a cut from the candidate cuts to split the leaf node into two child nodes. This choice is made greedily by picking the tree/leaf node/cut combination that provides the maximum I/O benefit for the workload. We consider a cut only if the child nodes after the node is cut have greater than a threshold number of tuples. The threshold number of tuples can be varied to trade off optimization time for higher-quality layouts (discussed in §6.2). The child nodes become new leaf nodes that are considered for cutting in later iterations. We continue splitting leaf nodes until no more nodes can be cut or cutting nodes yields no benefit.

The cost model in step 2 is the I/O cost which is simply the total number of blocks scanned (the same as the number of tuples as each tuple is in a separate block) for all queries in the workload. The I/O benefit is estimated using the query execution described in §5 i.e., for each query we only scan the blocks that are returned from all the trees. A cut c is useful for cutting a leaf node l in tree t only if queries can skip blocks using c that they could not already skip using predicates in other trees. This means that the benefit of a cut in one tree depends on the cuts in other trees. Our cost estimation accounts for using multiple trees to skip blocks for each query so that we do not end up with k identical trees. Using these trees can reduce the cardinality of the scan output (eg. join-induced cuts pre-filter tuples that would otherwise be dropped during join), thus can have an impact on query plan (eg. join order). Our cost model does not account for that and only models the I/O cost.

This algorithm picks expressions that are useful for minimizing the I/O needed for the workload when the indexes are used in conjunction. The expressions that yield a higher reduction in the amount of data scanned for most queries appear at the higher levels of the tree. For example in our running example, the predicates `X < 50` and `Y < 50` can reduce the number of records scanned by half for queries that filter in `X` and `Y` respectively, so will be picked first.

We note that this algorithm tends to pick correlated expressions in different trees, and not in the same tree. To understand why correlated expressions tend to be picked in different trees, consider `Y < 25` being considered as a candidate to cut the left child node `X < 25`. The benefit of this cut for a query that does not intersect the negation of the predicate is bounded by the number of tuples that satisfy `X < 25 AND NOT(Y < 25)` which is a very small number. Whereas in another tree, where the path to the leaf node has not been cut by `X < 25`, the benefit is likely higher. Our algorithm will pick such a leaf node if it is beneficial for the overall cost of the optimization process.

Now we discuss, how to scale this phase of the optimization algorithm for large datasets. Our cost model is the total amount of data that is scanned by the queries in the workload when k trees are used in conjunction. This requires finding the tuples that satisfy the leaf node expression for each candidate cut considered which can be very expensive. To reduce the runtime of the optimization algorithm, we run this stage using a sample of the dataset for estimating the cost, similar to qd-tree and MTO. This is done for all the tree, leaf node, and candidate cut triplets considered in step 2 in parallel.

At the end of this stage, we have constructed k trees for each table that cut the logical data space using a variety of expressions that are useful for avoiding unnecessary data access during query execution. The cuts that yield higher benefits or are useful for a large number of queries tend to be in the higher levels of the trees and correlated expressions if useful for block skipping tend to be in different trees. Fig. 5 shows the trees that are constructed by the algorithm with a threshold size of one-eighth of the table. We use l_j^i to denote the j^{th} leaf node of the i^{th} tree. We now discuss how we find the block assignment to maximize the benefit of these trees.

6.2 Bottom-Up Block Assignment

The goal of the second stage of the algorithm is to assign tuples to blocks that maximize the utility of the indexes in skipping blocks for the given workload. Each tree partitions the logical data space into mutually exclusive subspaces, so each tuple satisfies the leaf node expression for exactly one leaf node in a given tree. Logically, each tuple has an associated k-tuple whose i^{th} entry is the id of the leaf node of the i^{th} tree that the tuple belongs to. The grid in Fig. 5 illustrates the mappings between tuples and leaf nodes for the two trees constructed in the top-down phase of our algorithm. The two axes in the visualization correspond to the leaf nodes of the two trees, i.e., each partition along each axis corresponds to a leaf node in a tree. Each cell in the grid formed by these partitions corresponds to a k-tuple. In our running example, consider the tuple $(X = 5, Y = 5)$ in our running example. When routed down the first partitioning tree with predicates on X, this lands in the leaf node l_0^0. Similarly, it lands in the leaf node l_0^1 in the second tree. So this tuple has an associated k-tuple (l_0^0, l_0^1).

The goal of our bottom-up clustering algorithm is to merge the tuples into blocks such that queries when executed using the indexes created in the first phase scan as few blocks as possible. To minimize the number of blocks, we need to group tuples that are accessed together in the same block. In other words, tuples should be assigned to blocks such that queries that need to scan one tuple in the block also need most other tuples in the block. Outliers are those tuples that hinder this, i.e., they are needed for answering queries that scan very different sets of tuples. So however you assign these outlier tuples to blocks, it is likely that many queries that need to scan the outlier tuples do not require other tuples in the block. As trees in the first phase were constructed to align the leaf nodes well with the queries assuming each tuple is in a separate block, it is likely these outlier tuples are also outliers in the leaf-node expression space that is visualized in the grid Fig. 5.

In our example, nearby tuples along the diagonal `X=Y` are good candidates to be in the same block as most queries of both templates (the ones with a predicate on `X`, and ones with a predicate on `Y`) that intersect them scan many of the nearby tuples. In contrast, a tuple far away from the diagonal, such as `X=15, Y=100`, is an outlier. Queries with a predicate on `X` that need this tuple mostly scan the bottom left corner of the grid, whereas queries with a predicate on `Y` that need this tuple scan the top right corner of the grid. So if you group the outlier tuple with ones in the bottom left, then queries on `Y` will scan an unnecessary amount of data. Similarly, if you group it with tuples in the top right corner, then queries on `X` will scan irrelevant tuples. So to maximize skipping, we should minimize the number of blocks with outliers by assigning the outlier tuples to separate outlier blocks.

