Dynamic Data Layout Optimization with Worst-case Guarantees

Kexin Rong
Georgia Institute of Technology, VMware
krong@gatech.edu

Paul Liu
Stanford University
paul.liu@stanford.edu

Sarah Ashok Sonje
Georgia Institute of Technology
sarah.sonje@gatech.edu

Moses Charikar
Stanford University
moses@cs.stanford.edu

***Abstract*—Many data analytics systems store and process large datasets in partitions containing millions of rows. By mapping rows to partitions in an optimized way, it is possible to improve query performance by skipping over large numbers of irrelevant partitions during query processing. This mapping is referred to as a data layout. Recent works have shown that customizing the data layout to the anticipated query workload greatly improves query performance, but the performance benefits may disappear if the workload changes. Reorganizing data layouts to accommodate workload drift can resolve this issue, but reorganization costs could exceed query savings if not done carefully.**

In this paper, we present an algorithmic framework OREO that makes online reorganization decisions to balance the benefits of improved query performance with the costs of reorganization. Our framework extends results from Metrical Task Systems to provide a tight bound on the worst-case performance guarantee for online reorganization, without prior knowledge of the query workload. Through evaluation on real-world datasets and query workloads, our experiments demonstrate that online reorganization with OREO can lead to an up to 32% improvement in combined query and reorganization time compared to using a single, optimized data layout for the entire workload.

***Index Terms*—data layout optimization, online algorithms**

I. Introduction

To keep up with increasingly large data demands, modern data analytics systems partition data in chunks containing millions of records, which are then compressed and persisted in cost-effective storage options such as Amazon S3. These partitions are often the smallest unit for I/O operations, meaning that all partitions containing relevant data must be accessed in their entirety during query processing.

The mapping from individual data records to different partitions, known as the *data layout*, can have a significant impact on query performance. Query optimizers utilize partition-level metadata, such as the min-max ranges of each column, to determine which partitions can be skipped during query processing, thereby enhancing performance [1]–[3]. By default, many systems simply partition the dataset according to one or more predefined sort columns, such as the arrival time of data records [4], [5]. However, this means that queries unrelated to the arrival time may still need to access a large number of partitions. Recent research shows that customizing the data layout to specific query workloads can significantly improve data-skipping performance [6]–[8]. For example, Qd-tree [7] uses query predicates to partition the dataset in a way that maximizes partitions skipped for the given query workload.

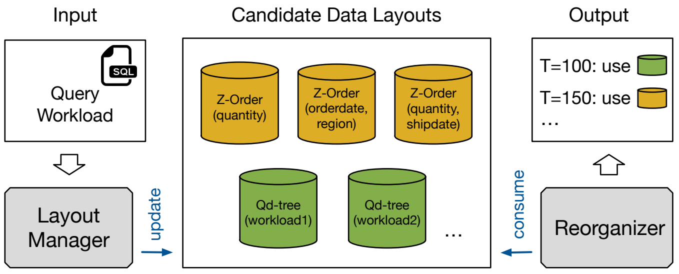

Fig. 1: Overview of OREO, an algorithmic framework for making online data reorganization decisions to minimize the total costs of query processing and data reorganization over an unknown query sequence.

However, the performance of these workload-aware layouts can degrade significantly when the target query workload drifts. Reorganizing the data, or switching to a different data layout, to adapt to workload changes requires accessing and rewriting large amounts of data, and could be $10\times$ to $100\times$ more expensive than running a full scan query on the table. It may be worth incurring this cost if the new layout leads to future query cost savings that outweigh the upfront cost of reorganization. Since users pay for the cost incurred by both query processing and reorganization, data analytics systems need to consider both costs instead of only focusing on reducing the query costs. In fact, rule-based reorganization heuristics have already been implemented in systems such as Snowflake [9] and Delta Lake [10] to continuously optimize the organization of datasets during query processing. However, the heuristics are often set empirically without any guarantees of performance, and it is not clear how these systems can reason about the performance implications of their reorganization decisions.

In this work, we introduce OREO (Online Re-organization Optimizer), an algorithmic framework that outputs a data reorganization schedule to minimize the total query and reorganization costs over an unknown query sequence, while providing provable worst-case performance guarantees (Figure 1). Since it may not always be possible to access historical workloads, such as with private customer datasets, our approach explores the "online" extreme on the design spectrum and assumes that *no prior knowledge of the query workload* is available. This forces us to design the algorithm to react to the query stream on the fly. Despite this, we show that our algorithm can perform favorably compared to algorithms that have access to the entire workload in advance. To achieve this, OREO leverages recent advances in workload-aware data layouts to generate new candidate layouts during query processing, and makes reorganization decisions by adapting classical results from the study of Metrical Task Systems (MTS) in online learning [11]. OREO provides guarantees in the form of *competitive ratios*, which is the total cost of online reorganization divided by that of an optimal offline solution.

A key technical contribution lies in the ability of OREO to decouple the process of generating data layouts from the process of making reorganization decisions. This separation is necessary because, in classic Metrical Task Systems (MTS), the system operates in a fixed set of states, equivalent to the set of candidate data layouts in our application. The total number of states directly affects the quality of the online solution, with a smaller state space leading to a better competitive ratio. However, the state space for online reorganization is prohibitively large and intractable to work with, as it includes all possible partitionings of a dataset. With prior knowledge of the workload, we can reduce the size of the state space by precomputing a small set of good-performing data layouts using workload-aware layout designs. Without such knowledge, the decision maker must adapt as the system observes more queries, adding better-performing data layouts to the state space incrementally. This requires the state space to change over the course of query processing, instead of remaining fixed as in traditional Metrical Task Systems.

To address this challenge, we propose a novel, *dynamic* variant of the classic uniform metrical task system problem, which permits arbitrary modification of the state space during query processing. The modifications are modeled as state management queries, which can add and remove any system state at any time during query processing. This new framework allows us to separate the state generation and state transition into two independent components: the REORGANIZER and the LAYOUT MANAGER. The REORGANIZER adapts a classic randomized algorithm for uniform metrical task systems to support dynamic state spaces and achieves a provably tight competitive ratio, similar to guarantees provided in the original setup. The LAYOUT MANAGER, on the other hand, is responsible for updating the state space. It starts with a default layout such as partitioning by time, and generates new layouts tailored to the current workloads on-the-fly as the system observes more queries. The LAYOUT MANAGER also takes into account the diversity of the layouts generated to prune redundant states and keep the state space compact. Together, the framework offers a systematic approach for balancing query improvements and reorganization costs over an unknown query sequence.

In summary, this paper makes the following contributions:

- A formal online algorithm framework, OREO, to model the online layout optimization problem. The problem setup differs from traditional online learning literature in that it does not assume that all the states are known apriori.
- An MTS algorithm that achieves an asymptotically tight competitive ratio under dynamically changing states.
- Evaluation of the framework on several real-world datasets and workloads. Our results show that online reorganization without workload knowledge improves upon an offline layout precomputed for the entire workload by up to 32% in end-to-end compute time.

II. Background and Motivation

We start this section with a discussion of current industry practices for dynamic data layout optimization, which inspire the design of OREO (§ II-A). We then provide background on classic results from Metrical Task Systems, which serve as the theoretical foundation of OREO (§ II-B).

A. *Motivation: Dynamic Layout Optimization in Practice*

Current systems often implement dynamic data layout optimization policies via predefined rules or thresholds. These heuristics, simple to implement and maintain, are particularly useful when historical workload data is not accessible, such as with private customer datasets. For example, Delta Lake provides a layout optimization feature based on Z-Order that can be activated manually or automatically when the number of small files exceeds a threshold during table creation and merges [10]. Snowflake has a rule-based automatic clustering feature that reorganizes data automatically when more than a certain number of data partitions overlap in a specified column [12]. Notably, users incur charges based on the computational resources used during this reorganization when the automatic clustering is enabled. Therefore, it is crucial to balance the costs of query processing with those of reorganization in these systems. Both Delta Lake and Snowflake perform reorganization in the background through a separate process, thereby minimizing its impact on query performance.

OREO draws inspiration from these current rule-based reorganization practices and aims to provide a more formal framework for considering the trade-off between query costs and reorganization costs. Similar to rule-based heuristics, OREO responds to the query stream and makes reorganization decisions on-the-fly. However, unlike heuristic methods which are often set empirically, OREO provides a provable worst-case performance guarantee over all possible input sequences.

B. *Background: Metrical Task Systems*

OREO builds upon classic results from Metrical Task Systems (MTS) [11]. In MTS, the decision maker controls a system that can be in one of n system states. The states are associated with a distance metric (i.e., obeying the triangle inequality), which defines the movement cost of switching system states. The system takes as input an unknown sequence of tasks, where each task incurs a (possibly different) service cost in each system state. For each new task, the system can choose to service it in the current state or to move to a different state and then service it by paying an additional movement cost. The goal of the decision maker is to minimize the sum of service costs and movement costs over the task sequence which is processed one-by-one, without knowledge of the future.

The main benchmark of comparison for MTS is the *competitive ratio*. The competitive ratio is the worst case over all possible input sequences of the performance of the online algorithm, which observes one new task at a time, divided by the performance of the optimal offline algorithm that is presented with the entire task sequence in advance. Several algorithms with varying competitive ratios for MTS exist, each with different assumptions on the state space. The tightness of the bound is typically related to the size of the state space [11], [13], [14].

The key difference between our setup and the traditional MTS problem is that MTS works with a fixed set of states that is known to the decision maker apriori, whereas we would like to allow the state space to evolve. Our work introduces a novel MTS problem formulation that addresses this difference and achieves a tight competitive ratio that is logarithmic in the maximum size of the dynamic state space. The results are therefore of interest beyond the scope of the online layout optimization problem we study in this paper.

III. Overview

In this section, we present the problem formulation (§ III-A), an overview of OReO's components and workflow (§ III-B), as well as the main assumptions and limitations (§ III-C).

