Replicated Layout for In-Memory Database Systems

Sivaprasad Sudhir
MIT
siva@csail.mit.edu

Michael Cafarella
MIT
michjc@csail.mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

ABSTRACT

Scanning and filtering are the foundations of analytical database systems. Modern DBMSs employ a variety of techniques to partition and layout data to improve the performance of these operations. To accelerate query performance, systems tune data layout to reduce the cost of accessing and processing data. However, these layouts optimize for the average query, and with heterogeneous data access patterns in parts of the data, their performance degrades. To mitigate this, we present CopyRight, a layout-aware partial replication engine that replicates parts of the data differently and lays out each replica differently to maximize the overall query performance. Across a range of real-world query workloads, CopyRight is able to achieve 1.1X to 7.9X faster performance than the best non-replicated layout with 0.25X space overhead. When compared to full table replication with 100% overhead, CopyRight attains the same or up to 5.2X speedup with 25% space overhead.

PVLDB Reference Format:
Sivaprasad Sudhir, Michael Cafarella, and Samuel Madden. Replicated Layout for In-Memory Database Systems. PVLDB, 15(4): 984-997, 2022.
doi:10.14778/3503585.3503606

1 INTRODUCTION

Carefully organizing data on storage is key to achieving high query performance in modern analytical DBMSs. Modern data warehouses use a variety of techniques including horizontal and vertical partitioning, clustered indexes, sort orders, etc. to accelerate scan performance by reducing the cost of accessing and processing data.

Several algorithms, often based on historical properties of a workload of queries run over the data, have recently been proposed to tune the physical layout of tables to optimize performance of the workload [3, 4, 11, 15, 19, 23, 46, 49]. Complex real-world workloads contain queries with varying access patterns including queries that access different attributes, have different selectivity characteristics, and read different subsets of records. For example, dashboard-based interactive data analytics platforms allow users to create query templates with a variety of predicates by zooming in on maps, enforcing ranges on graphs, selecting from drop-down menus etc. Users analyzing a dataset of New York City taxi trips [1] issue queries across a variety of dimensions to find the average fare amount of trips between two parts of New York City where the passengers paid by cash, and to find the number of trips in a particular month of the year that costed more than $5, etc. Existing layout tuning algorithms optimize for the workload on average. However, a single layout may not be optimal for queries of different types. As a result, the benefit of these specialized data layouts decreases as queries with different access patterns scan the same data.

To address these limitations, we built CopyRight, a novel in-memory read-optimized partial replicated layout engine that automatically optimizes the layout for a particular dataset and query workload. CopyRight replicates different parts of the data at varying degrees and organizes each replica under different layouts to maximize the overall query performance. Our layout engine supports sub-table replication where only a subset of columns are materialized in each replica.

Prior work has explored several approaches for automated physical database design in both single node and distributed systems [2, 3, 8, 9]. To the best of our knowledge, CopyRight is the first layout-aware partial replication engine for performance in an in-memory single node setup. Joint optimization of partitioning and replication has primarily been studied in the context of distributed databases [12, 20, 25, 27, 28]. The focus of these approaches is on minimizing the number of cross-partition queries rather than the exact layout of data in each partition.

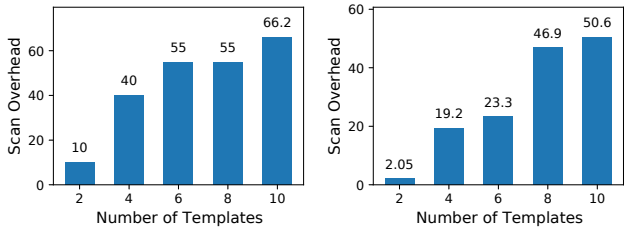
Under storage constraints, CopyRight achieves high performance by focusing replication efforts on parts of the data where it matters. A table-level replication scheme may waste storage by replicating areas that give minimal advantage such as cold parts of the data that is rarely accessed or parts of the data where all queries have similar data access patterns. CopyRight identifies sub-spaces of data that yield large benefit from replication and allocates available storage accordingly to maximize the overall gain. By materializing only a subset of columns in each replica, CopyRight is able to better utilize the available space and create more replicas.

Incorporating partial replication makes the layout design problem much harder as the space of layouts that need to be considered significantly increases. A table can be horizontally partitioned in a large number of ways (the so-called *Bell number*), which grows faster than exponential in the number of tuples. With replication, any subset of these partitionings that fits in the storage bound is a feasible layout, causing a combinatorial explosion in the space of physical design alternatives. With partial replication, a replica may materialize any subset of the tuples and any subset of the columns, further exacerbating the combinatorial nature of the problem.

CopyRight organizes data as a two-level structure: the data space is split into regions and each region is replicated differently. Our layout optimizer employs a number of novel techniques to intelligently prune the space of layouts. Instead of directly searching in the space of layouts, we approach replication as a workload partitioning problem. We partition the workload into clusters and create a replica that is optimal for each cluster. We present a scalable algorithm that jointly optimizes partial replication and layout of each replica by leveraging properties of the data and workload.

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503606

[1] https://www.omnisci.com/demos/taxis



(a) Range partition **(b) Tsunami**

Figure 1: As the number of query templates (different access patterns) in the workload increases, the scan overhead of non-replicated layouts increases, degrading performance.

In summary, this paper makes the following contributions

(1) We present CopyRight, a novel layout-aware partial replication engine that concentrates replication efforts on parts of the data where it matters.
(2) We provide an efficient heuristic algorithm to determine good partially replicated layouts and estimate the cost-space Pareto optimal frontier for a given dataset and workload.
(3) Finally, we demonstrate that CopyRight can outperform other layouts for a variety of real-world datasets and query workloads. CopyRight is 1.1X to 7.9X faster than the best non-replicated layout with 0.25X overhead. In 25% space overhead CopyRight's partial replicated layout is able to achieve the same or up to 5.2X better performance than a table-level replicated layout that incurs 100% overhead .

The paper is organized as follows: we make the case for partial replication in §2, provide an overview of our approach in §3, present the layout structure in §4, describe the optimization algorithm in §5, describe the query workflow in §6, show our experimental evaluation in §7, compare to related work in §8, and conclude in §9.

2 MOTIVATION

We begin with an overview of the impact of heterogeneous query access patterns on the performance of database systems for analytical workloads. We then make the case for layout-aware replication of data focusing the efforts on parts where it matters.

Background: In this work, we focus on multi-dimensional analytical workloads. A table with d numeric columns can be thought of as a d-dimensional space and tuples as points in the d-dimensional space. Queries are equivalent to hyper-rectangles in the space.

With queries that filter data on multiple attributes, DBMSs can use secondary indexes to improve scan performance. However, a secondary index scan works better than a full sequential scan only when the the predicate on the indexed attribute has very low selectivity [22, 41]. An alternative approach is to partition data, organize partitions in some sorted order on one-dimensional storage, and build auxiliary multi-dimensional indexes to skip partitions that don't match query filters. Examples include range partitioning on single or multiple columns, Grid Files [30], k-d tree [6], R-tree [18], Z-order [44], Flood [29], and Tsunami [15]. The partitions and sort order are adapted to reduce the *scan overhead*, the ratio of number of records scanned by the query to the number of records that satisfy the query filter, without incurring large index lookup costs.

Tsunami [15], is a recently proposed learned multi-dimensional layout that has been shown to outperform a variety of other traditional and learned variants. It horizontally partitions the data into non-overlapping regions and organizes each region as a grid. The two-level structure of Tsunami accounts for query skew (i.e, query frequency varies in different parts of data space) by laying out each region differently. The boundaries of the grid and regions are learned based on the data and workload distribution.

