Instance-Optimized Database Indexes and Storage Layouts

by

Jialin Ding

B.S., Stanford University (2018)
S.M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 26, 2022

Certified by .....
Tim Kraska
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Instance-Optimized Database Indexes and Storage Layouts

by

Jialin Ding

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

For any modern database system, its physical design, which is composed of both the storage layout of the data itself and auxiliary data structures such as indexes, is a critical piece of maintaining high performance in the face of increasing data volumes. Existing physical design components are general-purpose: they achieve adequate performance for the average use case but don't achieve optimal performance for any individual use case. These physical design components expose numerous configuration knobs that users must manually tune to achieve better performance for their individual use case, but tuning complex systems is labor-intensive, and poor tuning can result in degraded performance and increased costs.

In this thesis, we explore how database systems can maximize performance while minimizing manual effort through *instance-optimization*, which is the process of designing systems that are able to automatically self-adjust in order to achieve the best performance for a given use case. We leverage instance-optimization to introduce novel designs for database indexes and data storage layouts that outperform existing state-of-the-art indexes and data layouts by orders of magnitude. We also demonstrate how to incorporate multiple instance-optimized database components into an end-to-end analytic database system that outperforms a well-tuned commercial cloud-based analytics system by up to 3×.

Thesis Supervisor: Tim Kraska
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would first like to thank Tim Kraska, my advisor. As an incoming PhD student, Tim enthusiastically welcomed me into his research group, and what struck me was how much trust he immediately put in me. Even before I arrived at MIT, and before he had even started working with me, Tim arranged for me to do an internship at Microsoft Research, which kickstarted my path down my current research focus and led to years of fruitful collaboration. When I arrived at MIT in the fall, Tim placed me on an important project. This trust, and the opportunities that Tim opened for me, inspired me to do great work. Tim also taught me to think big, which was an invaluable skill in an age where many potential research directions feel like incremental work. By observing how he crafted the introductions to papers and talks, and by absorbing the feedback that he gave on my writing and presentations, I learned how to properly frame my research and communicate it to others. At the same time, Tim was always encouraging and supportive. Even after my first paper was rejected twice, and after I had failed to submit another paper to two consecutive deadlines, Tim remained supportive and understanding.

I would also like to thank Mohammad Alizadeh and Sam Madden, who have both been advisors on my projects and were on my thesis committee. They have both provided invaluable advice and feedback, both on research topics and on career directions. I appreciated seeing their perspectives and the way they approach research, which complemented Tim's and gave me even more insight into the qualities of great research.

I am grateful for the many students and postdocs at MIT with whom I had the pleasure of collaborating. I would especially like to thank Vikram Nathan, with whom I coauthored several papers. Vikram and I started working closely together almost immediately after I arrived at MIT. Having a collaborator certainly eased my transition into the PhD, and since he was a more senior student, I was able to learn a lot from the way he approached brainstorming, research meetings, and paper writing. Late nights at the office and paper rejections were much more tolerable with someone else to experience them with. I would also like to thank my other collaborators throughout the years: Andreas Kipf, Ryan Marcus, Kapil Vaidya, Ani Nrusimha, Siva Sudhir, Amadou Ngom, Darryl Ho, and Nesime Tatbul.

I am also grateful for my many collaborators outside MIT. I would especially like to thank Umar Farooq Minhas, with whom I spent two very happy and productive summers at Microsoft Research (though unfortunately the second summer had to be virtual). I would also like to thank the rest of the team at MSR, Donald Kossmann, Johannes Gehrke, Dave Lomet, Badrish Chandramouli, Chi Wang, Yinan Li, and Jaeyoung Do; the other interns I worked with at MSR, Hantian Zhang, Jia Yu, and Kyle Deeds; and my collaborators at other institutions, Tianzheng Wang, Eric Lo, Baotong Lu, Sanchit Misra, Alex van Renen, and Varun Pandey. Finally, I would like to thank Edward Gan and Peter Bailis for guiding me when I first began doing research as an undergrad.