A. Problem Formulation

We formulate the online layout optimization problem as a variant of uniform metrical task systems (UMTS) studied in the literature. We are given a set of states $\mathcal{S}$ and an ordered stream of queries $\mathcal{Q}$. For each $s \in \mathcal{S}$ and $q \in \mathcal{Q}$, there is some cost $c(s,q) \in [0,1]$ of servicing the query q in state s. To switch between states in $\mathcal{S}$, there is a reorganization cost of $\alpha > 1$. The goal is to service all queries in $\mathcal{Q}$ while minimizing the sum of query costs and reorganization costs over the entire sequence, reflecting the total compute costs users would pay to use systems like cloud data warehouses.

To model the introduction of new data layouts while processing the query stream, we propose a novel *dynamic* variant of UMTS, D-UMTS, where the set of states $\mathcal{S}$ is allowed to vary over time. In D-UMTS, we permit arbitrary addition and removal of states from the state space $\mathcal{S}$ in the midst of query processing, thus making the set of states dynamic. The flexibility of removing states allows for controlling the size of the state space in our applications. In the rest of the text, we use states and data layouts interchangeably in the context of describing our framework.

We model the query and reorganization costs as the total compute time incurred by querying and partitioning the datasets. To estimate the query cost, we use the fraction of the dataset accessed by each query, which has been shown to be a reliable proxy for query performance [7], [15]. To estimate the reorganization cost, we represent the relative overhead of reorganizing (which includes compressing and writing partitions) compared to querying (mostly reading partitions) via a parameter α in the cost model. α is the *expected* ratio between the compute time spent on reorganization compared to the time spent on a full table scan query ($\alpha := \mathbf{E}[\frac{\text{reorganization time}}{\text{full table scan}}]$). The value of

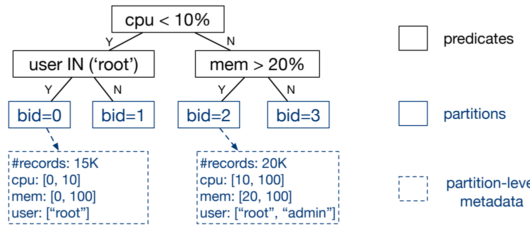

Fig. 2: Example of a Qd-tree and its partition-level metadata.

α varies based on system configurations and can be measured experimentally. In our experiments, α typically falls in the range of $60\times$ to $100\times$, similar to values reported in previous work [8].

We analyze the worst-case performance of the framework using the classic *oblivious* adversary model [13], meaning that an adversary must prepare the entire sequence for queries and state modifications in advance. Since the set of states is constantly changing through state modification requests, the adversary must additionally use the same set of states available to our algorithm at all times. Note that oblivious adversaries are quite powerful, as they necessarily force our algorithm to be randomized. If our algorithm is deterministic, the adversary can simulate our algorithm on its input in advance, and again prepare a sequence to remove the exact state that the system is currently in. Finally, we assume that the adversary does not have access to any random bits used by our algorithm.

B. Framework Overview

OReO consists of two main components, the Layout Manager and the Reorganizer (Figure 1).

The Layout Manager is the producer of the dynamic state space. It constantly generates new data layouts according to recent samples of the query stream and issues state management queries to add and remove states to and from the state space. Examples of layout generation methods include simple heuristics such as sorting and Z-ordering [16] with user-defined columns, as well as more sophisticated techniques that directly make use of query workloads such as bottom-up row grouping [6] and Qd-tree [7], [8]. The layout manager is agnostic to the underlying data layout generation mechanism as long as it supports the following two functionalities:

- `generate_layout`($\mathcal{D}$, $\mathcal{Q}$, k): This procedure uses the given dataset sample $\mathcal{D}$, query workload $\mathcal{Q}$, and target number of partitions k to generate a data layout, which is a mapping function that assigns data records to partitions.
- `eval_skipped`(s, $\mathcal{Q}$): This procedure estimates the fraction of data partitions skipped on the given query workload $\mathcal{Q}$ and data layout s.

As a concrete example, suppose Qd-tree is used to partition the dataset. The layout manager first calls `generate_layout` to compute a candidate layout on a small sample of the dataset (e.g., 0.1% to 1% of the data). Note that creating layouts based on these small samples is considerably less resource-intensive compared to executing a full reorganization. Furthermore, previous studies have demonstrated that layouts derived from small samples are sufficiently accurate representations of those that would be generated from the complete dataset. `generate_layout` returns the generated layout in the form of a binary decision tree, with each inner node containing a predicate selected from the query workload (Figure 2). We can assign data records to partitions by routing records through this tree until they reach one of the leaf nodes. In addition, by comparing the query with partition-level metadata (e.g., partition size, range/distinct values of each column) stored in the tree, the query optimizer can identify a list of partitions that can be skipped for the query. Since `generate_layout` operates on a small sample and `eval_skipped` references metadata instead of the actual data, the layout manager can efficiently explore multiple layout candidates.

The second component, the REORGANIZER, is the consumer of the dynamic state space. It queries the state space to get a set of data layouts that the system can choose from at a given time. During query processing, the REORGANIZER observes each new query and makes the decision of whether to keep the current data layout or switch to a new layout using D-UMTS. Similar to common practices in current systems [12], [17], we assume that reorganization happens via a separate process in the background using a (partial) copy of the data and that queries are still serviced on the existing data layout while reorganization is in progress. After reorganization is completed, the new layout is swapped with the existing layout with minimal impact on the query performance.

C. *Assumptions and Limitations*

We consider dynamically optimizing data layouts for a static dataset. For streaming data that is ingested continuously, reorganizing the entire dataset with each new data point arrival is not practical. Instead, we could batch newly arrived data and reorganize them separately from the already ingested data. This approach resembles existing layout optimzation practices that allow users to incrementally change clustering keys by only applying the change on newly ingested data [18].

In distributed settings, we assume that each node can independently reorganize its local data in response to queries directed at it. For vertically partitioned data, it makes sense to handle frequently accessed partitions separately from less accessed ones, as the former would likely benefit more from dynamic reorganization. In horizontal partitioning, dynamically changing the sharding key after the system is in operation could be challenging [19], which motivates the setup where each shard independently manages its reorganization. However, dynamically changing the sharding key is beyond the scope of our paper, as it entails additional considerations such as load balancing that is not covered by the D-UMTS framework.

We focus on scenarios where the time taken for reorganization is relatively short compared to the rate at which query patterns change. For example, we have observed that in production query workloads at VMware's internal data platform SuperCollider, the query patterns remain stable over short periods (e.g., days) but can shift over longer spans (e.g., months). Rapidly changing query patterns could render new data layouts outdated by the time reorganization is complete. In practice, systems can also mitigate this risk by dynamically allocating resources to ensure reorganization finishes efficiently [9].

IV. DYNAMIC REORGANIZATION VIA MTS

The first component of OREO is the MTS-based REORGANIZER. The REORGANIZER observes the query stream and decides when the system should switch to a different data layout and if so, which layout to switch to. On a high level, the REORGANIZER extends the classic algorithm of Borodin, Linial, and Saks [11] to allow arbitrary modification of the state space during query processing.

A. *Overview of the Algorithm of Borodin, Linial, and Saks*

We first present an overview of the classic algorithm [11] (Algorithms 1-3). The algorithm takes as input a query stream $\mathcal{Q}$ and a fixed set of states $\mathcal{S}$, and returns a set of states $\mathcal{H}$, one per query, that the system should be in *before* processing the query.

Algorithm 1 ProcessQueries$(\mathcal{Q},\mathcal{S})$

```
   Processes queries in ordered set $\mathcal{Q}$ with states in $\mathcal{S}$.
   $\mathcal{C}$: counters, $\mathcal{S}_A$: active states
1: $\mathcal{S}_A,\mathcal{C} \leftarrow$ ResetStates($\mathcal{S}$)
2: $s_c \sim$ Uniform($\mathcal{S}_A$)            ▷ Randomly select a current state
3: $\mathcal{H} \leftarrow \{s_c\}$
4: for $q \in \mathcal{Q}$ do
5:     $s_c,\mathcal{S}_A,\mathcal{S},\mathcal{C} \leftarrow$ UpdateCounters($q,s_c,\mathcal{S}_A,\mathcal{S},\mathcal{C}$)
6:     $\mathcal{H} \leftarrow \mathcal{H} \cup \{s_c\}$
7: return $\mathcal{H}$
```

Algorithm 2 ResetStates$(\mathcal{S})$

```
   Creates new active states $\mathcal{S}_A$ and resets all counters.
1: $\mathcal{S}_A \leftarrow \mathcal{S}$
2: $\mathcal{C}(s) \leftarrow 0$ for $s \in \mathcal{S}_A$
3: return $\mathcal{S}_A,\mathcal{C}$
```

Algorithm 3 UpdateCounters$(q,s_c,\mathcal{S}_A,\mathcal{S},\mathcal{C})$

```
   Updates the counters of states after processing query $q$ with current state
   $s_c$.
1: $\mathcal{C}(s) \leftarrow \mathcal{C}(s)+c(s,q)$ for $s \in \mathcal{S}_A$        ▷ Update counters
2: $\mathcal{S}_A \leftarrow \{s \mid \mathcal{C}(s) < \alpha$ for $s \in \mathcal{S}_A\}$
3: if $s_c \notin \mathcal{S}_A$ then                  ▷ If current state counter is full
4:     if $\mathcal{S}_A = \emptyset$ then                       ▷ All counters are full
5:         $\mathcal{S}_A,\mathcal{C} \leftarrow$ ResetStates($\mathcal{S}$)            ▷ Start new phase
6:     $s_c \leftarrow$ Uniform($\mathcal{S}_A$)                  ▷ Switch to new state
7: return $s_c$, $\mathcal{S}_A,\mathcal{S},\mathcal{C}$
```

Algorithm 1 initializes a set of "counters" starting at 0 for each state in $\mathcal{S}$. For each new query q that is processed, each counter increases by the cost $c(s,q)$ of serving q in state s. A counter is considered "full" when the cumulative cost is at least α. When the counter for the current state is full, the system randomly switches to another state whose counter is not full with uniform probability. When there are no more states to switch to (all counters are full), the algorithm "resets" all counters back down to 0 (Algorithm 2). The periods between the state resets are called "phases" of the algorithm. As shown in [11], the algorithm has a competitive ratio of $O(\log|\mathcal{S}|)$.