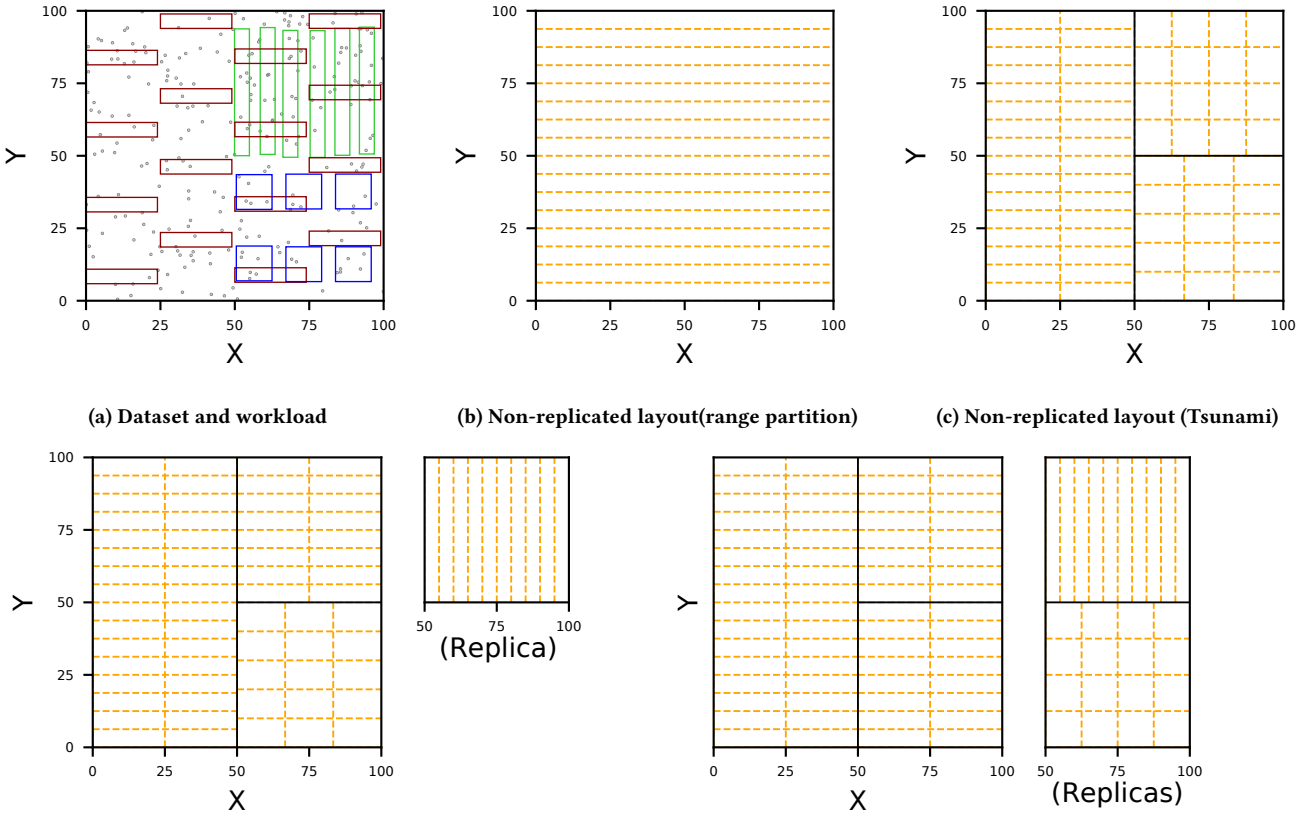
Performance of non-replicated layouts: Non-replicated layouts incur high scan overheads when queries with different access patterns scan the same parts of the data. Fig. 1 shows the scan overhead of the workload with increasing number of query templates in the workload for non-replicated layouts. Data is range-partitioned on the most frequently accessed column in Fig. 1a and organized using Tsunami in Fig. 1b. The table has 8 dimensions with uniformly randomly distributed data. The workload contains equally represented query templates that filter on two randomly selected columns. Queries are uniformly distributed in the data space. Even with a few templates accessing the same data, the scan overhead of the workload is significantly high and queries waste a lot of time scanning unnecessary data. This is a consequence of the *curse of dimensionality* and affects non-replicated layouts across the board.

To illustrate where this overhead comes from in more detail, consider an example of a 2-dimensional dataset and workload shown in Fig. 2a. Here, X and Y axes represent the 2 columns in the table. Data is uniformly randomly distributed, as shown by grey dots. The workload contains 3 types of queries with range predicates on both columns. Dark red queries that are highly selective along Y are uniformly distributed throughout the entire data space, light green queries that are very selective along X in the upper right quarter of the data space, and blue queries with moderate selectivity along both X and Y in the bottom right quarter.

Fig. 2b shows the data range-partitioned on column Y. The orange lines indicate the partition boundaries. While red queries are well-aligned with the partitioning, green ones are not and executing them will require processing many partitions and thereby yield a poor scan overhead. A single layout is efficient for some queries but inefficient for others. This is also applicable to more expressive layouts like Tsunami.

Fig. 2c shows the layout Tsunami finds for the workload. Here, the data space is divided into 3 regions having different query distributions. The black lines indicate the region boundaries. Within each region, the data is further partitioned into cells by constructing a grid. The orange dotted lines indicate the cell boundaries. Tuples within a cell is laid out contiguously on storage. At query time, we find all orange cells that intersect the query and scan all tuples from each intersecting cell. Grid dimensions are partitioned based on the "shape" of the queries, i.e, the selectivity of queries along each dimension. As the right half of the data space has queries with multiple access patterns, Tsunami optimizes its layout for the average query. Data is laid out as a grid that is optimal for the overall workload, but suboptimal for each query type individually. This is shown in the two quadrants in the right half of Fig. 2c.

To reduce the scan overhead of both query types with a single layout, we need to create more fine-grained cells. However, this incurs additional cost for looking up the intersecting cells to scan.



(a) Dataset and workload

(b) Non-replicated layout(range partition)

(c) Non-replicated layout (Tsunami)

(d) CopyRight with 0.25X replication budget

(e) CopyRight with 0.5X replication budget

Figure 2: Fig. 2a shows a multi-dimensional dataset and workload with 2 columns X and Y. The grey dots represent tuples and the rectangles represent queries. Fig. 2b shows the data range-partitioned on column Y. Fig. 2c shows the best layout that Tsunami is able to obtain for the workload. Fig. 2d shows CopyRight's layout with sufficient budget to replicate a quarter of the table. The upper right quarter of the table is replicated. Layout of one of the replicas is optimized for the short and wide red queries. Other one is optimized for the tall and narrow green queries. With enough budget to replicate half the table, CopyRight also replicates the bottom right quarter with one replica optimized for red queries and other for blue queries.

So a fine-grained grid is not a scalable solution for handling heterogeneous access patterns. We can see from the shape of the grid cells, that queries of both types incur a scan overhead. Such effects are exacerbated by workloads with a large number of access patterns, very different access patterns, queries scanning different columns, etc as suggested by Fig. 1.

The case for a partial replicated layout: If we had additional space, we could replicate parts of the data and layout replicas differently to reduce the overall query latency. With sufficient space to replicate a quarter of the data, we can create a replica for the upper right region $X \geq 50$ and $Y \geq 50$. One of the replicas can be optimized for answering red queries. The other one can be optimized for the green queries. This is shown in Fig. 2d. We choose to replicate the upper right quarter over the bottom right as red and green queries have very different access patterns. Allocating the replication budget smartly is important as splitting red and green queries into two replicas gives more advantage than splitting red and blue. If we had sufficient space to replicate half the table, the bottom right quarter can also replicated and each replica can be optimized for its query type as shown in Fig. 2e. This is exactly what CopyRight, our novel replicated layout engine, does.

Real-world workloads include queries with a variety of templates and access patterns. The distribution of these access patterns varies across the data space and so does the benefit from replication. For four real-world datasets, our algorithms found 12 - 20 query templates and 19 - 62 sub-spaces of data with different distributions of access patterns. We take a partial replication approach that concentrates replication efforts in parts of the data that provides maximum reduction in the execution cost. As workloads show patterns in columns being accessed together, we allow sub-table replication, materializing only a subset of columns in a replica, to efficiently utilize the available space by creating multiple smaller replicas.

3 OVERVIEW

As shown in Fig. 3, CopyRight consists of two components: a layout optimizer, for determining how to layout data, and a query executor, responsible for executing queries on this layout. We briefly describe these components below, starting with a description of the structure of data in CopyRight and then describing how our optimization engine automatically selects good layouts for a given workload.

Layout: (§4) CopyRight organizes data as a hierarchy of grids. The data space is hierarchically partitioned into non-overlapping

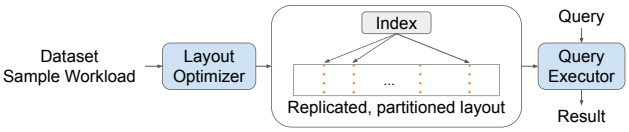


Figure 3: Overview of CopyRight architecture.

regions. In Fig. 2d the black lines indicate the region boundaries. Each region can be replicated any number of times. Each replica is organized as a grid-based layout on storage. A replica may contain only a subset of columns but contains all tuples in the region. In Fig. 2d the orange dotted lines indicate grid boundaries. For the rest of the paper, we use 'region' to denote the unit of replication and 'grid' to denote the layout of each replica of a region. This two-level structure reduces the size of the search space and allows us to efficiently navigate the search space in a top-down fashion by restricting the granularity of replication to horizontal partitions of the table. Throughout this paper, we use horizontal partitioning to denote splitting data into disjoint sets of rows. We use a grid-based layout as it is simple enough to tune efficiently. We use a space partitioning tree to index the regions. The tree index combined with the grid layout is expressive enough to achieve high performance for complex workloads.

Layout Optimizer: (§5) Under storage constraints, we need to concentrate the available replication budget in regions where it matters to get maximum performance from partial replication. Identifying these regions and allocating the replication budget among them is non-trivial as the benefit obtained from replication depends on a large number of factors such as the density of queries, distribution of data, similarity of access patterns, etc. For example, in Fig. 2, with sufficient space to replicate a quarter of the table, we chose to replicate upper right quarter over the bottom right quarter as red and green queries have vastly different access patterns. If the density of blue queries was much higher than green ones, it would have been better to replicate the bottom right quarter. Allocating available replication budget gets harder as the distribution of access patterns becomes more complex.