I would like to thank all my wonderful labmates in the Data Systems Group at MIT, for their friendship and for the memories that we have made together: in addition

to those I've mentioned before, Anil, Yi, Wenbo, Favyen, Oscar, Joana, Matt, Zeyuan, Tianyu, Anna, Markos, Geoffrey, Eugenie, Ferdi, Ziniu, Xinjing, Pascal, Dominik, Emmanuel, Xiangyao, Raul, Lei, Ibrahim, Brit, and Laurent. Spending time with them, whether it was during our daily lunches or grabbing dinner and watching a movie or going on weekend trips, made my four years at MIT so much more vibrant and fulfilling.

Finally, I would like to thank my friends, my brother, and my parents for their constant support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database systems are the backbone of any modern enterprise. They power increasingly diverse business-critical use cases, ranging from maintaining real-time inventory metrics for e-commerce companies to supporting business intelligence tools that enable data-driven decision making. At the same time, modern enterprises are faced with an exponentially-increasing deluge of data, with the amount of data created worldwide increasing from 2 zettabytes in 2010 to nearly 100 zettabytes in 2022[1]. To keep pace with the prolific growth of both use cases and data volumes, modern database systems must continue to push the boundaries of performance.

One impactful method of achieving high performance in database systems is by optimizing their physical design, which refers to the physical data structures that represent the data stored within a database. A physical design that is optimized for the database's intended usage patterns can dramatically improve performance. For example, over the past decade analytic databases have overwhelmingly adopted columnar storage over traditional row-based storage, which can reduce scan costs for analytic queries that typically only access a subset of a table's columns; this demonstrates that the way in which the data itself is laid out on physical storage has an impact on performance. As another example, auxiliary data structures such as indexes and materialized views can improve performance for specific data access patterns, such as point lookups over a primary key column or a specific grouped aggregation.

The physical designs of modern database systems are typically general purpose: they achieve adequate performance for the average use case, but don't achieve optimal performance for any individual use case (Fig. 1-1). Database systems allow users to make decisions about the physical design of their specific database instance by exposing configuration knobs that the user can tune to achieve better performance for their specific use case. For example, the user can decide which column to sort each table by, which columns to build an index over, and what materialized views to create. However, tuning complex database systems is labor-intensive, and poor tuning can result in degraded performance and increased costs.

To alleviate the burden that manual tuning places on the user, in the late 1990's