Intuitively, the counters measure the cost a state would have incurred if it processed all the queries in a phase. All the counters are at least α after a phase, so the optimal algorithm is guaranteed to incur a cost of at least α (had it stayed in a state during the entire phase). This is the key to bounding the performance of the optimal algorithm during the analysis.

Applying this algorithm to our layout optimization problem has the following implications. The states are different data layouts, and the service cost is the fraction of data accessed from running the query on the current data layout, which can be estimated using a small amount of partition-level metadata. Given that the relative overhead of reorganization versus querying is α, each phase ends when all counters are above α. Note that the algorithm's behavior is influenced by α – higher values of α generally result in fewer reconfiguration moves, while lower values of α will result in more reconfiguration moves. The effect of α is evaluated in Section VI-D.

In the original algorithm, each phase begins with transitioning to a random state. This makes the analysis of the algorithm simpler and cleaner. To reduce reorganization costs, a simple but effective optimization is simply allowing the algorithm to stay in the current state when starting a new phase instead of forcing it to move to a random state, thus saving on this initial random transition cost. Since each phase is independent, this optimization does not asymptotically affect the competitive ratio of the randomized algorithm. Empirically, we have observed that having the option to stay in the current state significantly improves the reorganization cost.

However, one difficulty lies in how to initialize the state space $\mathcal{S}$. Although the algorithm assumes that we can work with a fixed set of states throughout query processing, in practice we do not know this set apriori. Without knowledge of the query workload, what we would like to do is to start with a default data layout, such as Z-ordering on one or more predefined sort columns, and as the systems observe more queries, compute better-performing, workload-aware layouts to add to the state space $\mathcal{S}$ incrementally. The next section describes how we have modified this classic algorithm to permit such modification of the state space $\mathcal{S}$ during query processing.

B. Supporting Dynamic State Spaces

We model the modification of the state space via state update queries, which add and remove any state in the state space during the processing of service queries. The flexibility of removing states allows for controlling the size of the state space in our applications (e.g., to bound the amount of metadata maintained by the algorithm). Algorithm 4 summarizes our proposed modification to handle state update queries in the framework. If a new state is added in the middle of the phase, we simply defer the new state to the next phase. In other words, the algorithm behaves as if no additions have happened in the current phase, and resets the active state space to include any new states when the next phase starts. If an existing state is deleted in the middle of a phase, we set the counter of this state to α to mark that the algorithm can no longer switch to this state in the current phase. If all remaining counters are full after removing this state, we simply reset all counters to 0 and start a new phase with the updated state set. If the state that our system is currently in gets deleted, we follow the same procedure as if the current state is full, and randomly switch to another state that is still available.

Since the original guarantee of an $O(\log|\mathcal{S}|)$ competitive ratio is no longer applicable in the dynamic setting, one open question is how well do our proposed modifications work for D-UMTS. The following analysis shows that our algorithm costs no more than $O(\log|\mathcal{S}_{max}|)$ in a phase, where $|\mathcal{S}_{max}|$ is the maximum size of the active state space over the course of the query stream. Moreover, this competitive ratio is asymptotically optimal. In particular, we show the following:

Theorem IV.1. *Algorithm 4 solves* D-UMTS *with competitive ratio* $2H(|S_{max}|) \leq 2(1+\log|\mathcal{S}_{max}|)$ *where* $H(n)$ *is the* n*-th harmonic number and* $\mathcal{S}_{max}$ *is the largest set of states created by the update queries over the course of the stream.*

Proof. As discussed above, we break the execution of the algorithm into "phases", which are the execution intervals of the algorithm between calls to ResetStates (Algorithm 2). In other words, a phase begins when all counters in $\mathcal{S}_A$ are set to 0, and ends when $\mathcal{S}_A$ is empty. During a phase, the total set of states $\mathcal{S}$ may change as states are added or deleted. The counters have a natural interpretation: they are the lower bound of the cost a state would have incurred had an algorithm chosen to stay in that state during the entirety of the phase.

Let $\mathcal{S}_f$ and $\mathcal{S}_i$ be the set of states in $\mathcal{S}$ at the beginning and end of a phase. Let $\mathcal{A}_{opt}$ denote the optimal offline algorithm, and consider its cost in a phase. There are two cases, either $\mathcal{A}_{opt}$ moves to a newly added state in $\mathcal{S}_f \setminus \mathcal{S}_i$, or it remains in one of the original states in $\mathcal{S}_i$. In the former case, $\mathcal{A}_{opt}$ incurs movement cost at least α. In the latter case, every state in $\mathcal{S}_i$ has a counter at least α, which implies that $\mathcal{A}_{opt}$ incurs total cost at least α.

On the other hand, we show that the cost of our algorithm is bounded by $2H(|S_i|)$. Let $f(k)$ be the cost of our algorithm when $|\mathcal{S}_A| = k$ and note that $f(0) = 0$. We say a counter for state s has "reached" α if $\mathcal{C}(s)$ exceeds α at the end of line 1 in Algorithm 3. Let s_i be the i-th state in $\mathcal{S}_A$ whose counter reaches α, breaking ties arbitrarily. Note that being in state s_i means at most $|\mathcal{S}_A| - i$ states remain when we transition out of s_i. With equal probability, we transition to any state in $\mathcal{S}_A$, so

$$f(k) \leq 2\alpha + \sum_{i=1}^{k} f(k-i)/k = 2\alpha/k + f(k-1) = 2\alpha H(k),$$

where the factor of 2 accounts for the cost incurred by our current state serving queries (filling its counter from 0 to α) and from transitioning to a new state (movement cost α). The theorem follows by applying the above argument to every phase of the algorithm and noting that $H(|S_i|) \leq \log(|S_i|) + 1$. $\square$

Remark IV.1. *Algorithm 4 is also asymptotically optimal in the competitive ratio, as the competitive ratio for the non-dynamic uniform metrical task system is lower bounded by* $\log(|S_{max}|)$ *[11].*

Algorithm 4 ProcessQueries*$(\mathcal{Q},\mathcal{S})$

```
Processes queries in ordered set $\mathcal{Q}$ with states in $\mathcal{S}$.
1: $\mathcal{S}_A, \mathcal{C} \leftarrow$ ResetStates($\mathcal{S}$)
2: $s_c \sim$ Uniform($\mathcal{S}_A$)                              ▷ $s_c$: current state
3: $\mathcal{H} \leftarrow \{s_c\}$
4: for $q \in \mathcal{Q}$ do
5:     if $q$ removes state $s_d$ then                          ▷ Remove state
6:         $\mathcal{S}_A \leftarrow \mathcal{S}_A \setminus \{s_d\}$
7:         $\mathcal{C}(s_d) \leftarrow \alpha$
8:         if $\mathcal{S}_A = \emptyset$ then                         ▷ No state remains
9:             $\mathcal{S}_A, \mathcal{C} \leftarrow$ ResetStates($\mathcal{S}$)
10:        if $s_d = s_c$ then                      ▷ If current state is deleted
11:            $s_c \sim$ Uniform($\mathcal{S}_A$)            ▷ Switch to a random state
12:    else if $q$ adds state $s_a$ then                          ▷ Add state
13:        $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_a\}$
14:    else                                                  ▷ Service query
15:        $s_c, \mathcal{S}_A, \mathcal{S}, \mathcal{C} \leftarrow$ UpdateCounters($q, s_c, \mathcal{S}_A, \mathcal{S}, \mathcal{C}$)
16:        $\mathcal{H} \leftarrow \mathcal{H} \cup \{s_c\}$
17: return $\mathcal{H}$
```

C. Improving Reorganization with a Predictor

So far, our proposed algorithm makes no distinction between states and switches between states whose counters are not full randomly with uniform probability. In practice, users may have additional information on which states have better performance. In this section, we describe how to leverage such additional information to improve the performance of the algorithm.

We assume there exists a predictor $p(s,\mathcal{S}_A)$ which outputs transition scores for choosing the next state $s \in \mathcal{S}_A$. A higher transition score indicates a state that is predicted to be more efficient for the upcoming workload. A user can then use this predictor to construct a transition distribution where for each active state $s \in \mathcal{S}_A$, we jump to a state s with probability $\frac{p(s,\mathcal{S}_A)}{\sum_{s\in\mathcal{S}_A} p(s,\mathcal{S}_A)}$. In our algorithm, we would like to jump to the state in $\mathcal{S}_A$ that is the most efficient in its phase. Rank the states of S_A by their efficiency (each state s gets a rank from $\{1,...,|S_A|\}$, with 1 being the most efficient). When we choose uniformly at random, we get a state with $|S_A|/2$ on average. The hope is that the predictor p biases our jump towards the states with ranks closer to 1. More formally, the predictor p helps us construct a discrete distribution $\mathcal{D}_{|\mathcal{S}_A|}$ on $\{1,...,|S_A|\}$, where making a jump corresponds to sampling from $\mathcal{D}_{|\mathcal{S}_A|}$.

Concretely, in our application, we can use the query costs incurred by states in the previous phase as the predictor for their performances in the next phase. For states that are added in the middle of a phase, we can replay the queries processed in the current phase so far to fill in the counter, or simply initialize the counter to be the median of query costs incurred so far by existing states in the phase. When the counter for the current state is full, the algorithm can pick a new state based on a distribution that favors the better-performing states, instead of choosing a new state to switch to uniformly at random. For example, we can assign each state s with a weight w_s proportional to the average fraction of data skipped in the last phase, and switch to a new state s with probability $\frac{w_s^\gamma}{\sum_{s\in\mathcal{S}_A} w_s^\gamma}$. For $\gamma = 0$, this is equivalent to choosing the next state under a uniform distribution. For $\gamma > 0$, the probability distribution favors states that have larger weights, or better performances in the last phase. In fact, we can show that the competitive ratio is directly influenced by the accuracy of the distribution in predicting the "optimal state" over a single phase.