Given a dataset D, a representative workload W, and a space budget for replication B, CopyRight's layout optimizer finds the data layout that minimizes overall query execution time. The optimizer has to find the regions, allocate available storage for replication amongst them, decide how many replicas to create for each region, what columns to materialize in each replica, and how to layout each replica. All these sub-problems are tightly coupled, making the search process hard. We discuss our heuristic optimization algorithm that addresses these challenges in detail in §5.

Query Executor: (§6) When a query arrives, we find the regions that intersect the query predicates using the tree index. Within each region, we pick the best replica, the one that minimizes the execution time, to execute the query. Within each replica, we use find the grid cells that intersect the query filter, scan the tuples in the cells and emit the tuples that satisfy the filter.

Now, we describe these components of CopyRight in more detail.

4 LAYOUT

In this section we describe CopyRight's layout structure. CopyRight organizes data as a hierarchy of grids using a two-level structure.

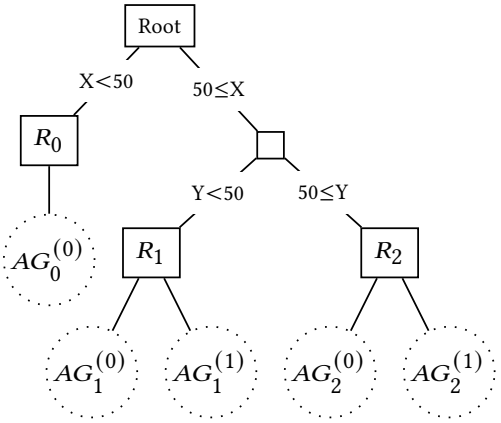
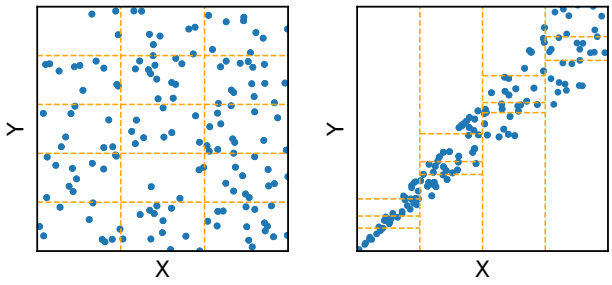


Figure 4: An example Grid Tree.



(a) Dimensions X and Y are partitioned independently

(b) Dimension Y is partitioned based on dimension X

Figure 5: An example Augmented Grid layout.

The top level structure is *Grid Tree*, a hierarchical space partitioning tree. Leaf nodes of the tree correspond to a region and are replicated. Each replica is laid out using *Augmented Grid*. These layouts were introduced in [15]. We extend them to support partial replication.

Grid Tree: CopyRight horizontally partitions the data space into non-overlapping regions. We use a hierarchical space partitioning tree, called Grid Tree to index these regions. The root node of the Grid Tree corresponds to the entire data space. Each node splits the data space into multiple child nodes based on a dimension. The dimension to split and the boundary values are picked based on the workload and dataset. Leaf nodes of the Grid Tree are the regions that are replicated. Fig. 4 shows the Grid Tree index for the replicated layout in Fig. 2e. The root node, the entire data space, is split into 2 child nodes based on dimension X. The node $50 \leq X$ is further split based on dimension Y. Leaf nodes are the regions that are replicated and organized as one or more grids represented by dashed circles. Region R_0 has only a primary replica ($AG_0^{(0)}$) whereas regions R_1 and R_2 are replicated twice ($AG_i^{(0)}, AG_i^{(1)}$).

Augmented Grid: An Augmented Grid is a generalization of a grid that takes advantage of the correlation in data. An Augmented Grid splits the data into p_X partitions along each dimension X of the table. It may split a dimension, say X, into equally sized partitions using $CDF(X)$, cumulative distribution function of X, independent of other dimensions. In Fig. 5a the data is partitioned into 3 columns based on X such that each column has approximately the same number of records. Independently, it is partitioned into 5 rows

along Y. Augmented Grid may also choose to partition a dimension Y dependent on another dimension X, uniformly in $CDF(Y|X)$. For example, in Fig. 5b, each partition along X is further split into cells with equal number of tuples using $CDF(Y|X)$. When combined, these partitions form a grid with $\Pi_{X \in Attrs} p_X$ cells, which are ordered. The tuples within each cell are stored contiguously on storage. The number of partitions along each dimension is decided based on the workload. Soft functional mappings between columns are stored for rewriting query filters during execution to take full advantage of data correlation.

5 LAYOUT OPTIMIZER

The input to our optimization algorithm is a dataset D, a representative workload W, and a replication budget B. The output is the set of regions and an associated Grid Tree index, how each region (leaf node of the Grid Tree) is replicated, and the Augmented Grid layout for each replica. We first give an overview of our heuristic optimization algorithm and then delve into details.

We approach replicating a region as a workload partitioning problem. Queries in the sample workload W that intersect the region are partitioned into clusters. A replica is created for each cluster and is organized optimally for queries in the cluster. To scale this problem, we first group queries with similar access patterns into *types*, each of which becomes a potential target for an optimized replica, and then solve replication as a partitioning of the set of all query types in the workload (§5.1).

Associated with each leaf node of the Grid Tree (region) is a *replication configuration* which describes how the data is replicated. As we explore the space of layouts in a top-down fashion, a block (horizontal partition) may be replicated according to a replication configuration or may further be divided into smaller blocks that are replicated under different replication configurations (§5.2).

Ideally, we want to partition the data space into as few regions as possible such that further dividing these regions into smaller blocks and replicating them under different configurations does not yield any additional benefit. Then, distribute the space budget among these regions and find the best replication configuration for each such region to maximize the overall performance. Finally, create replicas as specified by the configuration and optimize each replica's layout. The tightly coupled nature of the sub-problems along with huge search space make the optimization problem challenging. As shown in Fig. 6, the layout optimization algorithm works in 3 stages.

(1) First, we horizontally partition the data into candidate regions such that they are good candidates for replicating differently and the cost of executing the workload under CopyRight layout can be efficiently estimated for any space budget. Note that a candidate region may be split into regions that are replicated under different replication configurations in the final layout. (§5.3)
(2) For each candidate region, we then estimate the cost-space Pareto optimal frontier. A point (s, c) on this curve corresponds to the CopyRight layout of the candidate region that fits in the space budget s, where c is the cost of executing the workload on that layout. (§5.4)
(3) We use each candidate region's frontier to distribute the replication budget and obtain the layout of each candidate region . The layout of each candidate region combined with the Grid Tree that indexes the candidate regions forms the final CopyRight layout. (§5.5)

5.1 Clustering Queries

We approach replication of a region as a workload partitioning problem. Specifically, each replica is associated with a disjoint subset of the representative workload W that intersects the region. Each replica's layout is optimized for its share of the workload and is used for executing queries that are similar to the ones in its subset. Enumerating all possible subsets of the workload is not scalable. To address this, as a pre-processing step, we cluster the queries into *types* and assign all queries of a type to the same replica. To have minimal impact on the quality of the final layout, queries with access patterns that are likely to benefit from the same optimal layout should be placed in the same cluster.

Augmented Grid partitions the data space into cells having equal number of points along each dimension. So queries having similar selectivity along each dimension are likely to benefit from similar layouts. For example, in Fig. 2, all red queries are likely to benefit from similar Augmented Grid layouts whereas red and green queries have different access patterns and are likely to benefit from different layouts.

Queries with different templates are placed into different clusters. Queries from the same template with different selectivity characteristics should be placed in different clusters. For a d-dimensional table, we featurize a query as d dimensional vector where i^{th} entry in the embedding is its selectivity along dimension i. For each template, we group the queries into types by running a clustering algorithm on this query embedding. We use DBSCAN to cluster the queries into types. The epsilon parameter of DBSCAN can be tuned (0.2 worked well for our experiments). The overall performance was not sensitive to small changes in epsilon. The choice of clustering algorithm is orthogonal to our work. This algorithm is also used in [15] for computing workload skew. For the workload in Fig. 2, if X is dimension 0, and Y is dimension 1, the red queries have embeddings close to [1/4, 1/16], the green queries have embeddings close to [1/12, 1/2], and the blue queries have embeddings close to [1/8, 1/8]. As a result, the clustering algorithm will group all queries of the same color into one type.

5.2 Replication Configuration

A replication configuration describes how any block (horizontal partition) of data is replicated. If a block is organized under a replication configuration, this means that the block is not further horizontally partitioned into smaller partitions that are replicated differently. The final goal of CopyRight's layout optimizer is to find the optimal replication configuration for each *region*. In the search process, it will explore organizing a variety of blocks of data under a variety of replication configurations.