[1] `https://www.statista.com/statistics/871513/worldwide-data-created/`

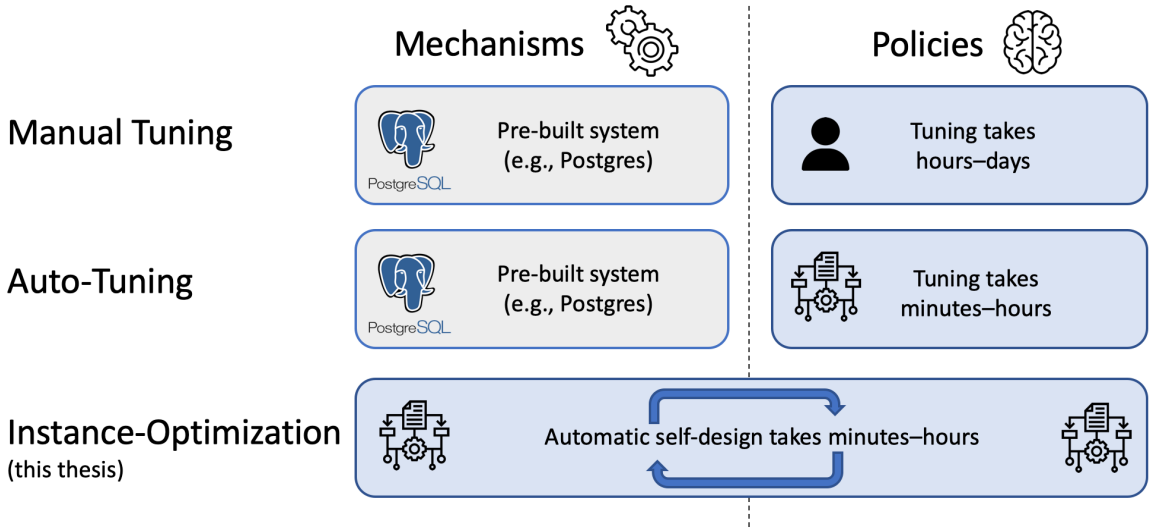

Figure 1-1: How can enterprises handle new data management use cases while maintaining high performance? Manual tuning can achieve some performance improvements but is labor-intensive and error-prone. Auto-tuning requires less manual effort but is still limited by the fact that it is applied to pre-built, often general-purpose, database systems that have a fixed set of mechanisms. Instance-optimization, which is the focus of this thesis, involves the careful co-design of both highly customizable mechanisms and policies for automatically specializing those mechanisms to achieve high performance for any given use case.

and early 2000's, commercial database systems such as Microsoft SQL Server made a concerted effort to automate certain tuning tasks, such as the selection of which indexes and materialized views to create [26, 5, 6]. They introduced *auto-tuning* tools (Fig. 1-1) that observe the user's interactions with the database to suggest configuration knob settings and physical design decisions to the user that improve performance. For example, the tool might observe that the user often retrieves data from a table that tracks shipments based on the date of the shipment, and therefore suggest to build an index over the ship-date column. Over the past decades, these auto-tuning tools have matured and are a part of many large-scale commercial database systems [7, 8].

However, the performance improvements that are achievable by these auto-tuning tools *depend* on the physical design mechanisms that are supported by the underlying database system. Fundamentally, a physical design mechanism that has more degrees of freedom has a larger number of possible configurations, and therefore it is more likely that for a specific use case, a specific well-chosen configuration can achieve better performance compared to a mechanism with fewer degrees of freedom that has fewer configurations to choose from.

The problem is that most database systems are designed with the goal of minimizing the complexity that is exposed to the user. Therefore, they typically expose a minimal set of configuration knobs and decision decisions for the user to make, in order to reduce the tuning burden on the user. Auto-tuning tools typically only tune this minimal set of configuration knobs. The limited degrees of freedom limits the ability

to specialize for a specific use case, and therefore limits the performance improvements achievable by tuning. In essence, the problem is that the mechanisms (i.e., the space of possible physical designs) were designed independently of the policies (i.e., the methods for searching in that space of designs).

A recent trend towards *instance-optimized* database components aims to improve the ability to specialize to a given use case. These instance-optimized components significantly expand their degrees of freedom far beyond what traditional database components are capable of, which therefore increases their configurability to any specific use case. For a given specific use case, instance-optimized database components can automatically find the best configuration from this much larger design space, often using machine learning or other optimization techniques. It is important to note that instance-optimization is not a competitor to auto-tuning, but rather the natural evolution of auto-tuning into a more powerful form that is not constrained by a pre-defined, often minimal, configuration space.

This thesis explores how instance-optimization can be used to improve the performance of database physical designs—with a focus on indexes, storage layouts, and materialized views—while requiring minimal manual effort from end users. In the remainder of this chapter, we more formally present the concept of instance-optimization, especially in the context of physical database design, then introduce the contributions of this thesis.

1.1 What is Instance-Optimization, Really?

Many existing publications on instance-optimization, including some of our own, loosely define instance-optimized systems as those that are designed to automatically self-adjust in order to achieve the best performance for a specific use case, i.e., a dataset and query workload. However, this definition is too broad and vague to be useful. By this definition, auto-tuning can be considered a form of instance-optimization because it is able to automatically self-adjust configuration knobs for a given dataset and workload. We now present a new definition that is more narrow in scope but is also more precise and gives us a better framework for reasoning about what constitutes a instance-optimized database and what doesn't.