Theorem IV.2. *Fix any phase of Algorithm 1, and let $\mathcal{S}$ be the initial states at the beginning of the phase. Order the states of $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ in order of the time their counters reach α. Let $\mathcal{D}_m$ be distributions $\{1,2,...,m\}$ such that $\mathbf{E}[\mathcal{D}_m]/m \geq \beta$ for all $m = 1,2,...,n$.*[1] *Then the expected competitive ratio of Algorithm 1 is at most $O\Big(\log_{(1-\beta)^{-1}} n\Big)$.*

Proof. Recall that in the analysis of Algorithm 1, random transitions are made whenever a counter reaches α, and that the competitive ratio is proportional to the number of transitions made. Let $\mathcal{S}_i$ be the active states left on the i-th transition. To bound the number of transitions, let X_i be the indicator random variable for the event that $|\mathcal{S}_{i+1}| \leq 2(1-\beta)|\mathcal{S}_i|$, where transition i is made according to distribution $\mathcal{D}_i$. By Markov's inequality, $\Pr[X_i] \geq 1/2$. To reduce n states down to 0, we need $O\Big(\log_{(1-\beta)^{-1}} n\Big)$ of the indicators to be 1. Since each indicator has a positive constant probability of occurring, this implies that we get $O\Big(\log_{(1-\beta)^{-1}} n\Big)$ transitions in expectation. □

In other words, the competitive ratio of our algorithm improves if a predictor gives distributions that are biased towards the most efficient states. We evaluate the impact of the transition distribution on the overall performance of the algorithm in Section VI-D.

V. On-the-fly Layout Generation

The second component of OReO is the Layout Manager. As discussed earlier, without knowledge of the query workload, we cannot precompute a small set of relevant data layouts for the system to choose from. Instead, the system needs to generate new data layouts incrementally as it observes more queries. In this section, we present the design of the Layout Manager that determines 1) which new states to compute and 2) whether to admit these new states into the dynamic state space.

A. Generating Diverse Data Layouts

Workload-aware data layouts can experience performance degradation under changing query workloads. Therefore, it is necessary to periodically update data layouts to keep up with the changes. There are a few common strategies for performing this update.

The first strategy is to periodically compute new data layouts based on all queries that have arrived so far. However, this strategy is expensive as the query history keeps growing over time. It is also ineffective since recent queries can get out weighted by the larger volumes of historical queries. The second strategy is to keep a sliding window (of a fixed size or of a fixed time interval) of recent queries and periodically compute new layouts based on queries in the sliding window. The sliding window is bounded in size and responsive to changes, but has no memory

[1]I.e., our predictor is expected to give something within the top β fraction of ranks.

Algorithm 5 **Layout Management**

```
    $\mathcal{S}_A$: dynamic state space, $s$: new state
    $Q$: sample of past queries, $\epsilon$: distance threshold
1:  procedure ADMIT_STATE($\mathcal{S}_A$,$s$,$Q$)
2:      $c =$ EVAL_SKIPPED($s$,$Q$)
3:      $dists \leftarrow []$
4:      for $s_i \in \mathcal{S}_A$ do
5:          $c_i =$ EVAL_SKIPPED($s_i$,$Q$)
6:          $dists$.add$\left(\frac{||c-c_i||_1}{\text{dim(c)}}\right)$
7:      return $min(dists) > \epsilon$
```

of the distant past. The third strategy is to keep a reservoir sample of queries, which biases towards recent events but also keeps memories from the past. Reservoir samples provide a holistic picture of the entire history with a limited memory budget. However, since reservoir sampling always keeps around a small portion of old queries, layouts generated from the reservoir sample tend to perform slightly worse on the current workload compared to layouts generated from the sliding window.

We experimented with the use of both sliding windows and reservoir sampling for generating data layout candidates. We empirically found that using layouts generated solely from sliding windows gives the best overall performance (§ VI-D). One explanation is that, since the cost of switching between layouts is constant, it is more beneficial for the online algorithm to switch between layouts that have good performance for a specific workload, rather than layouts that have mediocre performance for multiple workloads. For example, consider a workload that iterates through each column of the dataset and generates 100 random range queries per column. Since reservoir sampling incorporates historical data, the sampled workload would always include queries on multiple columns. As a result, the LAYOUT MANAGER is unable to generate the ”optimal” layouts that partition according to a single column.

B. Expanding the Dynamic State Space

Now that new data layouts are constantly being generated, the LAYOUT MANAGER needs to decide how to update the state space accordingly. One option would be to simply admit all newly generated states. However the competitive ratio of the algorithm is directly related to the size of the dynamic state space. On the other hand, if we do not expand the state space at all, the system might miss out on new data layouts that are better performing for current and future workloads. Therefore, the LAYOUT MANAGER needs a policy that can determine whether to admit a new data layout into the dynamic state space.

We draw inspiration from prior work in online learning that tries to improve performance guarantees by reducing the size of state space [20]. For example, one can find a small number of states that form a covering of the space such that at least one of them is close to the optimal state for the system. Using similar reasoning, admitting highly similar data layouts to the dynamic state space does not help much with the query performance. Furthermore, it can incur additional reorganization costs as the reorganization might end up switching back and forth between similar layouts.

In our setup, we consider two data layouts to be similar if they incur similar query costs over the query stream. Specifically, we use a reservoir-based time-biased sampling (R-TBS) algorithm proposed in [21] to curate a representative query sample of size s over the query stream. We evaluate each data layout candidate i on the sample to get a cost vector $c_i = (c_{i,1},...,c_{i,n})$, where $c_{i,n}$ is the cost of executing query sample n on data layout i. We define the difference between two data layouts i and j to be a distance function (e.g., normalized L1 distance) of the query cost vectors c_i and c_j. The LAYOUT MANAGER is configured with a distance threshold $\epsilon \in [0,1]$ and only admits a new data layout if it is at least ϵ distance apart from all existing layouts in the state space. As the size of the state space becomes larger, it is increasingly difficult for a new data layout to get admitted, since it needs to be different enough from *all* existing states in the space. The pseudo-code for the procedure is presented in Algorithm 5.

Since the query samples are updated as the system processes more queries, the similarity between data layouts measured on these query samples can also change over time. For instance, two data layouts that initially had a distance of $> \epsilon$ when admitted to the state space may have a distance of $< \epsilon$ under query samples at a later time. This is acceptable because at the time when the LAYOUT MANAGER decides to admit the new state, switching to this state would make a difference in query performance. The LAYOUT MANAGER can also periodically prune the state space based on recent query samples and remove candidates that incur similar query costs to other layouts. We investigate the impact of the distance threshold ϵ on the size of the dynamic state space and the performance of our algorithm in Section VI-D. We also compare the dynamic state space to a fixed state space that is precomputed based on knowledge of the query workload in Section VI-C.

VI. EVALUATION

In this section, we empirically evaluate OREO on a variety of datasets and workloads. Overall, the experiments show that

- Compared to using a single data layout, dynamically switching layouts using OREO results in up to 32% improvement in end-to-end runtime on various datasets and workloads.
- OREO remains competitive and often outperforms alternative online reorganization strategies.
- OREO works across different data layout generation mechanisms and is robust to changes in key parameters.

A. Setup

1) Implementation: OREO is implemented as a lightweight Python library. OREO keeps track of different data layouts via partition-level metadata. With information such as row count, range of values (or distinct values for categorical columns) for each column in the partition, OREO is able to estimate query costs incurred by different layouts without accessing the underlying dataset.

We experimented with two data layout optimization techniques in the evaluation: Z-ordering and Qd-tree. First, we use

Z-ordering on user-defined columns to split the dataset into equal-sized partitions. To make Z-ordering workload-aware, we use the top three most queried columns in the sliding window, which can change over the course of the query stream. Second, we construct data layouts using Qd-trees, a workload-aware layout optimization technique. Our implementation of the Qd-tree uses the greedy construction algorithm and does not include any advanced cuts. Similar to practices in prior work, we construct Qd-trees based on a 0.1% to 1% sample of the dataset.

We perform evaluations on a VM with 64GB of RAM and 8 Intel(R) Xeon(R) Gold 6230R 2.10GHz CPUs. For methods that use variations of the classic MTS algorithm, we report the average from three runs. We report results using two metrics: physical runtimes in end-to-end experiments and logical costs in simulation.

End-to-end. We report end-to-end query execution time and reorganization time from a shallow integration with Apache Spark, similar to setups in prior works [8], [22]. Specifically, we include a new column for each table that specifies the unique partition ID (BID) that each row maps to. During query processing, we use the partition-level metadata to calculate a list of BIDs that the query needs to read. We then rewrite the original query to include an additional predicate that filters by the list of computed BIDs. For example, if the metadata indicates that only partitions 6 and 10 are relevant for the query, we write the query predicate to include an explicit partition filter `BID IN (6, 10)`. During reorganization, we update the BID column according to mappings generated from Qd-tree/Z-ordering. We rewrite rows with the same BID into a new partition, which is stored as a Parquet file on local disks. We exclude the time taken to compute data layouts since it is dependent on the specific layout optimization technique and is not the focus of this work. In general, data layouts can be computed rather efficiently on large datasets [7], [17]. We run Spark in stand-alone mode estimate the total query time using a sample of 2000 queries (around 10% of the workload).

Simulation. We also report results using logical costs in simulation. Per our cost model, we use the fraction of data records accessed (between 0 and 1) as a proxy for the query costs and assume the reorganization cost to be α. We investigate the effect of varying key parameters on the overall performance of the framework as well as the impact of our proposed optimizations in simulation.

2) Dataset and Query Workloads: We perform evaluation on three real-world datasets: TPC-H, TPC-DS and a production workload from an internal data lake system at VMware (Telemetry). We set the target partition count such that each partition contains between 1 million to 2 millions rows, and the average size of the partitions (Parquet files) is between 100 to 200 megabytes, consistent with recommended best practices.