Formally, a replication configuration is a partitioning of query types. $RC = \{\{t_1, t_2\}, \{t_3\}, \{t_4\}\}$ is an example replication configuration for a workload W with 4 types, t_1, t_2, t_3, and t_4. Given a block of data, the replication configuration fully describes how the block is replicated. If the replication configuration has k clusters of query types, then the block has k replicas, one for each cluster. Each replica's layout is optimized for queries of types in this cluster that

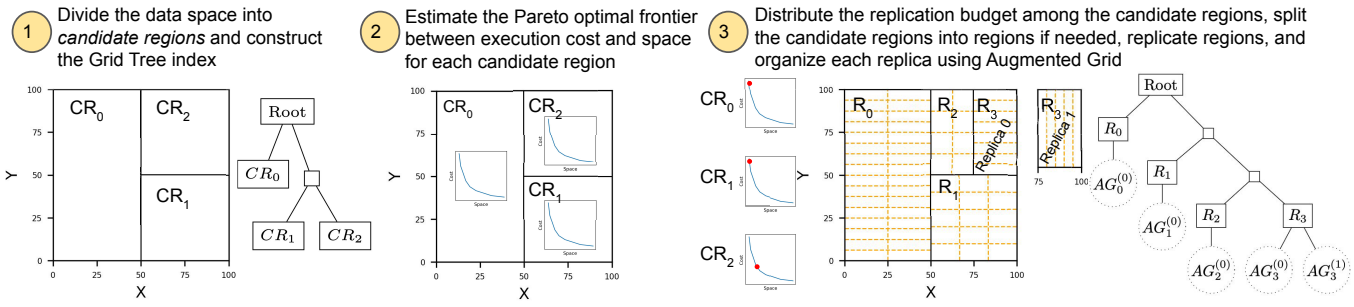


Figure 6: Overview of the layout optimizer with 0.125X replication budget.

intersect the block. For example, under RC, the block is replicated thrice (including the primary replica). The first replica optimizes its layout for queries of type t_1 and t_2 in W that intersect the block. The second and third replicas optimize their layout for queries of types t_3 and t_4 respectively. Each replica contains all tuples in the block. Each replica only materializes the columns that are required for answering the queries in its cluster. One primary replica always exists and materializes all columns.

The size of a block under a replication configuration is the total size required for materializing all replicas of the block specified in the configuration. The cost of executing queries over a block with a replication configuration is the cost of executing each query Q with type t_Q that intersects the block using the replica the configuration specifies for t_Q.

5.3 Candidate Regions

In this section, we describe how we partition the data space into candidate regions that are good candidates for replicating differently. We want candidate regions such that the cost-space Pareto optimal frontier of organizing the candidate region using CopyRight can be efficiently estimated.

The intuition that drives our approach is that two blocks of data are good candidates for replicating differently if they have different distributions of access patterns. Consider three different sub-spaces of data from the example in Fig. 2. The left half $X < 50$ (block 0) has only one type of query. The bottom right quarter $50 \leq X$ and $Y < 50$ (block 1) has two types of queries: red and blue. The upper right quarter $50 \leq X$ and $50 \leq Y$ (block 2) has two types of queries: red and green. As these blocks have different distributions of access patterns, the benefit that can be obtained from replicating vary across these blocks. It makes them good candidates for replicating differently under different storage constraints.

The data space can be split into two blocks with different distributions of access patterns if a query type has different distributions in the two blocks, i.e., if the distribution of a query type is skewed. Our goal is to partition the data space into candidate regions to minimize the overall query skew, i.e, to horizontally split the data space such that the distribution of queries is as close as possible to uniform distribution for each type. We take a top-down approach hierarchically partitioning the data space so as to construct a tree index to efficiently index the candidate regions. We use the skew-tree algorithm from [15] to divide the space into "uniform" candidate regions and construct an associated Grid Tree. At a high level, the algorithm recursively divides a Grid Tree node into child nodes by picking a dimension and values to split the node that minimizes the total query skew. For the workload in Fig. 2, the algorithm will split the data space into 3 candidate regions indicated by the black partitions in Fig. 2c.

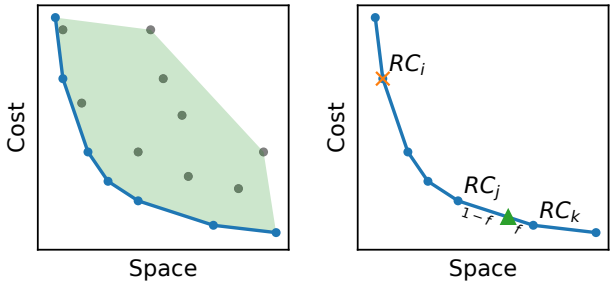
It is possible that two such candidate regions are replicated identically under same storage budget. It's also possible that a candidate region needs to be further split in the final layout. For example, in Fig. 2, if the storage budget for replication was an eighth of the size of the table, then the upper right quarter region could be further split into two so that one of the children could be replicated twice. How to split such a candidate region depends on the space available for replicating that candidate region in the final layout. This cannot be solved independently for each candidate region.

The benefit of uniform regions: To estimate the cost-space Pareto frontier, we need to consider splitting a candidate region in all possible ways and replicating them under different replication configurations. We make an observation about uniform candidate regions and leverage that to efficiently estimate the cost of dividing the candidate region further and replicating them differently.

Consider splitting a uniform candidate region into two blocks along some dimension, one containing a fraction f of the data and the other containing the remaining $1-f$. Let the block with fraction f of the data be replicated under configuration RC_1. Let c_1 be the cost and s_1 be the space requirement for replicating the entire candidate region under RC_1. Let the block with fraction $1-f$ of the data be replicated under configuration RC_2 that has cost c_2 and space requirement s_2 for the entire candidate region. If the distribution of each query type in the candidate region was truly uniform, by the symmetry of the uniform distribution, the cost of the block with fraction f under RC_1 is $f*c_1$. Similarly, the cost of the block with fraction $1-f$ under RC_2 is $(1-f)*c_1$. Due to uniformity, the cost of the split configuration is: $f*c_1 + (1-f)*c_2$. Similarly, the space requirement for the setup is: $f*s_1 + (1-f)*s_2$. This can be extended to any number of splits, with the cost of further dividing a uniform candidate region into any number of blocks and replicating them under different configurations approximated by an affine combination of the cost of the corresponding configurations for the entire candidate region. We now describe how this can be used to efficiently estimate the cost-space Pareto optimal frontier.

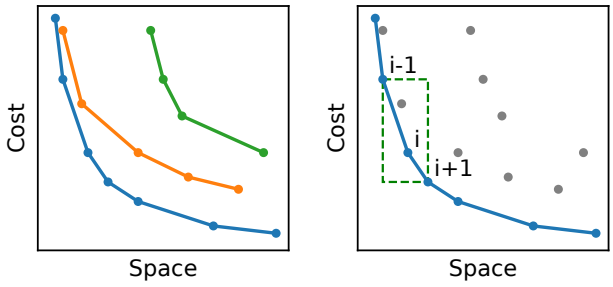
5.4 Finding the Pareto Optimal Frontier

Now we describe how we estimate the cost-space Pareto optimal curve of laying out a candidate region under CopyRight. This is done independently for each candidate region. We use this curve to find the benefit from allocating a certain amount of space to replicating a candidate region while distributing the available space among candidate regions (§5.5). For the rest of the paper, we assume that the space available for replication is on the X axis and the cost of execution is on the Y axis. If (s, c) is a point on this curve, c is the



(a) Feasible space **(b) Pareto optimal curve**

Figure 7: Each dot in Fig. 7a represents replicating a candidate region under a replication configuration. Shaded region represents partially replicated layouts. Blue lines form the Pareto frontier. In Fig. 7b, the point × on the frontier corresponds to replication configuration RC_i. The point ▲ on an edge corresponds to splitting the candidate region and replicating fraction f under RC_j and $1-f$ under RC_k.



(a) Non-dominated sorting **(b) Crowding distance**

Figure 8: Pareto Frontier estimation.

cost of executing the workload on the candidate region organized using the optimal CopyRight layout that fits in the space budget s.

A candidate region may be replicated under a replication configuration. The dots in Fig. 7a represent the cost-space pair for all possible replication configurations. A candidate region may also be split into smaller blocks that are replicated under different configurations. As discussed earlier, its cost/space can be approximated by an affine combination of the cost/space of replicating the entire candidate region under the replication configurations. The feasible space of all possible cost-space tuples is a convex polygonal hull as shown by the shaded region in Fig. 7a. The blue edges of the polygon form the convex piece-wise linear Pareto optimal frontier.

Each vertex on the frontier corresponds to replicating the entire candidate region under a replication configuration. A point on an edge of the curve corresponds to dividing the candidate region into two blocks and replicating them under configurations of the endpoints of the edge. For example, in Fig. 7b the orange cross corresponds to replicating the candidate region under RC_i and the green triangle corresponds to splitting the space and replicating fraction f under RC_j and $1-f$ under RC_k. This reduces the complexity of our algorithm as we can search in the space of replication configurations to find ones on the Pareto optimal front instead of searching in the space of all possible partial replicated layouts.

Algorithm: We propose a greedy multi-objective optimization algorithm to find the configurations on the Pareto optimal frontier. We start at the rightmost point on the Pareto curve, the fully replicated configuration, i.e., the configuration with a separate replica for each query type. For a workload W with 3 types, this corresponds to $RC = \{\{t_1\}, \{t_2\}, \{t_3\}\}$. We iteratively generate and explore other replication configurations. While exploring the space, we maintain the Pareto optimal frontier of all configurations seen so far. This can be found by the finding configurations that are not dominated by any other partial replicated layout. Note that dominance is different from traditional multi-objective optimization problems in our setup. As any point on the line segment connecting 2 configurations is a feasible layout, they need to be considered while checking dominance. A point is non-dominated if there is no line segment connecting two configurations that dominates it. The iterative algorithm works as follows.