We now describe our bottom-up algorithm that clusters tuples to minimize the total amount of data scanned using these trees.

(1) Route the tuples down each partitioning tree to find the associated k-tuple.
(2) Merge all tuples with the same k-tuple into a block.
(3) For each pair of blocks, find the cost of merging them.
(4) Merge the pair with the lowest cost from step 3.
(5) Repeat steps 3 and 4 until each block has greater than the required minimum block size b.

All tuples that have the same k-tuple cannot be differentiated by the trees that we have constructed, so there is no benefit in spreading the tuples across blocks. So all the tuples that satisfy the same k-tuple are merged into the same block. In Fig. 5, each k-tuple corresponds to a cell in the grid. Thus, all tuples that fall in the same grid cell are merged into a block.

We then iteratively merge these cells until each block has more tuples than the minimum block size b. We start with each block having just one k-tuple, corresponding to one cell in the above grid. In each iteration, we pick two blocks to be merged into one. We make this choice greedily by merging two blocks that incur the least cost for executing the queries. We continue merging blocks until each block has more tuples than the minimum number of tuples in a block, b.

To compute the cost of merging two blocks, we use a cost model that is the total number of records that are scanned according to query execution described in §5. For each query, we get the set of blocks to scan from each partitioning tree and scan the ones that are returned by all trees. When we merge two blocks B_i and B_j, all queries that scanned block B_i or B_j will now scan the data that is in both blocks. It is also possible that some queries that did not access either of the blocks will now scan the merged block. To see this, suppose that, before merging, the blocks to be scanned returned for a query Q in tree 1 are (B_i, B_k), and in tree 2 are (B_j, B_k). The query would only scan B_k as that is the only block that is in the intersection of sets returned by the two trees. After merging, both trees now return (B_{ij}, B_k) where B_{ij} is the merged block, so both blocks will be scanned during query execution.

Intuitively, two blocks are merged if most queries that scan one block also scan the other. In the running example, this is true for blocks along the diagonal of the data space. The outlier tuples are scanned by queries that scan a variety of other blocks. Outliers k-tuples that are fewer in number get merged into their own block. This outlier block is unlikely to be merged with the diagonal block as that will incur high costs for most queries that scan either of the blocks. This handling of outliers is essential for finding a block assignment that is good for the indexes. We eventually end up with a block assignment shown in Fig. 2c.

At this point, many sibling leaf nodes in the partitioning trees will point to the same set of blocks. A cut in the tree is not useful if both child nodes point to the same set of blocks. We truncate the trees by merging nodes of the tree in a bottom-up fashion, eliminating cuts if they provide no benefit. For example, both child nodes of `X < 12.5` in Fig. 5 will point to blocks B_0 and B_4. The cut can be eliminated and the node corresponding to `X < 12.5` can be made a leaf node that points to B_0 and B_4. This is done until all sibling nodes have different block mappings, thus yielding a tree shown in Fig. 2c.

We now discuss how to scale this phase of the algorithm. Step 3 can be done in parallel and the cost model depends on the number of tuples in each block, which can be easily maintained while merging blocks. If a table has NT tuples and each tree has NL_i leaf nodes, the number of possible values of k-tuples, NK, is $min(NT, \Pi NL_i)$ which can be huge. The complexity of this phase is quadratic in NK. Capturing correlated expressions in separate trees in the top-down phase results in many of these cells being empty, making this bottom-up algorithm tractable. The threshold parameter we have in the first phase can be tuned to trade-off time spent in optimization for a higher quality layout. The smaller the threshold, the more the number of leaf nodes, the more the number of distinct k-tuples, and the longer it takes for merging. But smaller thresholds can capture finer-grained outliers improving the layout quality.

We note that Pando with $k = 1$ is the same as MTO. We can use b as the threshold for the top-down algorithm and we will create the same tree as MTO. For values of threshold lower than b, the top-down phase will pick additional predicates in the deeper levels of the tree, which will be removed during the bottom-up phase. Increasing k will improve the performance of the layout with diminishing returns. However, this comes at the cost of an increasingly larger offline optimization time. In theory, the optimal value of k can be arbitrary as workloads could contain an arbitrary number of distinct expressions that are correlated. We did not observe many such expressions in the workloads at Meta and expect small values of k would be sufficient for most workloads. Based on our experiments in §8.3, we expect 3 or 4 trees to be sufficient for many realistic datasets. To pick k, we can evaluate the query performance for increasing values of k on a dataset sample until the query performance flattens.

7 DATA CHANGES AND WORKLOAD SHIFTS

In this section, we describe how Pando adapts to evolving data and workload distributions. Pando supports both bulk and regular data changes. Most analytic data systems handle regular inserts, updates, and deletes by absorbing them in a delta store which is periodically merged with the main data store, effectively turning point modifications into batch updates [26]. Any data layout that maintains data in some sorted order, including simple schemes like single-column range partitioning will impose some overhead as a result of merging to maintain its performance advantages. Pando faces two types of performance challenges related to updates.