TPC-H. We use the TPC-H data generator with a scale factor of 100 and denormalize all tables against the lineitems table. Due to the uniform nature of TPC-H data, we pre-divide the dataset into 10 equally sized partitions according to its primary keys and perform reorganization on one of the partitions, which contains around 40 millions rows, 58 columns. Similar to prior work [6], [7], we include 13 TPC-H query templates that touch the lineitem fact table (q1, q3, q4, q5, q6, q7, q8, q10, q12, q14, q17, q21[2]). The workload generator behaves like a state machine and samples queries from one query template for an arbitrary amount of time before switching to another random query template. The workload contains a total of 30,000 queries, generated from 20 query templates.

TPC-DS. We use the TPC-DS data generator with a scale factor of 10 and denormalize all tables against the `store_sales` table. The resulting table has around 26 million rows. We include 17 TPC-DS query templates that involve queries on the `store_sales` fact table and the dimension tables [3]. The workload includes 30,000 total queries, generated from 20 query templates using the same method as the TPC-H workload.

Telemetry. SuperCollider is an internal data platform that is used by over a hundred teams for data analytics in VMware. SuperCollider allows users to set up data jobs for bulk ingestion, run SQL queries over the data, discover and share datasets in a market space. We investigate a production table in SuperCollider that logs monitoring information for ingestion jobs. We extract a sample of the table as well as queries that touch this table in the past six months. In total, our samples include 24,000 queries and around 30 million rows for the table. The most popular predicates include range queries on the arrival time of the record, where the time interval ranges from a few hours to a few months, as well as filters on the name of the collector who has sent the data.

3) Methods of Comparison: We compare OREO against one offline baseline (Static) which observes the entire query workload in advance but is not allowed to change states, as well as two online baselines (Greedy and Regret) that do not have workload knowledge but can change states. The three online approaches (Greedy, Regret and OREO) utilize the same set of data layout candidates computed periodically based on a sliding window of recent queries, but use different reorganization strategies. By default, the relative reorganization cost α is set to 80 based on measurements obtained on our system setup, and the sliding window is set to include the most recent 200 queries.

Static. The method observes the entire query workload in advance and constructs a single layout that optimizes data skipping for the entire workload.

Greedy. The method compares the performance of the current data layout with a new data layout computed based on a sliding window of recent queries, and greedily switches to the new layout if it has a smaller query cost than the current one, without considering the reorganzation cost.

Regret. This method is similar to the Greedy strategy but considers the reorganization cost, inspired by work on storage

[2]q9 and q18 are excluded because they involve predicates that can not be directly evaluated using basic partition-level metadata. Specifically, q9 includes LIKE operator on a high cardinality column P_NAME and q18 includes a filter on the aggregate value of a group by.

[3]We used q3, q7, q13, q19, q27, q28, q34, q36, q46, q48, q53, q68, q79, q88, q89, q96, q98.

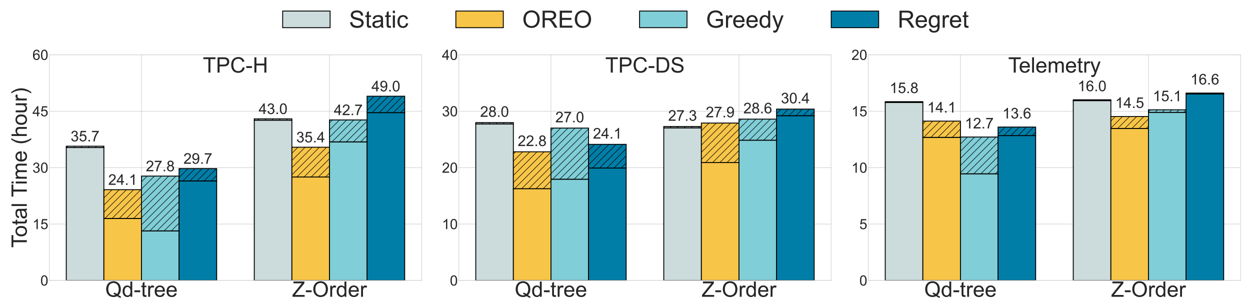

Fig. 3: Comparison of total query and reorganization time in Spark enabled by OREO with baselines. The top half of each bar with the hatches represents reorganization time, while the bottom half represents query time. Overall, dynamic reorganization improves upon a static, optimized layout by up to 32% in total compute time.

management in video analytics [23]. The method keeps track of the cumulative difference in query costs between the current data layout and alternative layouts over the query history. For each new layout, the method retroactively computes performance improvement compared to the current layout, using all queries that have been serviced on the current layout. The method switches to a new layout when the cumulative saving in query cost exceeds the reorganization cost.

OREO. A prototype that matches the descriptions given so far. Unless otherwise specified, the default parameter values for OREO in the experiments are $\epsilon = 0.08$ and $\gamma = 1$.

B. End-to-end Results

Figure 3 summarizes the end-to-end query and reorganization time incurred by different methods. The bottom half of each bar represents query time, while the top half with the hatches represents reorganization time. On the datasets that we experimented with, dynamic reorganization can outperform a single, precomputed data layout for the entire workload (Static) by 32.5%, 18.6% and 10.8% respectively using Qd-trees as the underlying partitioning technique. Compared to using Qd-trees, layouts generated using simple heuristics such as Z-ordering tend to have worse data skipping ratio, as reflected by the increased query costs. One notable exception is that for the TPC-DS dataset, the static Z-ordering layout outperforms the Qd-tree layout despite having a worse data skipping ratio. This is because the Z-ordering layout has a better compression ratio (16% smaller in file size) and the compression ratio is not taken into account in the design of Qd-trees. On the other two datasets, OREO is still 17.6% and 9.4% better compared to the best static layout.

Among the three online reorganization strategies, the Greedy baseline is the most aggressive in reorganization: it always switches to a better-performing layout regardless of the reorganization cost. Therefore, this baseline represents the smallest query costs that can be obtained by online strategies that share the same set of layout candidates. However, the downside is that it incurs large reorganization costs, especially when α is large. In contrast, the Regret baseline is most conservative, thus having overall large query costs and small reorganization costs. OREO lies somewhere in between and achieves the best overall cost in all but one case. In addition, Greedy and Regret both become less effective with Z-ordering. Since layouts generated from Z-ordering generally offer smaller improvements in query performance, the two baselines end up making fewer layout changes and incurring query costs that are close to those of the static layout. In comparison, OREO remains dynamic and offers noticeable improvements in query costs compared to the best static layout.

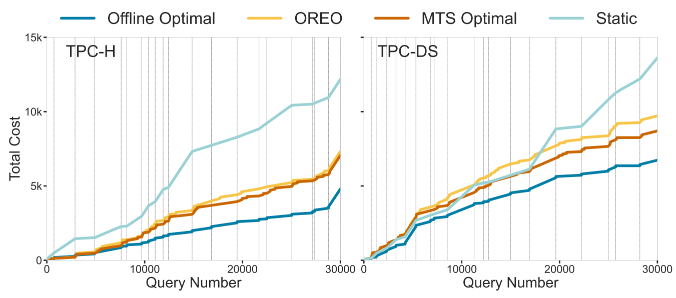

Fig. 4: Visualization of total query and reorganization cost over the query stream. Vertical gray lines indicate when the workload switches to a different query template. The query costs of OREO is 74% and 44% larger than the query costs of the optimal offline algorithm that observes the entire workload in advance and can switch states.

C. Gap to Optimal Algorithms

In the previous experiments, we compared with a static offline algorithm that observes the entire workload in advance but is not allowed to switch states. Note that our theoretical bounds are with respect to *any* algorithm that sees the entire workload in advance, even those that are able to switch states. To further understand the quality of the solutions produced by OREO, we compare with two additional methods that are allowed to switch states and leverage additional workload information:

- MTS Optimal: Instead of incrementally generating new data layouts, the method is given a fixed state space that includes the best data layout precomputed for each query template. The method uses OREO's modified MTS algorithm to determine how to switch between layout candidates.
- Offline Optimal: The method is presented with the entire workload in advance and switches to the best data layout

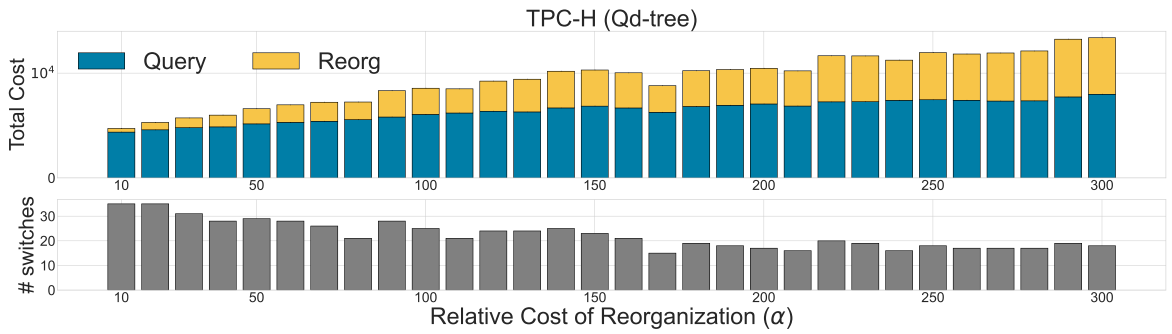

Fig. 5: Impact of reorganization cost (α) on the overall performance. As reorganization becomes more expensive, the total gains from dynamic reorganization decrease. The decrease is not monotonic due to changes in reorganization strategies.

for a query template as soon as template changes in the workload. Since this benchmark algorithm is given knowledge of the entire workload and has complete flexibility in changing the layout, it gives the lower bound on the query costs for any online solutions.

Figure 4 visualizes how the total costs change over the course of the query stream for all methods of comparison. The vertical gray lines in the background indicate when the workload switches to a different query template. Overall, OREO (yellow) using the dynamic state space performs slightly worse than the MTS Optimal (orange) which uses a fixed, precomputed state space. OREO's query costs are within 14% and 17% of the query costs of the MTS Optimal. This shows that additional workload information can benefit online algorithms by improving the quality of the state space. In addition, OREO's costs are 74% and 44% larger on the two datasets compared to the query costs of Offline Optimal (dark blue). Note that this is much better than the worst case $O(\log k)$ bound provided by the analysis, and the larger gap is because the Offline Optimal knows the entire workload and therefore does not experience any delay between template change and layout switches. MTS Optimal and OREO on the other hand, use online algorithms and do not have access to such information. The Offline Optimal makes 20 layout changes in total, corresponding to the number of template switches. OREO makes 22 and 29 layout changes while MTS Optimal makes 27 and 30 changes on the two datasets.