(1) Generate offspring configurations from the current generation of configurations.
(2) Evaluate the cost and space of all offspring configurations.
(3) Select k configurations from the offspring as parents for the next generation.
(4) Update the Pareto optimal frontier.
(5) Repeat steps 1-4 until the termination condition is met.

In step 1, we generate an offspring configuration by merging 2 replicas in a parent configuration. A parent $RC = \{\{t_1\}, \{t_2\}, \{t_3\}\}$ will generate $\{\{t_1, t_2\}, \{t_3\}\}$, $\{\{t_1\}, \{t_2, t_3\}\}$, and $\{\{t_1, t_3\}, \{t_2\}\}$ as offspring. All offspring configurations have one less replica than the parent configuration.

The offspring can have varying sizes and costs which is computed in step 2. The space requirement of a merged configuration can be easily computed as the space required to materialize the relevant columns. Computing the cost of a merged replication configuration is time-intensive as it involves optimizing the layout of each replica in the configuration. We use an approximation as described later.

In step 3, we select k configurations from the offspring as parents for the next iteration. k is a tunable parameter. Picking configurations with the lowest cost or space can be suboptimal. In this multi-objective setting, we pick ones that push the Pareto frontier forward while maintaining diversity on the frontier by sorting them based on non-domination rank and crowding distance, as in [14].

We first compute the non-domination rank of all offspring configurations by organizing them into a series of non-intersecting non-dominated frontiers. For example, in Fig. 8a the configurations are organized into three frontiers. The non-dominated blue frontier is assigned rank 0. The orange frontier is dominated by the one other frontier (blue), so it gets rank 1. Green is assigned rank 2. Smaller ranks correspond to better configurations.

To have a more uniformly spread-out Pareto frontier, we prefer configurations from less crowded parts. We calculate the crowding distance of a point as the average distance of two points on either side of this point along each of the objectives on its non-domination frontier as shown in Fig. 8b. This can be thought of as an estimate of the density of configurations surrounding a particular point.

We first sort the configurations in the increasing order of the non-domination rank. Within the same non-domination rank, we sort the configurations in the decreasing order of crowding distance. We then pick the top-k configurations as parents for the next iteration.

We continue iterating until the only remaining configuration has a single replica that is responsible for queries of all types.

We do not discard all dominated configurations as they can eventually lead to better solutions. Consider the following configurations for a workload with four types. $RC_1 = \{\{t_1, t_2\}, \{t_3\}, \{t_4\}\}$, $RC_2 = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$, and $RC_3 = \{\{t_2, t_3\}, \{t_1, t_4\}\}$. Types t_3 and t_4 are more popular in the workload. RC_1 will likely have lower cost than RC_2 as the popular query types get their own replicas. So RC_1 can dominate RC_2. RC_3 can be the optimal configuration under some storage budget but will not be considered if we discard RC_2.

Cost Evaluation: Each replica is optimized for a disjoint subset of the workload. The cost of a replica is the cost of executing these queries on the grid layout optimized for them. The cost of a configuration is the sum of costs of each replica. Computing this involves optimizing the layout of each replica. We use a gradient descent style iterative algorithm from [15] to optimize the Augmented Grid layout parameters. Only the relevant columns of data and its share of the workload are used for optimizing the layout of a replica. The cost of executing a query is estimated using a cost model

$$cost = w_0(\#cell\ ranges) + w_1(\#scanned\ pts)(\#scanned\ dims)$$

A cell range is a set of adjacent cells in physical storage. $\#cell\ ranges$ is the number of such ranges that are scanned. $\#scanned\ pts$ is the number of tuples that are scanned and $\#scanned\ dims$ is the number of columns that are accessed by the query. w_0 and w_1 are weights that are learned. The first term in the cost model accounts for random accesses and the second term for sequential accesses. The features of this cost model can be efficiently estimated from the grid parameters and the number of tuples per grid cell. The latter is estimated from a sample of the data.

To avoid too many re-optimizations of the grid layout while merging replicas, we use a heuristic to approximate the cost of the merged replica. Let $c_{i,j}$ be the cost of executing the queries of replica i on the layout of replica j. The cost of the replication configuration is $\Sigma_{i \in replicas} c_{i,i}$. The cost of merging replicas in R into a new replica, m, is approximated as

$$c_{i,m} = \frac{\Sigma_{j \in R} c_{j,j} * c_{i,j}}{\Sigma_{j \in R} c_{j,j}} \qquad \forall i \neq m \wedge i \notin R \qquad (1)$$

$$c_{m,i} = \Sigma_{j \in R} c_{j,i} \qquad \forall i \neq m \wedge i \notin R \qquad (2)$$

$$c_{m,m} = \frac{\Sigma_{i,j \in R} c_{j,j} * c_{i,j}}{\Sigma_{j \in R} c_{j,j}} \qquad (3)$$

Eq. 1 estimates the cost of evaluating the workload of replica i on the merged replica m as a weighted average of the cost of executing queries of i on the layout of each replica in R. The intuition is that when we merge two workloads containing queries of different types the layout is optimized to be good on average. The layout will be biased in favor of the expensive set of queries. We use $c_{j,j}$ as the weight to mimic that. Eq. 2 is the cost of evaluating the workload of m on the layout of i. As the workload of m is the union of the workloads of all replicas in R, the cost is the sum of cost of individual workloads. Eq. 3 is a combination of Eq. 1 and Eq. 2.

We found that the above approximation worked well in our experiments. We tried other heuristics such as estimating the merged layout (instead of the cost) by taking a weighted average of the layout parameters (number of partitions along each dimension), merging the queries with the most similar access patterns, etc., but found they did not work as well for the datasets that we tested.

For the starting configuration with a replica for each query type, we train Augmented Grid for each replica and compute $c_{i,j}$ for all i, j. For other configurations explored, we approximate the cost as described above. The cost estimate gets stale as more nodes are merged into the same replica. The costs are recomputed by optimizing the layout when the number of merges exceeds a threshold.

5.5 Space Allocation

We now discuss how to allocate the space budget available for replication among the candidate regions and determine the final layout. Our objective is to find the operating point on each candidate region's cost-space Pareto frontier that minimizes the total cost of execution with the total size under the available budget.

The total cost of execution is the sum of costs and the total size of the layout is the sum of sizes of each candidate region. Each Pareto frontier is piece-wise linear and convex. So we can start at the left-most point on the Pareto curve in each candidate region and descend down the steepest edge of the curve until the replication budget is exhausted to find the optimal space allocation.

We index the points on the Pareto optimal frontier from left to right for all candidate regions. (s_0, c_0) is the leftmost point corresponding to the replication configuration with one replica for all query types. On each candidate region's Pareto frontier, we may move along the curve by taking a *full step* from (s_i, c_i) to (s_{i+1}, c_{i+1}). This consumes $(s_{i+1} - s_i)$ space from the available budget. The benefit per unit of space this step is $(c_i - c_{i+1})/(s_{i+1} - s_i)$. If there is not enough space to take a full step, then we may take a *partial step* from (s_i, c_i) to $(f * s_i + (1-f) * s_{i+1}, f * c_i + (1-f) * c_{i+1})$ where $0 < f < 1$, exhausting the budget. The benefit per unit of space is the same as a full step.

Our iterative algorithm works as follows: We initialize the operating point to (s_0, c_0) for each candidate region.

(1) Pick the candidate region with the highest benefit per unit of space for the next step.
(2) Take a full step on the candidate region's Pareto curve if there is sufficient remaining budget. Otherwise, take a partial step.
(3) Iterate until space is exhausted or every candidate region is fully replicated.

After finding the optimal space allocation, if the operating point on the Pareto curve is on a vertex, then the candidate region is replicated using the configuration of that vertex. It's possible that the operating point is on an edge, in which case the candidate region is split into two and replicated under the configurations of the end points of the edge. A fraction of the data is replicated using the configuration of one end point and remaining fraction using the configuration of the other end point to fully utilize the available space budget. The Grid Tree node corresponding to the candidate region is split into two. As the candidates regions are nearly uniform as defined in section §5.3, we can choose an arbitrary dimension to split the node. For example, with enough storage budget to replicate an eighth of the table, we will split the Grid Tree node corresponding to the upper right quarter into two and replicate one of the child nodes as in Fig. 6.

At this point, we have the final Grid Tree and the replication configuration for each leaf node. If the layout for any replica in the

final configuration was not already computed during the Pareto frontier estimation (because we approximated its cost or split the Grid Tree node after space allocation), we train the layout for each replica independently using its subset of the workload.

6 QUERY EXECUTION

When a query arrives, we first identify the type of the query. We featurize the query into the same embedding used at training time for clustering queries. We classify the query by finding its nearest neighbor among the embeddings of the centroid of the clusters. We find the regions that intersect the query predicates using the Grid Tree index. At each leaf node we pick the best replica for executing the query. If the replica corresponding to the query type has all columns required to answer the query, we pick that. It's possible that the replica does not have all columns required for executing the query because of misclassification of query type or if the query template was unseen during training. In that case, the query is answered from the primary replica. For each replica, we find all cells in the Augmented Grid that intersect the query filter. For each intersecting cell, we identify the corresponding range in physical storage using a lookup table. Finally, we scan all records within the range and emit the tuples that satisfy all query filters. The overhead of picking the replica to execute the query is minimal and does not offset the benefit from replication.