First, mapping from leaf nodes of the partitioning tree to physical blocks can become stale with data changes because of join-induced predicates. Consider a join-induced predicate `T1.Key IN (SELECT T2.Key FROM T2 WHERE T2.X > 10)` that is present in one of the partitioning trees of `T1`. When the tree was constructed, suppose this predicate was equivalent to the *literal cut* `T1.Key IN (1, 2, 7)`, i.e., `SELECT T2.Key FROM T2 WHERE T2.X > 10)` evaluated to `(1, 2, 7)`. When a new tuple, say `(Key:10, X:15)`, is added to `T2`, the join-induced predicate is now equivalent to `T1.Key IN (1, 2, 7, 10)`, but the mappings in the leaf nodes still correspond to the old literal cut. When data is inserted to `T2`, the mapping from logical blocks to physical blocks in `T1` might be stale and requires remapping. This can be expensive.

We use the same approach as MTO for handling inserts and deletes. Specifically, for the case of induced join predicates, we restrict the joins to be foreign key-primary key joins and only induce predicates from the primary key table to the foreign key table. Continuing the example, we use the cut `T1.Key IN (SELECT T2.Key FROM T2 WHERE T2.X > 10)` in `T1`'s partitioning tree only if `T2.Key` is the primary key of `T2` and `T1.Key` is a foreign key that references it. More generally, for star schemas, we induce predicates from a dimension table to a fact table, but not from a fact table to a dimension table. Intuitively, this has minimal impact on performance as dimension tables are typically smaller than fact tables. Assuming referential integrity, data inserts or deletes do not affect the logical to physical block mapping. In our example, none of the tuples in `T1` will have `Key=10`, the newly inserted unique value in `T2`, so the mappings in `T1`'s trees are still correct.

To route tuples down the partitioning tree during insertion, we need to evaluate the predicate in each node. To do this efficiently for join-induced predicates, we store the literal version of the cut in each node, as in MTO. This can potentially be very large, so we store them as compressed bitmaps. This literal cut needs to be maintained in the presence of inserts and deletes. This can be performed by evaluating the relevant join-induced predicate on only the inserted or deleted records, not the entire table. In our example, the join-induced predicate is evaluated on the newly inserted tuple in `T2`, `(Key:10, X:15)`, to update the literal cut from `T1.Key IN (1, 2, 7)` to `T1.Key IN (1, 2, 7, 10)`. Updates can be implemented as a delete followed by an insert but may cause existing tuples to shift between blocks.

The second update-related challenge is that Pando's logical partitioning trees do not give a unique block to insert the data into. To merge deleted records from the delta store, we need to locate the blocks with the deleted records; Pando's logical partitioning trees can be used to find those blocks. However, if we route a tuple down k partitioning trees in Pando, we will get a set of possible blocks the tuple could be in. In Pando, each block is a collection of k-tuples. To support point (trickle) inserts, we can maintain the non-truncated trees and the mapping from k-tuples to blocks. This can be used to route tuples to blocks during insertion.

While this can work well for small in-distribution inserts, most analytics systems ingest data in large batches and contain out-of-distribution data. For example, a day's worth of data is loaded into the table by nightly ETL jobs, and the data contains date/timestamp columns, serial ids, etc. with increasing values that are out-of-distribution (unseen during layout optimization done in the past). Each such new batch is written into a new partition, naturally resulting in a partitioning on the data/timestamp column as data is usually inserted in the order of these columns. Most queries that run on this data filter on these columns. Pando's layout optimizer can be the last step of these ETL jobs, which optimizes the layout for each bulk insertion batch independently and writes the data blocks according to this layout. Intuitively, this is a two-level partitioning. At the first level, data is partitioned based on the date column, and each such partition is organized into blocks using Pando at the second level. Tuples that arrive out-of-order can be buffered and periodically merged with data that is already partitioned according to Pando as described above. This is similar to how systems like Amazon Redshift maintain data with sort keys [4]. The staleness of the layout from shifts in workload and data distribution can be monitored for each partition. Reorganization of data can be triggered at the granularity of each partition if required.

Decoupling logical and physical partitioning gives more degrees of freedom for evolving the index structures and the data layout. Logical partitioning trees can be adapted in a lightweight manner without reorganizing the physical data blocks. Physical layout of the data can be reorganized based on how well the logical structures align with the physical block boundaries. We defer exploring these strategies to future work.

8 EVALUATION

In this section, we present an experimental evaluation of Pando. In our experiments, we seek to answer the following questions.

(1) How does Pando compare against other baselines on the amount of I/O performed and overall query execution time?
(2) What is the computational and storage overhead in finding the optimized layouts, organizing data according to it, and storing the associated metadata?
(3) How does the performance advantage of Pando scale with dataset and workload complexity?
(4) How does dynamic data affect Pando's performance?

8.1 Datasets and Workloads

We evaluate Pando on TPC-H [43], TPC-DS [42], and two production workload traces from Meta. We use a scale factor of 100 for the synthetic datasets which corresponds to 100GB of data. For TPC-H, we support all 22 templates and use a workload with 176 queries, 8 queries per template. For TPC-DS, we use 184 queries, 4 queries for each of templates 1-50, except for 14, 23, 24, and 39, which contain multiple queries. We use two real-world datasets from Meta. Meta 1 is a single table dataset. We use a sample from one day's data amounting to around 40 GB of data. Queries are generated from an interactive query engine and involve many correlated `LIKE` predicates on different attributes. The second dataset, Meta 2, contains 4 tables amounting to 120GB of data that powers an interactive dashboard. Queries involve joins over these tables with a variety of predicates on each table. We use a sample of 100 queries from the production traces that were generated on one day in our experiments.