D. Detailed Analysis

In this section, we evaluate OREO's behavior under changes in key framework parameters.

1) Effect of reorganization cost α: Figure 5 shows the effect of relative reorganization cost α on the overall performance of the framework. Overall, the total gains from dynamic reorganization decrease as reorganization becomes more expensive. When α gets larger, the algorithm initially sticks to similar reorganization strategies despite the slightly increased reorganization cost. Eventually the reorganization becomes expensive enough that the algorithm adapts its strategy to make fewer layout changes in compensation. As a result, the number of layout changes decreases as α increases (35 changes at $\alpha = 10$ and 18 changes at $\alpha = 300$), with a few noticeable drops happening at around $\alpha = 80$ and 170. This also explains why the overall costs do not increase monotonically as α increases.

TABLE I: Relative cost of reorganization over query (α) ranges from 60× to 100× in our setup.

	File Size (MB)				
	16	64	256	1024	4096
Query (sec)	0.36±0.05	0.89±0.03	2.9±0.1	12.5±0.2	81.0±2.0
Reorg (sec)	24.6±0.7	70.0±4.3	276.6±4.1	1231.3±43.9	4854.1±88.9
α	69.0	78.7	95.4	98.4	59.9

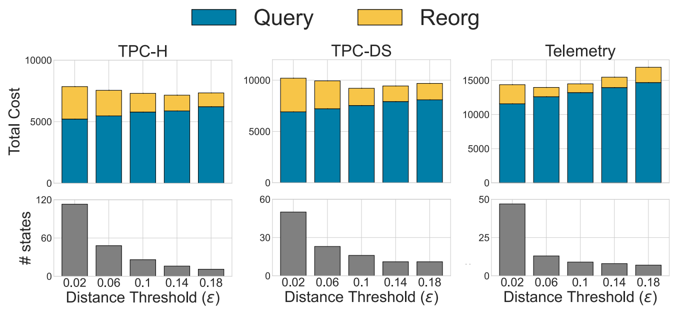

Fig. 6: Impact of distance threshold (ϵ) for admitting new layouts. Overall, the framework's performance is not very sensitive to specific choices of the ϵ.

In practice, users can measure typical values of α based on their system configuration to provide as inputs to OREO. We run Spark in stand-alone mode with files stored using the Parquet format on a local hard drive and measure the time taken to run a full table scan SQL query versus data reorganization for files ranging from 16MB to 4GB. In particular, reorganization includes 1) reading partitions from disk 2) updating the `BID` column and 3) repartitioning the dataset based on `BID` and 4) compressing and writing the new partitions to disk. Overall, we observe that the cost ratios range from 60× to 100× (Table I).

2) Effect of distance threshold ϵ: Figure 6 reports the effect of changing the distance threshold ϵ on the size of the dynamic state space and the overall performance of the framework. As ϵ increases, the size of the state space shrinks and we also observe a slight increase in query cost. The overall performance of the framework is not very sensitive

TABLE II: Impact of the transition distribution (γ), use of sliding window (SW) versus reservoir sampling (RS) for candidate data layout generation, and the impact of reorganization delay (Δ) on the MTS algorithm. The row in bold represents the default experiment parameter configuration. All results are reported in logical costs from simulation in unites of 10^3. Changes above 5% have been marked in the table.

	Query Cost			**Reorg Cost**		
	TPCH	TPCDS	Telemetry	TPCH	TPCDS	Telemetry
$\boldsymbol{\gamma}$**=1**	5.56	7.39	12.60	1.68	2.24	1.52
$\gamma=0$	5.75	7.49	12.60	2.32 (+38%)	3.04 (+36%)	1.84 (+21%)
$\gamma=2$	5.56	7.39	12.60	1.68	2.24	1.60 (+5.3%)
$\gamma=3$	5.56	7.39	12.56	1.68	2.16	1.52
SW	5.56	7.39	12.60	1.68	2.24	1.52
RS	6.51 (+17%)	9.03 (+22%)	14.66 (+16%)	2.00 (+19%)	2.16	2.24 (+47%)
SW+RS	5.59	7.19	12.55	2.40 (+43%)	3.04 (+36%)	1.44 (-5.3%)
$\boldsymbol{\Delta}$**=0**	5.56	7.39	12.60	1.68	2.24	1.52
Δ=40	5.88 (+5.7%)	7.65	12.67	1.68	2.24	1.52
Δ=80	6.20 (+12%)	7.89 (+6.8%)	12.75	1.68	2.24	1.52

to choices of the distance threshold ϵ, which makes it easy for users to set default parameters for the framework.

3) Effect of non-uniform transition distribution γ: Table II reports the effect of improving the transition distribution with a non-uniform transition distribution enabled by a state performance predictor (§ IV-C). Specifically, the parameter γ controls how much the transition distribution favors states that performed well in the last "phase" of the algorithm. The original MTS algorithm uses a uniform transition distribution ($\gamma=0$). Overall, using a biased distribution ($\gamma > 0$) improves the reorganization costs of the randomized algorithm by 17.3% to 27.6% but does not have a significant impact on the query costs. The performance is not very sensitive to specific choices of γ.

4) Sliding window vs reservoir sampling: OREO's layout manager continuously generates data layout candidates using a sliding window (SW) of recent queries. We evaluate the effect of alternative workload sampling strategies such as reservoir sampling (RS) and a combination of candidates from sliding window and reservoir sampling (SW+RS). Our results in Table II shows that the use of reservoir sampling led to an increase in query costs by up to 22% and reorganization costs by up to 47% when compared to the sliding window strategy. In addition, while the combined strategy (SW+RS) exhibited similar query costs to using a sliding window alone, the reorganization costs can increase by up to 43%.

5) Effect of reorganization delay Δ: OREO performs reorganization in the background, meaning that while a new layout is being created, some queries may still be using the old layout. We study the effect of the delay in the background reorganization on performance by changing the number of queries that are executed using the outdated layout each time the reorganizer decides to switch layouts. The delay does not impact the cost of the reorganization, as the cost is incurred as soon as the decision is made. However, longer delays lead to increased query costs, since the query savings do not take effect until the actual layout switch happens. The query costs increase by around 7 to 12% compared to the case where there is no delay when the number of queries served on outdated layouts equals α. In practice, this latency can be further decreased by dedicating more compute resources for reorganization.

VII. RELATED WORK

In this section, we discuss related work in data layout optimization and automatic database tuning.

1) Data Layout Optimization: Data layouts are mapping functions that assign each record in the dataset to different partitions. Partitions are often stored as individual files using compressed, columnar format on local disks or in remote storage.

Traditional layout designs such as round-robin, range, and hash partitioning use mapping functions that are independent of both the data distribution and the query workload [24], [25]. More recently, researchers have started to explore specialized layouts that purposefully overfit the layout design to specific datasets and workloads to achieve superior data skipping performance [26]–[28]. Fine-grained workload information such as query predicates are often used in these works to tightly couple the layout designs with target query workloads [6]–[8], [22]. Our framework leverages these recent developments in workload-aware data layouts as black boxes to generate a list of candidate data layouts that the systems can switch between. In fact, the better these workload-aware layouts perform, the more query cost we can potentially save by dynamically switching between them. We emphasize that our framework is agnostic to the specific partitioning mechanism used, as long as it can produce different data layouts given different target query workloads.

However, one caveat is that these workload-aware layout designs can experience significant performance degradation when the target query workload changes. We provide an example in Appendix A of the technical report [29], which shows that a static layout results in almost no savings under changing workloads. Recent work has proposed strategies such as modeling and embedding the variance of workloads into the partitioning design to improve its robustness to workload drifts [30]. Instead of improving the layout design, our work takes an orthogonal approach and explores how to make better use of existing layouts. Many more opportunities remain in the design and implementation of such systems that can self-optimize and adapt to changes to workload and data [31], including for non-tabular data such as videos [23].

2) Automatic Tuning in Databases: The online layout optimization problem is an instance of a broader class of problems that aim to automatically tune the physical design and configurations to improve the performance of database systems [32]–[37]. We focus the discussion below on online settings in which the system adapts the tuning based on changes in the query workload.

One promising approach is to make tuning decisions according to predictions of future behaviours. For example, researchers have used supervised learning to predict future workload patterns and the performance of different system configurations [38]–[41], reinforcement learning to model system behavior through experiences collected from interacting with the environment [42]–[44], or make assumptions on the distribution of query workloads [8]. For example, MTO [8] designs a reward function under the assumption that a certain number of additional queries from the same distribution can be executed before the workload switches, and searches for the reorganization strategy with the maximal reward via dynamic programming. In constrast, OREO uses online algorithms that neither assume prior knowledge of the query workload nor a specific workload distribution. Another approach is to design rules or heuristics for reorganization conditions, such as current industry practices discussed in § II-A. For example, SAT [45] monitors the ratio of the actual query selectivity and the data skipping rate, and triggers reorganization process when the ratio is a below certain threshold. While OREO is inspired by rule-based designs, it introduces a formal framework with worst-case performance guarantees.

3) Adaptive Index Tuning using Online Algorithms: The philosophy of our dynamic layout optimization is closely related to a line of work that leverages online algorithms for index tuning and recommendation in database systems [46]–[48]. In indexing tuning, one wants to adaptively create and delete in-memory indices to reduce query execution costs based on workload characteristics.