7 EVALUATION

In this section, we present an experimental evaluation of CopyRight. Our experiments show that

(1) Layout-aware replication improves query performance.
(2) Partial and sub-table replication gives significant speedup over full table replication.
(3) CopyRight is consistently superior to simpler alternatives.
(4) The performance advantage of CopyRight scales with dataset and workload complexity.

7.1 Experimental Setup

We implemented CopyRight as a custom in-memory column store in C++. All attributes are stored as 64-bit integers. Any string and categorical values are dictionary encoded. We limit floating point values to a fixed number of decimal points and scale them by the smallest power of 10 that converts them to integers, as in [29]. Our implementation has a scan-time optimization: if the range of data being scanned is exact, i.e, all elements in the range satisfy the query filter, we skip applying predicates and scan only the columns that are required by downstream aggregation operators.

We performed all experiments on an Ubuntu Linux machine with Intel(R) Xeon(R) Platinum 8275CL 3.00GHz CPU with 192 GB RAM. All experiments are single-threaded. We compare CopyRight to the following baselines implemented on the same column store.

No Replication (No Rep): We use Tsunami [15] as a layout-aware baseline that does not replicate any data.

Full Table Replication (FTR): We implemented a simple layout-aware replication scheme that replicates data at the granularity of tables. For a d-dimensional table, we embed the queries as a $2d$-dimensional feature vector. The first d entries encode the selectivity of the query along each dimension. The remaining d entries are binary variables corresponding to the columns in the table to encode the query template. Variables corresponding to the columns accessed in a query are set to 1. If there is sufficient budget to replicate the entire table k times, we split the workload into k clusters using k-means clustering of the query embeddings. Each replica is responsible for queries in one of these clusters and is laid out using Tsunami, trained independently for each replica using data in the entire table and queries in its cluster. At execution time, we find the replica to run the query on by identifying the cluster with the most similar embedding.

Replicated k-d Tree (RKT): We split the space using a k-d tree [6] for identifying candidate regions for replication instead of constructing a Grid Tree using the algorithm in §5.3. The k-d tree recursively partitions space using the median value along each dimension, until the number of tuples in each leaf falls below a threshold. A threshold of 1% of the number tuples in the table worked well for the four real-world datasets that we tested on. The dimensions are selected in a round robin fashion, in the order of selectivity. Each leaf node of the k-d tree is replicated as in CopyRight. We use the same algorithm to solve for the replication configuration and corresponding layouts.

Horizontal Only (HO): We implemented a fine-grained replicated layout without sub-table replication. HO is same as CopyRight except that all columns in the table are materialized in every replica.

Query Based (QB): This baseline implements layout-aware partial replication including sub-table replication similar to CopyRight. But, it uses similarity between query templates as a proxy cost function for merging replicas in the algorithm described in §5.4. We use the cosine similarity between $2d$-dimensional query embedding discussed above. Prior works on layout design have used distance between the binary encoding of query templates (the last d entries of our embedding) for clustering queries [4, 40].

We note that the latter three baselines are variants of CopyRight and use various aspects of the system. We discuss them in detail in the ablation study in §7.3. We do not compare against other non-learned layouts as Tsunami was shown to be superior over them [15]. We also don't compare against other learned multi-dimensional layouts as they do not adapt based on query distribution [13, 36, 48] or are optimized for disk [26, 49].

7.2 Datasets and Workloads

We evaluate the layout schemes using four real-world datasets and query workload traces from a dashboard-based interactive data analytics platform at OmniSci [32]. The dashboard has a variety of interactive knobs to change the templates and parameters of a query. This includes zooming in on maps, enforcing ranges on graphs, selecting from drop-down menus etc. These knobs vary across the datasets. The workload contains scan-oriented queries with multidimensional range and equality predicates. We dropped columns that are not used in any queries from all datasets.

For each dataset D, we use 2 versions of the workload, D-F and D-M, in our experiments. D-F contains the full workload. D-M is a modified workload where we dropped queries with selectivity over 10%. When queries scan a large fraction of the data, non-replicated layouts can achieve low scan overheads without incurring high look-up costs. High selectivity can thus delay the curse of dimensionality and the benefit from replication is sometimes lower. The

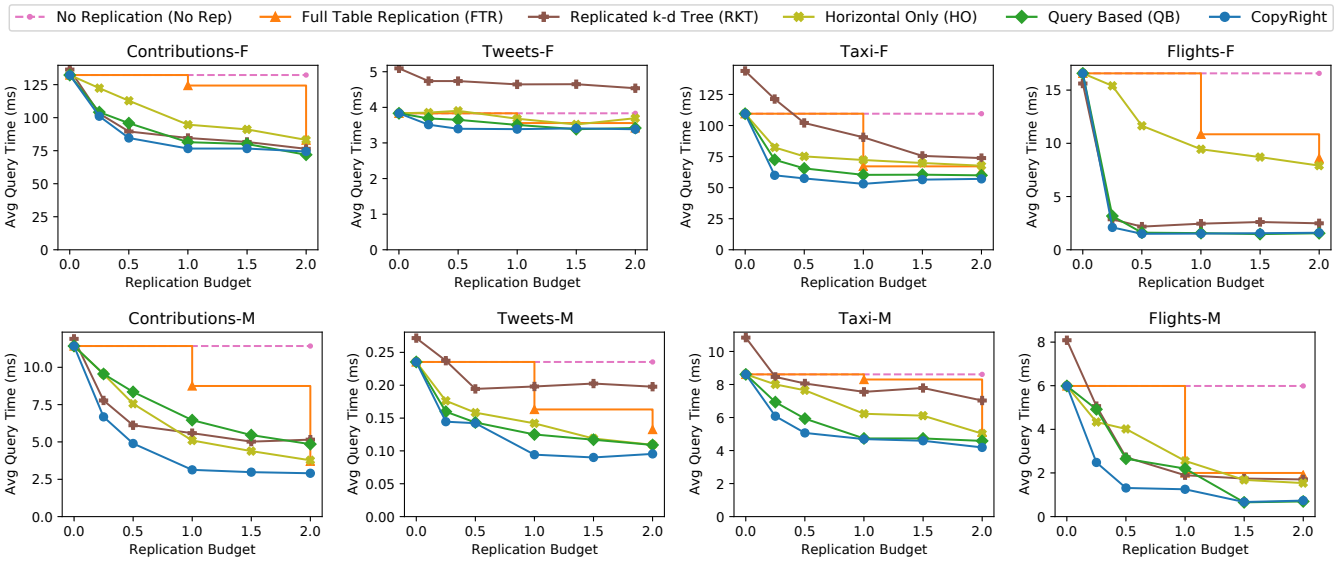


Figure 9: Performance of layouts at various replication budgets.

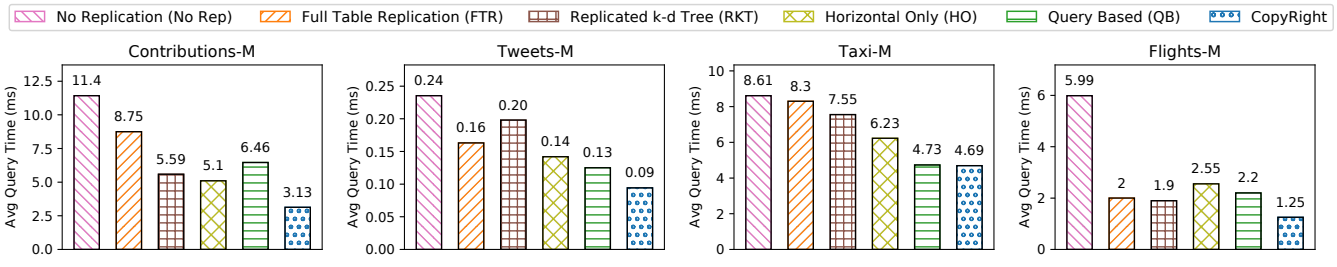
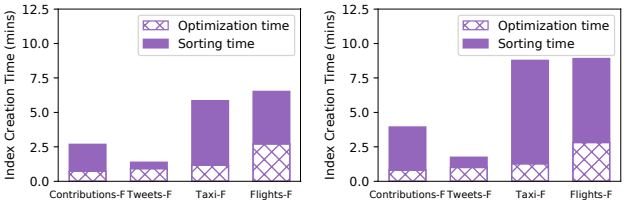


Figure 10: Performance of layouts at replication budget 1X.



(a) Replication Budget: 0.25X **(b) Replication Budget: 1X**

Figure 11: Index creation time across workloads.

Table 1: Characteristics of the datasets and workloads. Properties of the modified workload in parenthesis.