8.2 Experimental Setup

We implemented Pando's layout optimizer and a simulator for the number of blocks accessed during query execution in Python. We performed a shallow integration with Meta's internal version of Spark. We use Hive [41] for storing the data in our experiments. We augment each table in the dataset with an additional partition column `block_id`. This field is the block id for each tuple obtained after running Pando's layout optimizer. Each table is partitioned on `block_id` in the Hive cluster. Each block is stored as an ORC file. Integer columns are run-length encoded and strings are dictionary encoded which are in turn compressed using Zstandard algorithm. We augment each query with an additional filter for each table of the form `block_id IN ()` with the block ids obtained from Pando index data structures. For example, if the index on T1 return blocks 1 and 7, and the index on T2 returns blocks 8 and 14 for the query `SELECT * FROM T1 JOIN T2 on Key WHERE T1.X < 17 AND T2.Y >= 45`, we execute the query `SELECT * FROM T1 JOIN T2 on Key WHERE T1.X < 17 AND T2.Y >= 45 AND T1.block_id IN (1, 7) AND T2.block_id IN (8, 14)`. Spark uses this filter to skip unnecessary blocks. The partition column `block_id` is encoded in the directory path and not materialized as a part of the table's data in Hive. So the overhead of this integration on query performance is minimal. We report the end-to-end query execution time for a single-node spark environment.

We compare Pando against two other layouts. First, a baseline, that range partitions each table on a single user-tuned column. For TPC-H, we sort lineitem by shipdate, orders by orderdate, and all other tables by primary key for the baseline. 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. For Meta datasets, we use the same single/multi-column range partitioning as in the production instance. Second, we use MTO as the instance-optimized layout that organizes data according to a single hierarchical physical partitioning tree.

Data-induced predicates (diPs) [21] is a recently proposed technique to propagate predicates that are pushed down on one table through joins to benefit from block skipping on other tables at query execution time, similar to join-induced predicates in Pando. We use diPs to show that baseline layouts along with runtime optimizations cannot achieve the same speed up as a layout that is aware of correlated expressions. We use this in our simulator to show the reduction in I/O. Spark does not support diPs and it needs a deeper integration into Spark's query optimizer and execution engine. So we could not show end-end runtime numbers with diP.

We performed all our experiments on a machine running CentOS Stream 8 equipped with a 72core Intel Core Processor (Broadwell) with 224GB RAM and a 2TB Toshiba XD5 local SSD. We use a block size of 1M tuples for all our experiments which is the default for many systems like Microsoft SQL Server [26]. We optimized the layout using 50% of the workload and evaluated using the remaining half. We used a 1% sample of the data for optimization. The threshold number of tuples for the top-down stage is set such that there are under 10000 k-tuples for the bottom-up stage. This means the more the number of trees, the larger the leaf nodes of each tree, and the smaller each tree. We report the end-to-end query execution time, the number of blocks accessed during query processing, the space overhead in storing the logical partitions, and the time overhead in optimizing the layout and reorganizing data.

8.3 Results

Fig. 6 shows the overall performance on various layouts. Pando-k means, we created k trees for each table in the dataset. Fig. 7 plots the number of blocks that are accessed during query execution.

Overall Performance: Pando is consistently better than MTO and range partitioning (the baseline) across datasets and workloads. Pando achieves up to 4.2X faster query performance than range partitioning and up to 2.3X faster performance than MTO. Pando attains up to 9.5X reduction in I/O compared to the baseline and up to 2.8X reduction compared to MTO. By capturing correlated expressions in the workloads, Pando is up to 2X faster than MTO by creating just one additional logical partitioning tree. As the number of indexes increases, the performance of the layout also increases, but with diminishing returns. For all the workloads we tested on, adding a fourth tree resulted in minimal performance advantages. The trends in the amount of data that is scanned are similar to the overall performance as shown in Fig. 7.

Impact of diPs: diPs compute join-induced predicates at query optimization time to improve skipping blocks. Its effectiveness depends on how the data is blocked and is sensitive to outliers. While this gives some benefit over all the layouts, it does not match the performance of Pando that co-optimizes its indexes and the data layout. diPs decrease the I/O done by MTO as well as Pando.

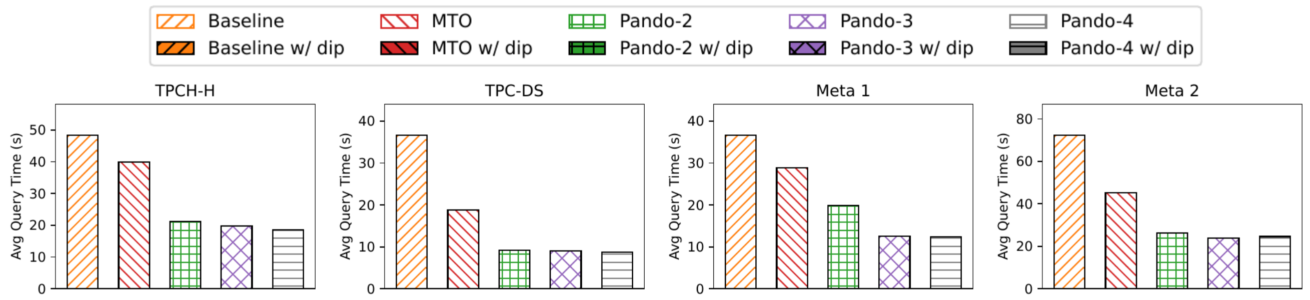


Figure 6: End-to-end query execution time on Spark.