The two problems may seem similar at first glance, but the problem setup in adaptive index tuning has a different setup since it assumes asymmetric movement costs and a fixed state space. In index tuning, (nonclustered) indexes store pointers to data records and therefore, maintaining multiple indices requires one copy of the data with some additional storage for the indexing data structures. In contrast, data layouts determine how the dataset is actually sorted and stored and *realizing* multiple data layouts require storing multiple copies of the dataset; this should not be confused with *evaluating* the query costs on multiple data layouts, which can be estimated using partition-level metadata without accessing the underlying dataset. As a result, in index tuning, there is only the cost of creating additional indexes, and little to no cost with the "changing" of states, introducing an *asymmetric* movement cost. This difference allows algorithms in our model to achieve a much lower competitive ratio, as uniform metrics are much easier to analyze than asymmetric costs. For example, the WFIT algorithm [47] achieves a competitive ratio exponential in the number of possible indexes, whereas our ratio grows logarithmically. In addition, in all prior works on index tuning, the theoretical analysis assumes that the state space of possible indices on the table is fixed.

Metrical task systems can still be applied for asymmetric movement costs, albeit with a worse competitive ratio. The original paper of Borodin et al. [11] provides an $O(|S|^2)$-competitive algorithm for asymmetric costs following the triangle inequality. In [46], a 3-competitive algorithm for two-state MTS with asymmetric costs was presented as a special case. We include a proof of an improved competitive ratio for the classic algorithm [11] in this special case in Appendix C [29].

VIII. Discussion and Future Work

Our work marks an encouraging step towards applying online algorithms in dynamic data layout optimization. In this section, we outline key directions for future exploration.

First, our study primarily investigates single-table data layouts, but OREO is also compatible with multi-table configurations. In such setups, each table can maintain its own instance of OREO and make decisions based on a subset of query predicates relevant to the table. We report preliminary results in Appendix B of the technical report [29], which show that multi-table layouts that utilize predicates induced from joins [8], [49] show greater benefits from dynamic reorganization compared to layouts that optimize each table separately.

Second, OREO is based on results from uniform metrical task systems, which assume uniform switching costs between any pair of states. Extending our framework to support non-uniform metrics would increase the possible state space of data layouts. However, algorithms for non-uniform metrics in MTS [50]–[54] are considerably more complex. Adapting them to the dynamic variant we introduced in this paper presents an interesting venue for future work.

Third, OREO does not keep additional copies of the data with different layouts, except temporarily during reorganization. However, if we have the storage budget to maintain multiple layouts of the same dataset simultaneously, we could extend Algorithm 4 to accommodate this scenario. A variant of our algorithm tailored for this purpose is introduced in Appendix D of the technical report [29]. Further analysis of the optimal tradeoffs involved in maintaining multiple layouts, as well as exploration of alternative variants of our algorithm, could be pursued in an extended work.

IX. Conclusion

We introduce OREO, an algorithmic framework that decides when and how to change the data layout to minimize overall data access and movement over the entire query stream, without prior knowledge of the query workload. OREO leverages recent developments in workload-aware data layout designs and offers a systematic way to reason about the trade off between query and reorganization costs when utilizing such layout designs. OREO makes reorganization decisions in an online fashion by extending classic results for metrical task systems to support dynamic state space and achieves a tight competitive ratio that is logarithmic in the maximum size of the dynamic state space. Our empirical findings show that dynamic reorganization using OREO achieves sizable speed ups in end-to-end query and reorganization time compared to using a single, optimized data layout throughout, and compares favorably with oracles that observe the entire query stream in advance and switch states dynamically.

References

[1] G. Graefe, "Fast loads and fast queries," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2009, pp. 111–124.

[2] "Oracle Database Data Warehousing Guide: Using Zone Maps," https://docs.oracle.com/en/database/oracle/oracle-database/21/dwhsg/using-zone-maps.html, 2020, accessed October 2023.

[3] "Data skipping with Z-order indexes for Delta Lake," https://docs.databricks.com/en/delta/data-skipping.html, 2023, accessed October 2023.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[5] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash *et al.*, "Apache hadoop goes realtime at facebook," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 1071–1080.

[6] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1115–1126.

[7] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 193–208.

[8] J. Ding, U. F. Minhas, B. Chandramouli, C. Wang, Y. Li, Y. Li, D. Kossmann, J. Gehrke, and T. Kraska, "Instance-optimized data layouts for cloud analytics workloads," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 418–431.

[9] "Automatic Clustering," https://docs.snowflake.com/en/user-guide/tables-auto-reclustering, 2024, accessed February 2024.

[10] "Configure Delta Lake to control data file size," https://docs.databricks.com/en/delta/tune-file-size.html, December 2023, accessed Feburary 2024.

[11] A. Borodin, N. Linial, and M. E. Saks, "An optimal on-line algorithm for metrical task system," *J. ACM*, vol. 39, no. 4, pp. 745–763, 1992. [Online]. Available: https://doi.org/10.1145/146585.146588

[12] R. Shelly, "Automatic Clustering at Snowflake," https://medium.com/snowflake/automatic-clustering-at-snowflake-317e0bb45541, January 2022, accessed February 2024.

[13] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. cambridge university press, 2005.

[14] S. Irani and S. S. Seiden, "Randomized algorithms for metrical task systems," *Theor. Comput. Sci.*, vol. 194, no. 1-2, pp. 163–182, 1998. [Online]. Available: https://doi.org/10.1016/S0304-3975(97)00006-6

[15] K. Rong, Y. Lu, P. Bailis, S. Kandula, and P. A. Levis, "Approximate partition selection for big-data workloads using summary statistics," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2606–2619, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2606-rong.pdf

[16] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[17] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak *et al.*, "Delta lake: high-performance acid table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, 2020.

[18] "Use liquid clustering for Delta tables," https://docs.databricks.com/en/delta/clustering.html, February 2024.

[19] "Data partitioning guidance," https://learn.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning, accessed February, 2024.

[20] A. Cohen and S. Mannor, "Online learning with many experts," *CoRR*, vol. abs/1702.07870, 2017. [Online]. Available: http://arxiv.org/abs/1702.07870

[21] B. Hentschel, P. J. Haas, and Y. Tian, "General temporally biased sampling schemes for online model management," *ACM Transactions on Database Systems (TODS)*, vol. 44, no. 4, pp. 1–45, 2019.

[22] S. Sudhir, W. Tao, N. Laptev, C. Habis, M. Cafarella, and S. Madden, "Pando: Enhanced data skipping with logical data partitioning," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2316–2329, 2023.

[23] M. Daum, B. Haynes, D. He, A. Mazumdar, and M. Balazinska, "Tasm: A tile-based storage manager for video analytics," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1775–1786.

[24] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[25] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu *et al.*, "Enhancements to sql server column stores," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1159–1168.

[26] T. Kraska, "Towards instance-optimized data systems," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, 2021.

[27] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 985–1000.

[28] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "Lisa: A learned index structure for spatial data," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 2119–2133.

[29] "Dynamic Data Layout Optimization with Worst-case Guarantees," https://github.com/d2i-lab/oreo/blob/master/docs/tr.pdf, accessed November, 2023.

[30] Z. Li, M. L. Yiu, and T. N. Chan, "Paw: Data partitioning meets workload variance," *ICDE*, 2022.

[31] S. Madden, J. Ding, T. Kraska, S. Sudhir, D. Cohen, T. Mattson, and N. Tatbul, "Self-organizing data containers," in *CIDR*, 2022.

[32] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database tuning advisor for microsoft sql server 2005," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 930–932.

[33] S. Chaudhuri and G. Weikum, "Rethinking database system architecture: Towards a self-tuning risc-style database system." in *VLDB*, 2000, pp. 1–10.

[34] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "Db2 design advisor: Integrated automatic physical database design," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 1087–1097.

[35] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: a decade of progress," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 3–14.

[36] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM international conference on management of data*, 2017, pp. 1009–1024.

[37] A. Pavlo, M. Butrovich, L. Ma, P. Menon, W. S. Lim, D. Van Aken, and W. Zhang, "Make your database system dream of electric sheep: towards self-driving operation," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3211–3221, 2021.

[38] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 631–645.

[39] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based query performance modeling and prediction," in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, A. Kementsietsidis and M. A. V. Salles, Eds. IEEE Computer Society, 2012, pp. 390–401. [Online]. Available: https://doi.org/10.1109/ICDE.2012.64

[40] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Ai meets ai: Leveraging query executions to improve index recommendations," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1241–1258.

[41] A. Mahgoub, A. M. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "{OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 189–203.

[42] G. C. Durand, M. Pinnecke, R. Piriyev, M. Mohsen, D. Broneske, G. Saake, M. S. Sekeran, F. Rodriguez, and L. Balami, "Gridformation: towards self-driven online data partitioning using reinforcement learning," in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2018, pp. 1–7.

[43] B. Hilprecht, C. Binnig, and U. Röhm, "Towards learning a partitioning advisor with deep reinforcement learning," ser. aiDM '19. New

York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3329859.3329876

[44] B. Hilprecht, C. Binnig, and U. Rohm, "Learning a partitioning advisor for cloud databases," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 143–157. [Online]. Available: https://doi.org/10.1145/3318464.3389704

[45] X. Xie, S. Shi, H. Wang, and M. Li, "Sat: sampling acceleration tree for adaptive database repartition," *World Wide Web*, vol. 26, no. 5, pp. 3503–3533, 2023.

[46] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 826–835.

[47] K. Schnaitter and N. Polyzotis, "Semi-automatic index tuning: Keeping dbas in the loop," *Proc. VLDB Endow.*, vol. 5, no. 5, p. 478–489, jan 2012. [Online]. Available: https://doi.org/10.14778/2140436.2140444

[48] T. Malik, X. Wang, D. Dash, A. Chaudhary, A. Ailamaki, and R. Burns, "Adaptive physical design for curated archives," in *International Conference on Scientific and Statistical Database Management*. Springer, 2009, pp. 148–166.

[49] S. Kandula, L. Orr, and S. Chaudhuri, "Pushing data-induced predicates through joins in big-data clusters," *Proc. VLDB Endow.*, vol. 13, no. 3, p. 252–265, nov 2019. [Online]. Available: https://doi.org/10.14778/3368289.3368292

[50] J. Fakcharoenphol, S. Rao, and K. Talwar, "A tight bound on approximating arbitrary metrics by tree metrics," *Journal of Computer and System Sciences*, vol. 69, no. 3, pp. 485–497, 2004.