Instance-optimization is often defined in terms of the capabilities of the component, e.g., the instance-optimized storage layout is able to adapt to the dataset and workload and therefore achieves better performance for a specific use case. We instead choose to define instance-optimization in terms of the process of designing the component itself. An instance-optimized design process involves two parts: first, we aim to expand the design space of its mechanisms to be able to adapt to any given use case, i.e., any given dataset and workload. In the extreme case, an instance-optimized database should support all possible mechanisms and should therefore be infinitely configurable, but this is not achievable in practice, and so instance-optimized systems must carefully build its mechanisms to be configurable along the most impactful degrees of freedom for adapting to different datasets and workloads. In other words, unlike most modern database systems, which aim to reduce complexity by reducing the number of tunable

configuration knobs, instance-optimized systems typically expose more knobs and parameters, which expands the design space and therefore increases the degrees of freedom with which to optimize for a certain use case.

Second, we simultaneously must create policies for navigating this larger design space of mechanisms to find the configuration that optimizes performance for a specific use case. In other words, it should have an extremely powerful auto-tuner that can select a configuration from the extremely large space of possible database designs. In summary:

> ***Instance-optimization*** *is the process of co-designing (1) a mechanism with enough degrees of freedom to allow for fine-grained customization to any particular use case and (2) a policy, or optimization algorithm, for automatically specializing the mechanism to a specific use case, i.e., a well-defined dataset and workload.*

It is also important to point out that the idea of instance-optimization is often incorrectly used synonymously with the idea of "ML for systems," that is, applying machine learning techniques to improve database systems. For example, ML can be used to improve auto-tuning algorithms for configuring knobs, but it would not qualify as instance-optimization because there is no co-design of mechanisms and policies. Furthermore, this false equivalence is misleading because instance-optimization does not necessarily involve ML. Instance-optimized systems do require efficient algorithms for navigating a expanded design space of mechanisms, which are often based on machine learning (ML), but not necessarily. In the end, ML is simply a means to an end, not the end in itself. Even when an instance-optimization technique uses ML, we generally prefer to call it instance-optimized, to place more focus on the goal rather than the method for achieving that goal.

1.2 Thesis Statement and Contributions

In this thesis, we make the following claim:

> ***Thesis Statement:*** *Instance-optimization of the physical design of a database system, through careful co-design of highly configurable mechanisms and policies for automatic specialization to a given use case, improves the performance of database systems and their ability to adapt to new use cases.*

We explore instance-optimization in the context of several fundamental components of database physical design: indexes, data storage layouts, and materialized views.

1.2.1 Indexes

Database indexes are data structures that aim to improve the speed of retrieving data from a database. One of the most fundamental database indexes is the B+ Tree,

which is constructed over an array of n key-value pairs that are sorted by the key, can perform a lookup of any given key in $O(\log n)$ time, and takes $O(n)$ storage space. The B+ Tree is general-purpose and has been used in commercial databases for many decades. However, the generality of the B+ Tree comes at a cost. In some cases, knowledge of the data helps improve performance. As an extreme example, if the keys are consecutive integers, we can store the data in an array and perform lookup in $O(1)$ time.

Indeed, Kraska et al. [100] observed that B+ Tree indexes can be thought of as models. Given a key, they predict the location of the key-value pair within the sorted array. If indexes are models, they can be learned using traditional ML techniques by learning the cumulative distribution function (CDF) of the input data. The resulting *Learned Index* [100] is optimized for the specific data distribution, and empirically outperforms a traditional B+ Tree index over the same data distribution in both lookup time and space usage.

The Learned Index is an instance-optimized data structure because it satisfies both properties described in Section 1.1: first, its mechanism is vastly more configurable than what can be found in traditional systems. Typical indexes such as the B+ Tree only have a few user-tunable parameters, such as the page size. On the other hand, a Learned Index is fundamentally a ML model, which can approximate any continuous function [33]. Furthermore, the Learned Index has algorithms for (semi-)automatically[2] configuring their ML model based on the use case, i.e., the specific data that is being index.