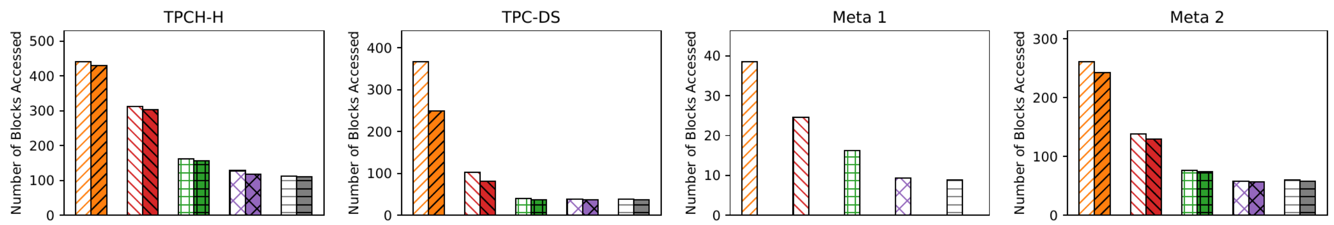


Figure 7: Average number of blocks accessed during query execution. Data-induced predicates (diPs) is not relevant to Meta1 as it does not involve any joins.

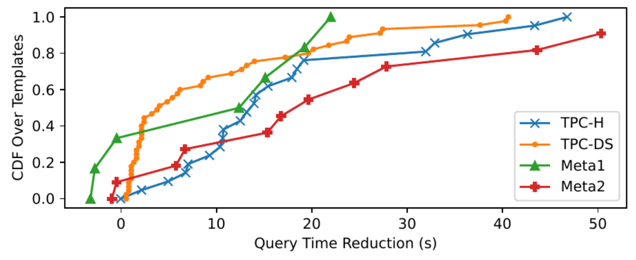


Figure 8: Reduction in query times achieved by Pando compared to MTO for each query template.

But Pando still achieves a better reduction in I/O than MTO with diPs. Meta 1 is a single table dataset and the workload does not involve any joins, so diPs are not relevant.

Where does the advantage come from?: Meta 1 is a single table dataset with many `LIKE` predicates on different columns in the table. Several of these predicates are correlated and Pando leverages these in creating multiple partitioning trees to improve query performance. Pando accelerates query performance in the other three datasets by capturing correlated expressions across tables. Let us look at TPC-H as an example. Clearly, partitioning the `lineitem` using predicates on `l_shipdate` is useful for TPC-H queries. Attributes `l_shipdate` and `o_orderdate` are correlated, so are the predicates `l_shipdate > ?` and `l_orderkey in (SELECT o_orderkey FROM orders WHERE o_orderdate > ?)`. A single physical partitioning does not fully capture these predicates in the same tree, thus limiting the skipping opportunities. In contrast, Pando effectively distributes these predicates over multiple trees and optimizes the data layout to maximize its effectiveness. This improves skipping for queries that predicate on `l_shipdate` as well as that join `lineitem` and `orders` with a predicate on `o_orderdate`. Fig. 8 shows the reduction in query time Pando achieves compared to MTO for different query types. As Pando optimizes for the overall workload, performance may regress for some query types.

Index Size: Fig. 9 plots the space required in storing the partition trees. This includes the space required for the literal cuts, non-truncated trees, and the mapping from k-tuples to block id that is required for supporting data changes. Without including these, all schemes take under 1MB for all their indexes for each of the datasets. The size of the index with joins-induced cuts can be high, depending on the number of cuts in the tree, selectivity of joins, number of unique keys, etc. As discussed earlier, storing multiple trees has the same expressive power as a much larger single tree. So storing multiple trees can also save space. The takeaway from Fig. 9 is that the partitioning trees that index data blocks are not prohibitively large like secondary indexes. The size of the index does not increase linearly with the number of partitioning trees. We are able to achieve better performance without a major increase in the space requirements for the index.

Offline Optimization Time: Instance-optimized data layouts are only useful when they can be efficiently found and organized. The offline optimization time required for Pando is shown in Fig. 10. `opt time` (optimization time) is the time spent in finding the optimal layout. In Pando, this includes the time spent in the top-down and bottom-up stages of the algorithm. `routing time` is the time spent in locally partitioning the full dataset using the tree indexes, i.e., finding the tuple assignment for each block. This combined is the overhead that Pando adds over other methods. After local partitioning, each partition is uploaded to the Hive cluster. All partitioning schemes with the same minimum block size take roughly the same time to upload the partitions to Hive, so we do not include it here. In general, the optimization time and data routing time increase with more trees. Fig. 10 shows that overhead is in ~10s of minutes for datasets with $\sim$ 100 GBs of data. For the workloads that we tested on, Pando-2 offsets this overhead in as low as 7 queries for Meta1 and as high as 37 queries for TPC-H. The minimum number of queries is 7 and the maximum is 36 for Pando-2 to offset the overhead over MTO. A C++ implementation of the layout optimizer may be faster, and the absolute overhead incurred by Pando may be smaller. Pando can make up for the overhead in fewer queries.

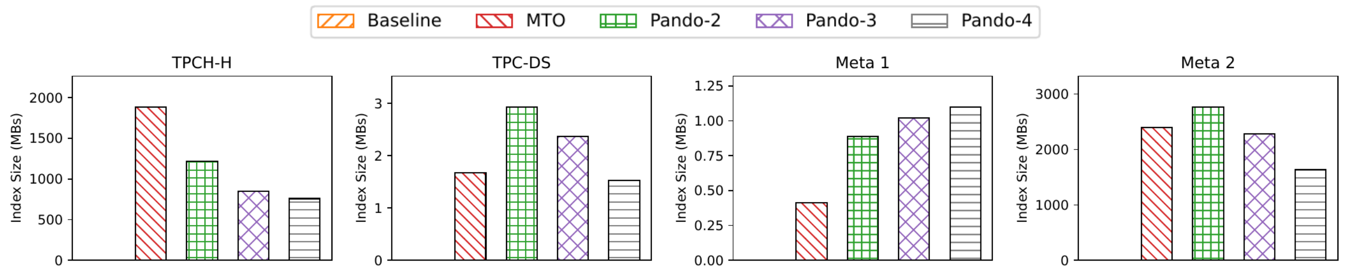


Figure 9: Size of the partition tree indexes.