Dataset	Contrib.	Tweets	Taxi	Flights
#tuples	86M	15M	175M	120M
#dimensions	9	16	13	21
#templates	13(13)	14(6)	20(15)	12(9)
#types	34(27)	16(8)	31(25)	18(13)
#regions	19(31)	46(28)	62(65)	46(57)
selectivity(%)	25.89(1.45)	14.36(0.08)	26.05(0.23)	15.97(0.38)

workload traces we used are from OmniSci, a GPU based DBMS that is heavily optimized for scans, and many of their workloads include a number of large scans that we believe are not particularly representative of other analytics workloads. Prior work on analytics and partitioning from researchers with access to corporate workloads confirm that many real-world workload traces have low overall scan selectivity, e.g., 0.0005% and 0.0697% from Microsoft [49] and 0.3% from Conviva [47]. We use the modified workload to demonstrate the results on workloads with moderate to low selectivity. The characteristics of the datasets and workloads is summarized in Table 1. We now describe them in detail.

The **Contributions** dataset contains data about 25 years of political donations in the US. Columns include contribution amount, coordinates of contributor's address, recipient details, etc.

The **Tweets** dataset contains details about tweets from November 2014. Columns in the table include date and time of tweets, latitide and longitude of the location, language, country, etc. We use one month's worth of data from November 2014.

The **Taxi** dataset has records of yellow taxi trips in New York City in 2014. Columns in the table include pickup and drop-off times, pickup and drop-off coordinates, trip amount, tip, etc.

The **Flights** dataset has historic flight data in the US for 3 decades. Columns include date of the trip, day of the month, day of the week, source and destination coordinates, etc. The full Flights workload has a large fraction of `COUNT(*)` queries that benefit significantly from partitions that are subsumed by the query filters.

To study the behavior of algorithms by varying dataset parameters, we use synthetic datasets that are discussed in §7.4.

7.3 Overall Results

Fig. 9 shows the performance of various layouts at different replication budgets. The replication budget is expressed as a fraction of the table size. A replication budget of 0 means there is no space for replicating any tuple. A replication budget of 2 means that there is sufficient space to create two additional full replicas of the table. Fig. 10 highlights the performance at space budget = 1X.

CopyRight is consistently better than all baselines at all budget points across datasets and workloads. At 0.25X space overhead, CopyRight attains 1.1X to 7.9X faster performance than non-replicated layout and the same or up to 5.2X faster performance than full table replication on the raw workloads. On low selectivity workloads, with 0.25X replication budget, CopyRight is 1.4X to 2.4X faster than No Rep. Compared to FTR with 100% overhead, CopyRight achieves 80% of the performance to 1.4X faster performance with just 25% space overhead. With 1X overhead, our replication engine attains 1.8X to 4.8X speedup over No Rep and 1.6X to 2.8X speedup over FTR for low selectivity workloads. At very high or very low replication budgets, the relative advantage of CopyRight over other baselines is low. At high budgets, with sufficient space to create a separate replica for frequent or expensive query types, finding a good layout is easy and at low budgets, there is limited opportunity to improve. While CopyRight achieves the same or up to 2X faster performance than the next nearest baseline, no single baseline dominates others across all datasets. FTR, HO, RKT, QB are at least 7.1X, 7.8X, 2.1X, and 2.1X slower than CopyRight at some replication budget in at least one of the eight workloads.

Ablation Study: Layout-aware replication boosts query performance compared to non-replicated layouts as all replicated layouts achieve significant speedup over No Rep. FTR is up to 3X faster than No Rep with 1X additional space requirements. CopyRight's performance advantage comes from focusing replication efforts in parts of the data that gives maximum benefit. By focusing replication efforts in sub-spaces of the data, HO, CopyRight without sub-table replication, attains up to 1.7X better performance than FTR at 1X replication budget. With sub-table replication, CopyRight is able to better utilize the replication budget by materializing a subset of columns in each replica, achieving up to 7.7X faster performance than HO. By being layout-aware while grouping query types to replicas, our engine achieves up to 2.1X better speedup than QB which groups queries based on similarity of query templates. CopyRight performs up to 2.6X better than RKT, by identifying the correct granularity of replication and focusing efforts there.

Optimization Costs: CopyRight's layout optimizer took under 3 minutes to find the layout for all four datasets. Data reorganization completed in less than 7.5 minutes with 1X overhead. Fig. 11 shows the layout optimization and data reorganization costs. The layout optimizer has a peak memory usage of 225.08MB for Contributions, 76.86MB for Tweets, 436.45MB for Taxi, and 332.78MB for Flights. This includes the 1% sample of the data that the optimizer uses.

7.4 Scalability

To demonstrate the scalability of our approach we compare CopyRight against all baselines by varying different parameters of the workload. We use a synthetic dataset with 8 columns. Each column is independently uniformly distributed. All queries use 2 columns for predicates and perform a `COUNT` aggregation. The filter columns in each template are randomly generated. Unless specified otherwise, we use a dataset with 100M tuples and a workload with 8 equally represented query templates uniformly distributed in the data space. In each experiment, we re-optimize CopyRight and baselines for each dataset/workload configuration.

Selectivity: We vary the selectivity of the workload from 0.001% to 10%. Fig. 12a shows that performance of CopyRight scales with selectivity. The relative advantage of CopyRight over other layouts decreases at higher selectivities as all layouts incur low scan overhead with large query rectangles.

Number of templates: We vary the number of templates from 2 to 20. The average selectivity of the workload is 0.1%. The benefit from replication initially increases, as we can create replicas optimized for each template. Further increase in the number of templates eventually decreases the relative benefit, as more templates get assigned to each replica. Fig. 12b shows that CopyRight is able to delay the curse of dimensionality by replicating data.

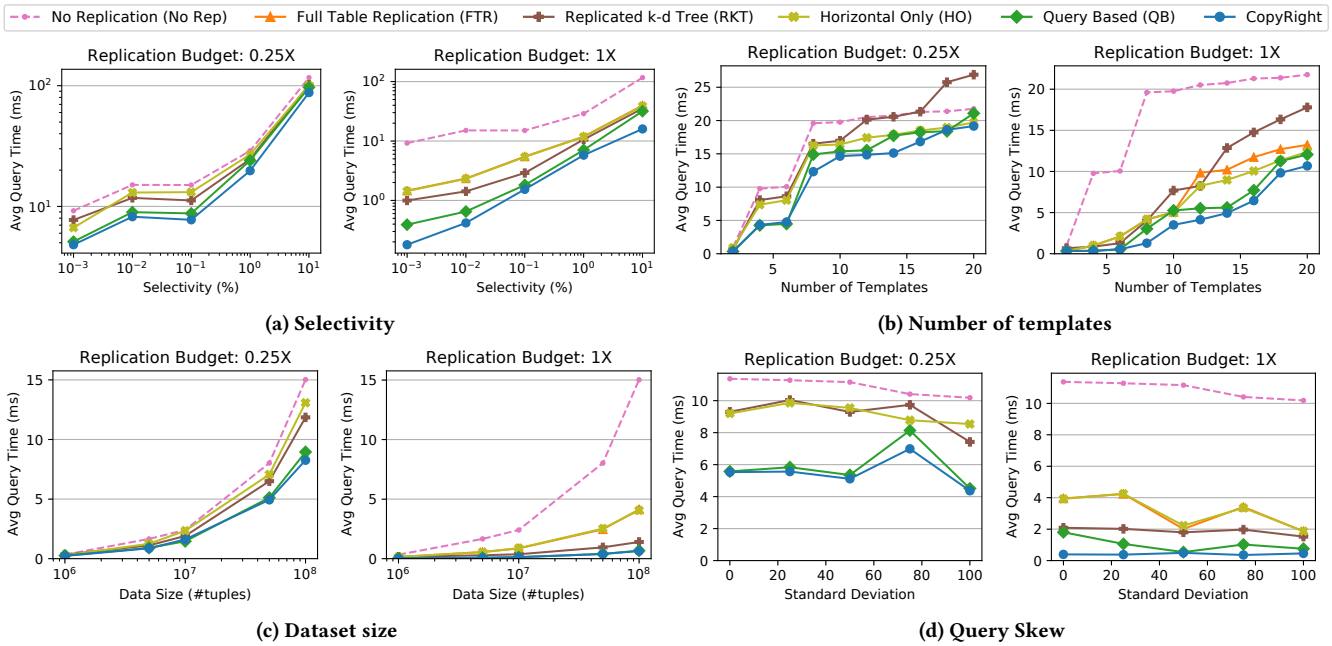
Dataset size: We vary the number of tuples in the table from 1M to 100M. The average selectivity of the workload is 0.01%. Fig. 12c shows that CopyRight maintains its performance benefit across datasets with different sizes.

Query skew: We vary the distribution of templates in the workload. The number of queries of each type is sampled from a normal distribution with mean 200. We vary the standard deviation from 0 to 100. The average selectivity of the workload is 0.01%. Each point on the x-axis in Fig. 12d corresponds to a workload with a skewed distribution of queries. The performance gain depends on the similarity of query templates and their relative distributions. CopyRight maintains its performance advantage across workloads with varying query skew.