[51] A. Fiat and M. Mendel, "Better algorithms for unfair metrical task systems and applications," *SIAM Journal on Computing*, vol. 32, no. 6, pp. 1403–1422, 2003.

[52] S. Bubeck, M. B. Cohen, J. R. Lee, and Y. T. Lee, "Metrical task systems on trees via mirror descent and unfair gluing," *SIAM Journal on Computing*, vol. 50, no. 3, pp. 909–923, 2021.

[53] C. Coester and J. R. Lee, "Pure entropic regularization for metrical task systems," in *Conference on Learning Theory*. PMLR, 2019, pp. 835–848.

[54] F. Ebrahimnejad and J. R. Lee, "Multiscale entropic regularization for mts on general metric spaces," in *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[55] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

Appendix

A. Illustration of Reorganization Opportunities

We illustrate the potential performance benefits from dynamic reorganization analytically via a synthetic example. Specifically, we compare the query cost of a static data layout optimized for the entire query sequence to the cost of dynamically switching layouts based on workload changes.

Consider a dataset consisting of k numeric columns $c_1,...,c_k$, where each column contains values that are independently sampled from the uniform distribution $\mathcal{U}(0,1)$. The query set consists of point queries with predicates like $p_i = x$, $i \in \{1,...,k\}$ where x is drawn from the distribution $\mathcal{U}(0,1)$, i.e. asking for data points p whose i^{th} coordinate is x. The workload consists of k rounds, where round i generates Q queries on column c_i. Suppose that the underlying data storage is an axis-aligned space partitioning structure such as the k-d tree [55]. In this case, the $[0,1]^k$ data space is partitioned into a set of k-dimensional hyper-rectangles $\mathcal{B}$, whose volume sums to 1. Note that since each column is independently sampled, partitioning based on one column does not improve data skipping on other columns.

First, consider the static strategy which uses the same set of partitions $\mathcal{B}$ for all queries. We show that for this query workload, a single data layout does not allow good skipping performance especially for high-dimensional data. Consider a single partition $B \in \mathcal{B}$ with side lengths $l_1^{(B)}, l_2^{(B)}, \ldots, l_k^{(B)}$. The probability that a query of the form $p_i = x$ hits partition B is exactly the side length $l_i^{(B)}$ in dimension i. By linearity of expectation, the expected number of queries hitting this partition is proportional to $Q\sum_i l_i^{(B)}$. Each time the partition is hit, a cost proportional to $\mathrm{Vol}(B)$ is incurred since all of the partition's data needs to be read. The expected cost is at least

$$\sum_{B\in\mathcal{B}} Q\mathrm{Vol}(B)\sum_{i=1}^{k} l_i^{(B)} \geq \sum_{B\in\mathcal{B}} Qk\cdot \mathrm{Vol}(B)^{1+1/k} \geq Qk/|\mathcal{B}|^{1/k}.$$

The first inequality follows from the Arithmetic Mean-Geometric Mean inequality and the fact that $\mathrm{Vol}(B) = \prod_{i=1}^{k} l_i^{(B)}$; the second inequality follows from Jensen's inequality and the fact that the total volume of the partitions is equal to 1. Note that the cost without any data skipping is Qk. As the dimensionality k gets large, $|\mathcal{B}|^{1/k}$ tends towards 1, resulting in *almost no savings* from the partitioning.

In comparison, consider the dynamic strategy of partitioning the entire column i in round i (when column i is queried). This means that column i is divided into $|\mathcal{B}|$ equal sized intervals, with each interval corresponding to a partition in $\mathcal{B}$. Now each query in round i only reads one partition in $\mathcal{B}$ (which has volume $1/|\mathcal{B}|$), resulting in a total expected cost of $Qk/|\mathcal{B}|$ over the k rounds of Q queries. The dynamic strategy reduces the query costs by a factor of $|\mathcal{B}|^{1-1/k}$ compared to the static one, which is roughly $|\mathcal{B}|$ as k gets large.

B. Preliminary Multi-Table Results

In this section, we present our preliminary experiments on using OREO in a multi-table setup.

1) Implementation Details: We implement a simplified multi-table layout generation scheme inspired by MTO [8]. Instead of independently optimizing each table's layout, multi-table layout schemes leverage additional information about the joins through Data-Induced Predicates (DIPs) [49] to further improve data-skipping performance.

To illustrate, consider a scenario involving a dimension and a fact table. A query on the dimension table creates a set of local predicates. Using metadata of the partitions satisfying these local predicates, we can generate additional predicates (DIPs) that capture the range of join column values in the partitions. These DIPs can then be propagated to the fact table via the primary-foreign key relationship. This process effectively translates predicates on one table to data-skipping opportunities in its join tables.

For our experiments, we have limited our implementation to handle DIPs through one level of join. We included an extra query transformation step that augments fact table queries with DIPs generated from dimension tables. The LAYOUT MANAGER is agnostic to whether a predicate is induced. The REORGANIZER uses the transformed query to evaluate layout savings and make reorganization decisions. Hence, our implementation did not require modifications to OREO's layout management and reorganization components.

TABLE III: Comparison of query and movement cost on the `lineitem` table in the TPC-H dataset. Multi-table layout scheme that leverages DIPs demonstrates more gains from dynamic reorganization.

	Qd-tree			**Qd-tree+DIP**		
	Total	Query	Reorg	Total	Query	Reorg
Static	15324	15324	0	13037	13037	0
OREO	15355	14715	640	11815	10855	960
Periodic	14869	14149	720	13031	10471	2560
Regret	14703	14303	400	11565	11085	480

2) Performance Evaluation: We follow a similar setup to that in § VI-A. We compare OREO with both offline (Static) and online (Periodic and Regret) baselines, using the TPC-H dataset at a scale factor of 10 and a skew factor of 2. A total budget of 100 partitions was split across all tables proportional to their sizes, with each partition containing 700,000 to 2 million rows. We use the same TPC-H workload and reorganization cost ($\alpha = 80$) as in the main experiments. In the multi-table setup, each table independently runs its own instance of OREO. Table III reports the logical query and movement costs on the largest lineitem table using two layout generation schemes.

We observe that single-table layout schemes, such as the Qd-tree, offer negligible performance benefits in dynamic reorganization compared to the best static layout. This is because the layout management and reorganization components can only utilize a subset of queries that have predicates on the lineitem table. In comparison, the total query costs for all methods decrease when input queries are augmented with DIPs. OREO remains competitive and reduces the total query costs by up to 9.4% compared to the best static layout, which is generated using the entire query workload. These findings are in line with those from our main experiments.

C. MTS for Two-state Asymmetric Costs

In this section, we show that a variant of the classic MTS algorithm [11] also works for two-state asymmetric costs. Following [46], we assume there exists two states, s_0 and s_1, where the cost of moving s_0 to s_1 is 1, but the cost of moving s_1 to s_0 is 0.

Theorem A.1. *The classic* MTS *algorithm [11] is 6-competitive for two-state MTS with asymmetric costs.*

Proof. Let $\mathcal{A}_{OPT}$ be the optimal algorithm. Consider a single phase of the classic algorithm. If $\mathcal{A}_{OPT}$ switches states more than once, then it incurs a cost of at least 1, while classic incurs a cost of 1 from each state, and at most 1 from transitioning states. So the competitive ratio is 3. Now consider the case for when $\mathcal{A}_{OPT}$ switches states at most once, and consider two consecutive phases of the classic algorithm.

In this case, there are 8 different possibilities, corresponding to the two different state transitions that $\mathcal{A}_{OPT}$ could perform. For example, $\mathcal{A}_{OPT}$ can start in s_1 and transition to s_0 in the first phase, and go from s_0 to s_1 in the second phase. Across all 8 different possibilities, $\mathcal{A}_{OPT}$ incurs cost at least 1. In each possibility, classic incurs a cost of at most 6 across the two phases. Thus the competitive ratio of the classic algorithm is 6. □

D. Extension: Maintaining Multiple Layouts in Parallel

In this work, we assume that we do not have additional storage budget to maintain multiple layouts on the same datasets, except temporarily during the background reorganization. However, if we can afford to store multiple physical copies of the data simultaneously, each with a distinct layout or state, we can extend OREO in the following way (without considering load balancing implications).

Suppose we are in the state where the set $\mathcal{S}$ of layouts does not change. One variant of the algorithm would be to keep q randomly chosen states around in parallel, and switch only when *all states* have cost $> \alpha$. To handle queries, we assign them to the state with the lowest accumulated cost or counter thus far. Let $\mathcal{S}_A$ be the set of states in $\mathcal{S}$ which do not currently have their counters filled, and let $f(k)$ be the expected cost of our algorithm when $|\mathcal{S}_A| = k$. We have

$$f(k) \leq q*\alpha + \alpha + \sum_{i=0}^{k-q} f(i) * p_{i,k,q}$$

, where $p_{i,k,q}$ is the probability that the $i+1^{st}$ worst state of the k states was chosen in our set of q states. This is because:

- It takes $q \cdot \alpha$ to completely change to a new set of states.
- It takes α cost to query before we switch to a new set of states. We only perform queries on the best of the q states at any given time, and we stop when the best state's counter reaches α.
- When we do change to a set of new states, we have q "chances" to land at a state that has cheap expected cost. So $p_{i,k,q} = \frac{\binom{k-i-1}{q-1}}{\binom{k}{q}}$.

The number of states maintained in parallel, denoted as q, presents a tradeoff between query and reorganization costs. If $q = |\mathcal{S}|$ then we always get the best query cost, but it costs $q \cdot \alpha$ to reorganize each phase. If $q = 1$ then we rarely get the best query cost, but the expected query and reorganization cost is about $2\log(\alpha)$ in expectation. The best tradeoff likely lies between these extremes.

Another variant of the algorithm would be to switch to a new state whenever one of the q different states reach counter cost α. These variants presents an interesting opportunity for future work.