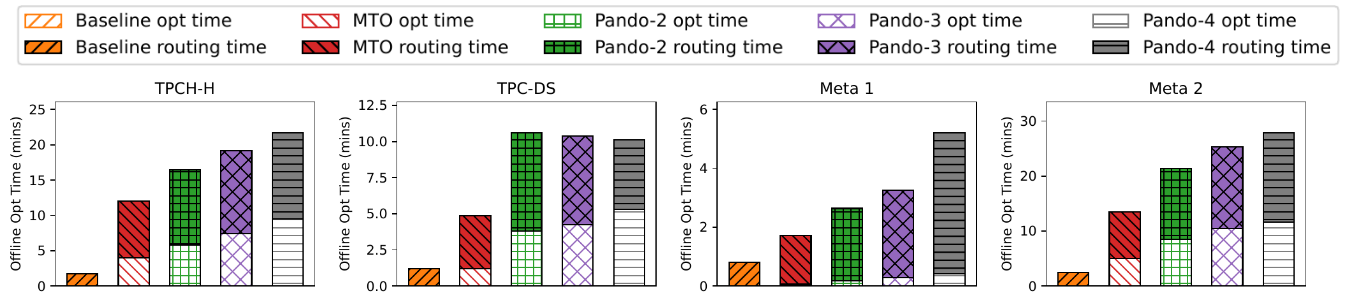


Figure 10: Time overhead incurred by Pando in offline optimization.

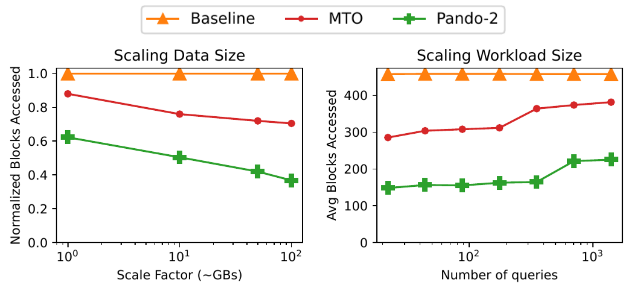


Figure 11: Pando's performance across varying dataset sizes, and workload sizes.

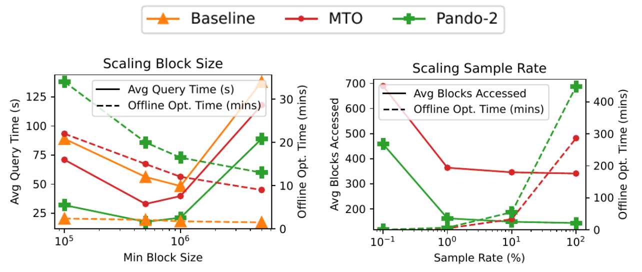


Figure 13: Pando's performance across varying minimum block sizes, and optimizer sample rates.

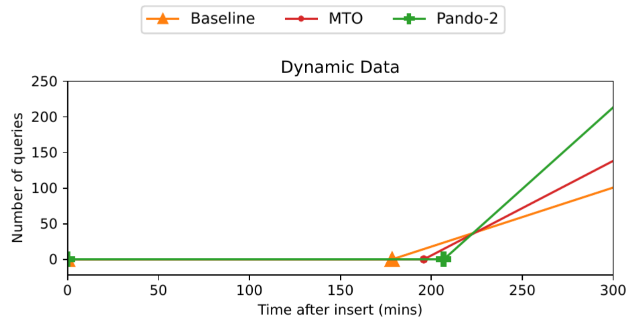


Figure 12: Pando's performance with data inserts. Although inserted data is available for querying quickly in the baselines, Pando makes up for it with faster execution of queries.

Scalability: Fig. 11 shows that Pando maintains its performance advantage over the baseline across varying dataset and workload sizes. To study the impact of increasing dataset sizes, we vary the scale factor on TPC-H from 1 to 100, resulting in 1 to 100GBs of data. We use a workload with 176 queries, 8 per template, and set the minimum block size to 1 million tuples. Fig. 11 plots the number of blocks scanned normalized against the number of blocks scanned by the baseline as the total number of blocks varies with data size. Pando (and MTO) performs better at higher dataset sizes. This is because larger datasets have a larger number of blocks and the partitioning tree can store more expressions for more fine-grained skipping. To study the impact of scaling the workload size, we vary the number of queries per template in the workload from 1 to 64. With 22 templates, this results in workloads ranging from 22 to 1408 queries. We use scale factor 100 for dataset size. Fig. 11 shows that the number of blocks scanned by Pando (and MTO) increases as the workload size increases. This is because, with more templates, we need to store more expressions to achieve better skipping. This affects Pando slower than MTO. Pando continues to perform better than other baselines across varying workload sizes.

Dynamic Data: To show how Pando adapts to data changes, we use the Meta2 dataset. We start with one day's worth of data that is already optimized according to Pando in the database. We then bulk-insert another day's worth of data into this database as described in §7. Pando spends around 9.6 mins in repairing the join-induced predicates in the previous day's partitioning trees and MTO spends around 6.4 mins. Optimization and local data partitioning take 21.3 mins for Pando, 13.4 mins for MTO, and 2.4 mins for the baseline.