Optimization Time: We study the impact of varying data and workload complexity on the index creation time. We vary the number of templates in the workload from 2 - 10 and use datasets with 1M to 100M tuples. The average selectivity of the workload is 0.1%. Fig. 13 shows that the index creation time increases slowly with number of query templates and fast with dataset size as the data sorting time dominates layout optimization time.

8 RELATED WORK

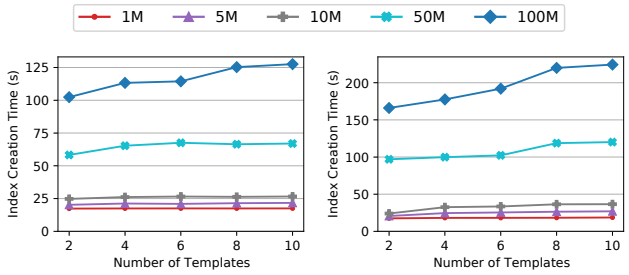
Layouts: DBMSs often use horizontal, vertical and hybrid partitioning to reduce the amount of data accessed during query processing [3–5, 10, 16, 19, 38, 46]. To accelerate the performance of multi-dimensional queries, data is often laid out as multi-dimensional clustered indexes [6, 18, 30, 44] or using specialized sort order [31, 44]. See [33, 42] for a survey. Kraska et al. introduced the idea of learned indexes [24] and recent work has extended it to multi-dimensional layouts. The most relevant to our work is Flood [29] and Tsunami [15], layouts optimized for in-memory databases. Tsunami, described in this paper, builds on top of Flood. [17, 26, 49] learn layouts for disk-based systems. [13, 36, 48] adapt layout based on data distribution but not query distribution. Array DBMSs organize large arrays by partitioning them into sub-arrays called chunks or tiles. Prior work has studied various strategies for chunking [34, 39, 45, 50]. Chunking is also studied in multi-dimensional databases for computing data cubes for accelerating aggregation queries [21, 51]. CopyRight and these works share the idea of optimizing layouts by learning from data and query distributions.



(a) Selectivity (b) Number of templates

(c) Dataset size (d) Query Skew

Figure 12: CopyRight's performance across workloads with varying dataset and workload parameters.



(a) Replication Budget: 0.25X (b) Replication Budget: 1X

Figure 13: Index creation time with workload complexity.

None of these works consider partial or full replication of data while optimizing their layouts. [40] proposes maintaining copies of multi-dimensional arrays, each chunked differently, but does not consider partial replication. Qd-tree [49] constructs a decision tree to partition data to reduce disk-based block accesses. They briefly discuss the possibility of incorporating table-level replication by simultaneously learning 2 trees, but does not give any details.

Replication: Joint optimization of partitioning and replication has primarily been studied in the context of distributed databases. These approaches focus on minimizing the number of cross-partition queries and are agnostic of the exact layout of data in each partition. [20, 27] only supports horizontally splitting a table into a fixed number of shards (number of nodes in the cluster) based on one attribute and fully replicating a table at nodes. Schism [12] takes a graph partitioning approach and NashDB[28] uses an economic model based approach to handle fine-grained replication, but their methods are tightly coupled to the cost model (number of cross-partition queries) and cannot be easily extended to a layout-dependent cost model. These systems assume that all replicas are equally good for answering a query. [25] supports partitioning each replica by a different key but only at table level. The complexity of layouts and performance that can be gained is limited by the fault tolerance and recovery requirements in distributed databases [43]. Our work is for in-memory single node setup and focuses only on performance. Ramamurthy et al. makes the case for employing different storage models (column-store/row-store) in different full replicas of the table [37]. Our work goes much beyond this in replicating parts of the data and considering more complex layouts for each 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, 8, 9, 35]. Arguably, each partial replica of a table can be expressed as a materialized view. However, materialized view selection algorithms explore the space of syntactically relevant views, focusing on creating logical views that cover queries, whereas we focus on replicating tuples themselves and on optimizing the physical layout of replicated regions [2, 7, 52]. CopyRight can be used to partially replicate materialized views. The most relevant to our work is integrating horizontal and vertical partitioning with index and materialized view selection [3] which explores a large space of layouts like CopyRight. They consider only single-column range and hash partitioning.

9 CONCLUSION

We designed and implemented CopyRight, a layout-aware partial replication engine that replicates parts of the data that matters to maximize overall query performance. CopyRight achieves 1.1X to 7.9X speedup over the best non-replicated layout with 0.25X overhead across a range of real-world workloads. At 25% space overhead CopyRight attains the same or up to 5.2X better performance than full replication of the table that has 100% space overhead.

ACKNOWLEDGMENTS

This work was supported by the MIT Data Systems and AI Lab (DSAIL) and NSF Convergence Accelerator Award 2132318.

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] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.

[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] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[7] 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

[8] 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.

[9] 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.

[10] 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

[11] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. 2009. The Case for RodentStore, an Adaptive, Declarative Storage System. *CoRR* abs/0909.1779 (2009). arXiv:0909.1779 http://arxiv.org/abs/0909.1779

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

[13] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. [n.d.]. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[15] 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 (Oct. 2020), 74–86. https://doi.org/10.14778/3425879.3425880

[16] 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.

[17] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2021. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. arXiv:2103.04541 [cs.DB]

[18] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[19] Richard A. Hankins and Jignesh M. Patel. 2003. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 417–428.

[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] Nikos Karayannidis and Timos Sellis. 2003. SISYPHUS: The implementation of a chunk-based storage manager for OLAP data cubes. *Data & Knowledge Engineering* 45 (05 2003), 155–180. https://doi.org/10.1016/S0169-023X(02)00178-7

[22] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 715–730. https://doi.org/10.1145/3035918.3064049

[23] 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*.

[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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[25] Juchang Lee, Kyu Hwan Kim, Hyejeong Lee, Mihnea Andrei, Seongyun Ko, Friedrich Keller, and Wook-Shin Han. 2020. Asymmetric-Partition Replication for Highly Scalable Distributed Transaction Processing in Practice. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3112–3124. https://doi.org/10.14778/3415478.3415538

[26] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119–2133. https://doi.org/10.1145/3318464.3389703

[27] 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

[28] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: an end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. 1253–1267.

[29] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.

[30] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (March 1984), 38–71. https://doi.org/10.1145/348.318586

[31] Shoji Nishimura and Haruo Yokota. 2017. QUILTS: Multidimensional Data Partitioning Framework Based on Query-Aware and Skew-Tolerant Space-Filling Curves. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1525–1537. https://doi.org/10.1145/3035918.3035934

[32] OmniSci [n.d.]. OmniSci. https://www.omnisci.com/. Accessed: 2021-03-24.

[33] Beng Chin Ooi, Ron Sacks-davis, and Jiawei Han. [n.d.]. Indexing in Spatial Databases.

[34] E. J. Otoo, Doron Rotem, and Sridhar Seshadri. 2007. Optimal Chunking of Large Multidimensional Arrays for Data Warehousing. In *Proceedings of the ACM Tenth International Workshop on Data Warehousing and OLAP* (Lisbon, Portugal) *(DOLAP '07)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/1317331.1317337

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

[36] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 12 (July 2020), 2341–2354. https://doi.org/10.14778/3407790.3407829

[37] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. 2002. A Case for Fractured Mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China) *(VLDB '02)*. VLDB Endowment, 430–441.

[38] 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

[39] Florin Rusu and Yu Cheng. 2013. A Survey on Array Storage, Query Languages, and Systems. arXiv:1302.0103 [cs.DB]

[40] S. Sarawagi and M. Stonebraker. 1994. Efficient organization of large multidimensional arrays. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. 328–336. https://doi.org/10.1109/ICDE.1994.283048

[41] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[42] Hari Singh and Seema Bawa. 2017. A Survey of Traditional and MapReduceBased Spatial Query Processing Approaches. *SIGMOD Rec.* 46, 2 (Sept. 2017), 18–29. https://doi.org/10.1145/3137586.3137590

[43] Muthian Sivathanu, Midhul Vuppalapati, Bhargav S. Gulavani, Kaushik Rajan, Jyoti Leeka, Jayashree Mohan, and Piyus Kedia. 2020. INSTalytics: Cluster Filesystem Co-Design for Big-Data Analytics. *ACM Trans. Storage* 15, 4, Article 23 (Jan. 2020), 30 pages. https://doi.org/10.1145/3369738

[44] 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/. Accessed: 2021-05-31.

[45] Emad Soroush, Magdalena Balazinska, and Daniel Wang. 2011. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *Proceedings of the*

2011 ACM SIGMOD International Conference on Management of Data (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/1989323.1989351

[46] 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.

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

[48] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, United States, 569–574. https://doi.org/10.1109/MDM.2019.00121 20th IEEE International Conference on Mobile Data Management (MDM), MDM 2019 ; Conference date: 10-06-2019 Through 13-06-2019.

[49] 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

[50] Ramon Antonio Rodriges Zalipynis. 2018. ChronosDB: Distributed, File Based, Geospatial Array DBMS. *Proc. VLDB Endow.* 11, 10 (June 2018), 1247–1261. https://doi.org/10.14778/3231751.3231754

[51] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton. 1997. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD Rec.* 26, 2 (June 1997), 159–170. https://doi.org/10.1145/253262.253288

[52] 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.