Meta's implementation of Hive does not support repartitioning the data based on a new column. So we have to repartition the data separately and insert each partition separately into the Hive cluster by explicitly mentioning the value of the partition column (aka the directory path). Uploading the data to Hive takes around 3 hrs in our setup. As shown in Fig. 12, Pando is able to quickly recover the additional time spent in optimization and repair and surpass the baselines in the number of executed queries.

Sensitivity: Fig. 13 shows the impact of varying minimum block sizes (100k, 500k, 1M, and 5M tuples) on query performance and offline optimization time using TPC-H with scale factor 100 using 8 queries per template. The offline optimization time decreases with increasing block size. The query latency initially decreases as queries scan fewer data, but eventually increases due to too many small block accesses. We study the impact of varying sample rates during layout optimization (0.1%, 1%, 10%, and 100%) on Pando's query performance and optimization time. We use TPC-H with a scale factor 10, 8 queries per template, and a minimum block size of 100k. Pando is able to significantly reduce the optimization time by sampling without significantly affecting the layout quality.

9 RELATED WORK

Data Partitioning: Carefully organizing data on storage for accelerating the performance of queries is a well-studied area. Most production systems often partition the data either based on ingestion time, or using range, hash, or round-robin distribution schemes [26]. Several techniques have been proposed to tune the data layout to reduce the amount of data that is scanned using horizontal, vertical, and hybrid partitioning [3, 5, 12, 17, 36, 37]. To prune the set of the blocks that are scanned, these systems maintain additional statistics along with these blocks such as minimum and maximum values for each attributes called Small Materialized Aggregates (SMAs) [31], and zone maps [19]. Slalom [33] logically partitions the data and build lightweight partition-specific indexes, but they do not consider optimizing the physical layout of the data. [18] combines physical and virtual partitioning to fragment and dynamically tune partition sizes for efficient intra-query parallelism, but not data skipping. Distributed database systems employ various strategies to partition data between nodes to minimize the number of cross-partition queries [13, 20, 28]. Pando can be used to optimize the layout within each node.

Learned Layouts: Recent work has shown that significant gains in query performance can be achieved by more expressive partitioning schemes that collect rich metadata and can adapt to datasets and workloads [15, 16, 24, 29, 38, 45]. Qd-tree [45] and MTO [15] are learned layouts that maximize data skipping in cloud-based DBMSs (discussed in this paper). Feature-based data skipping [39, 40] maintains a bit vector per block where each bit indicates whether a predicate corresponding to the bit is satisfied by any tuple in the block. These methods and Pando use bottom-up clustering to find the data layout, but their method only scales to a small number of predicates and we store much richer metadata in the form of multiple expression trees. Qd-trees [45] already showed better performance than feature-based skipping with just one partitioning tree. Tsunami [16] is a learned partitioning scheme that is aware of correlations, but they only capture simple correlations in data and only work for multi-dimensional range queries on in-memory databases. CopyRight [38] co-optimizes multiple expressive partitionings, but replicates data and creates one partitioning per replica.

Automated Physical Database Design: There is a rich corpus of work on automatically tuning a database and its physical design that focuses on selecting structures like indexes and materialized views for optimizing query performance [1, 2, 9, 10, 35]. Each partitioning tree in Pando is a coarse-grained index that indexes data blocks. Many index selection works consider co-optimizing the layout along with indexes [1, 3, 10]. However, they only work for simple schemes like single or multi-column range and hash partitioning, b-tree indexes, etc. Materialized view selection algorithms explore the space of syntactically relevant views, focusing on creating logical views that cover queries, whereas we focus on optimizing the physical layout of the table [2, 8, 46]. Pando can be used to partition materialized views.

Correlation Awareness: Prior work has explored several methods to take advantage of column correlations such as automatically discovering algebraic constraints [6], soft functional dependencies [7], and approximate dependencies between columns [25] in the data. Our method goes beyond these methods and captures correlations using predicates that appear in the workload. Various techniques such as Hermit [44], Correlation Maps [22], Cortex [32] exploit correlation among columns to improve the performance and reduce the size of the secondary indexes. Hermit [44] captures monotonic functional dependencies between attributes, while Correlation Map [22] maintains a mapping between correlated columns. Cortex and Pando look at outliers from a similar lens of query performance, but Cortex focuses on capturing correlation between a user-provided set of columns. None of these techniques consider optimizing the physical layout of the table. CORRADD [23] is a correlation-aware physical database designer but they explore a smaller search space that consists of only single-column sort orders for physical layouts and single-column b-tree indexes.

10 CONCLUSION

I/O for accessing data from storage is a major bottleneck for cloud-based data analytics systems. Modern cloud-based database systems employ a variety of strategies to partition the data into blocks and skip accessing irrelevant blocks during query execution. The effectiveness of these systems depends on how well the data layout and the associated metadata align with the queries that are run on it. Existing methods miss substantial opportunities for block skipping blocks because they do not collect enough metadata or do not exploit correlated predicates in the workloads. In this paper, we propose Pando a metadata-rich data layout framework that achieves a significant reduction in the amount of I/O performed by jointly optimizing the physical layout of the data and multiple correlation-aware logical partitionings. Our experiments show that Pando achieves up to 2.3X speedup in end-to-end query execution time and up to 2.8X reduction in the number of blocks scanned compared to the state-of-the-art data layouts.

ACKNOWLEDGMENTS

This work was supported by the MIT Data Systems and AI Lab (DSAIL).

REFERENCES

[1] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek Narasayya. 2006. AutoAdmin: Self-Tuning Database Systems Technology. *IEEE Data Engineering Bulletin* (2006), 7–15.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505.

[3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 359–370. https://doi.org/10.1145/1007568.1007609

[4] Amazon [n.d.]. Managing the volume of merged rows. https://docs.aws.amazon.com/redshift/latest/dg/vacuum-managing-volume-of-unmerged-rows.html.

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

[6] Paul G. Brown and Peter J. Hass. 2003. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 668–679.

[7] Paul G. Brown and Peter J. Hass. 2003. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 668–679.

[8] Nicolas Bruno and Surajit Chaudhuri. 2007. Physical Design Refinement: The 'Merge-Reduce' Approach. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 28–es. https://doi.org/10.1145/1292609.1292618

[9] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 3–14.

[10] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 146–155.

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

[12] Douglas W. Cornell and Philip S. Yu. 1990. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Softw. Eng.* 16, 2 (Feb. 1990), 248–258. https://doi.org/10.1109/32.44388

[13] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).

[14] Databricks [n.d.]. Data skipping index. https://docs.databricks.com/delta/data-skipping.html/. Accessed: 2022-12-01.

[15] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 418–431. https://doi.org/10.1145/3448016.3457270

[16] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (nov 2020), 74–86. https://doi.org/10.14778/3425879.3425880

[17] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. [n.d.]. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management.

[18] Camille Furtado, Alexandre A. B. Lima, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. 2005. Physical and Virtual Partitioning in OLAP Database Clusters. In *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing (SBAC-PAD '05)*. IEEE Computer Society, USA, 143–150. https://doi.org/10.1109/CAHPC.2005.32

[19] Goetz Graefe. 2009. Fast Loads and Fast Queries. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery* (Linz, Austria) *(DaWaK '09)*. Springer-Verlag, Berlin, Heidelberg, 111–124. https://doi.org/10.1007/978-3-642-03730-6_10

[20] 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*. 143–157.

[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), 252–265. https://doi.org/10.14778/3368289.3368292

[22] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. 2009. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. *Proc. VLDB Endow.* 2, 1 (aug 2009), 1222–1233. https://doi.org/10.14778/1687627.1687765

[23] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. 2010. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 1103–1113. https://doi.org/10.14778/1920841.1920979

[24] Tim Kraska, M. Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, S. Madden, Hongzi Mao, and V. Nathan. 2019. SageDB: A Learned Database System. In *CIDR*.

[25] Sebastian Kruse and Felix Naumann. 2018. Efficient Discovery of Approximate Dependencies. *Proc. VLDB Endow.* 11, 7 (mar 2018), 759–772. https://doi.org/10.14778/3192965.3192968

[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. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1159–1168. https://doi.org/10.1145/2463676.2463708

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[28] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (July 2019), 1316–1329. https://doi.org/10.14778/3342263.3342270

[29] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy Mattson, and Nesime Tatbul. 2022. Self-Organizing Data Containers. (2022).

[30] Microsoft [n.d.]. Columnstore Indexes Query performance. https://learn.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-query-performance?view=sql-server-ver16. Accessed: 2022-12-01.

[31] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487.

[32] Vikram Nathan, Jialin Ding, Tim Kraska, and Mohammad Alizadeh. 2020. Cortex: Harnessing Correlations to Boost Query Performance. https://doi.org/10.48550/ARXIV.2012.06683

[33] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2017. Slalom: Coasting through Raw Data via Adaptive Partitioning and Indexing. *Proc. VLDB Endow.* 10, 10 (jun 2017), 1106–1117. https://doi.org/10.14778/3115404.3115415

[34] Oracle [n.d.]. Database Data Warehousing Guide. https://docs.oracle.com/database/121/DWHSG/zone_maps.htm#DWHSG-GUID-BEA5ACA1-6718-4948-AB38-1F2C0335FDE4. Accessed: 2022-12-01.

[35] Stefano Paraboschi, Giuseppe Sindoni, Elena Baralis, and Ernest Teniente. 2003. *Materialized Views in Multidimensional Databases*. IGI Global, USA, 222–251.

[36] 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* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 558–569. https://doi.org/10.1145/564691.564757

[37] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[38] Sivaprasad Sudhir, Michael Cafarella, and Samuel Madden. 2022. Replicated Layout for In-Memory Database Systems. *Proc. VLDB Endow.* 15, 4 (apr 2022), 984–997. https://doi.org/10.14778/3503585.3503606

[39] 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* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1115–1126. https://doi.org/10.1145/2588555.2610515

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

[41] Ashish Thusoo, Joydeep Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - A Petabyte Scale Data Warehouse Using Hadoop. *Proceedings - International Conference on Data Engineering*, 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[42] TPC-DS [n.d.]. TPC-DS. https://www.tpc.org/tpcds/. Accessed: 2022-12-01.

[43] TPCH-H [n.d.]. TPC-H. https://www.tpc.org/tpch/. Accessed: 2022-12-01.

[44] 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*

(Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1223–1240. https://doi.org/10.1145/3299869.3319861

[45] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 193–208. https://doi.org/10.1145/3318464.3389770

[46] D. Zilio, C. Zuzarte, S. Lightstone, Wenbin Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, Jarek Gryz, E. Alton, Dongming Liang, and G. Valentin. 2004. Recommending materialized views and indexes with the IBM DB2 design advisor. *International Conference on Autonomic Computing, 2004. Proceedings.* (2004), 180–187.