However, the main drawback of the Learned Index is that it does not support any data modifications (i.e., writes), including inserts, updates, or deletes. This critical drawback makes the Learned Index unusable for the vast majority of real-world workloads, which are composed of a combination of reads and writes. In Chapter 2, we introduce **ALEX** [46], which is an updatable learned index that dynamically adjusts both its models and its structure in the presence of data modifications, while maintaining its speed and space advantages over non-learned indexes. ALEX is built on several core ideas:

- **Storage layout optimized for models:** To store sorted key-value records, ALEX uses an array with gaps, a *Gapped Array*, which (1) amortizes the cost of shifting the keys for each insertion because gaps can absorb inserts, and (2) allows more accurate placement of data using *model-based inserts* to ensure that records are located closely to the predicted position when possible.

- **Search strategy optimized for models:** ALEX exploits model-based inserts combined with *exponential search* starting from the predicted position.

- **Adaptive tree structure:** ALEX provides robust performance even when the data distribution is skewed or dynamically changes after index initialization. ALEX achieves this by exploiting adaptive expansion and node splitting mecha-

[2] In the original publication [100], the user does need to manually tune the number of second-level models, which has a large impact on performance.

nisms, paired with selective model retraining, which is triggered by intelligent policies based on simple cost models.

1.2.2 Storage Layouts

Indexes such as B+ Trees and ALEX are designed to support workloads with frequent data modifications and lookups for either a single key or short range of key values. These types of workloads are common in operational (OLTP) use cases, where databases must excel at efficiently reading and writing individual rows of data.

On the other hand, analytic (OLAP) use cases often involve queries that compute an aggregation over a large number of data records. In addition, data modification operations are less frequent, and when they do occur, data records are typically modified in large batches rather than individually. To efficiently process increasingly larger volumes of data, modern data analytics services use a variety of techniques to reduce data access, i.e., the amount of data that must be read during query processing. For example, one standard technique is to use columnar storage to avoid accessing columns that are not relevant to a query.

An extremely impactful technique for reducing data access is to adjust the data storage layout, i.e., the way in which data records are ordered and clustered on physical storage, whether that be in memory, on disk, or in cloud object stores. Since filtering data based on predicates is one of the most fundamental operations for any modern analytic database system, a good storage layout separates filtered records from non-filtered records, and therefore allows the database to minimize the amount of data read during query processing. The challenge is that every query in the user's workload uses potentially different filters, so the set of filtered and non-filtered records is different for each query, and the data storage layout should perform well across the entire user workload.

The most commonly-used data storage technique in practice is to sort all data records in a table by their value in a certain column, called the sort key. Additionally, some database support specialized multi-column sort orders (e.g., Z-order) or multi-dimensional indexes. However, these data layout schemes are hard to tune and their performance is inconsistent. Our own recent work on learned multi-dimensional indexes [137] has introduced the idea of instance-optimizing the storage layout. However, the performance of that work suffers in the presence of correlated data and skewed query workloads, both of which are common in real applications.

Therefore, in Chapter 3 we introduce **Tsunami** [48], an instance-optimized data storage layout that handles correlated data and skewed query workloads. First, Tsunami achieves high performance on skewed query workloads by using a lightweight decision tree, called a Grid Tree, to partition space into non-overlapping regions in a way that reduces query skew. Second, Tsunami achieves high performance on correlated datasets by indexing each region using an Augmented Grid, which uses two techniques—*functional mappings* and *conditional CDFs*—to efficiently capture information about correlations. Finally, Tsunami has an optimization algorithm that automatically configures the Grid Tree and Augmented Grid structures for a given use case. Tsunami achieves up to 11× faster query performance than optimally-tuned

traditional data layouts, as well as up to 6× faster query performance than the previous state-of-the-art instance-optimized data layout.

However, Tsunami can only optimize a single table's storage layout, for a query workload that only queries that table. In practice, analytics workloads typically contain many tables and the queries use diverse join patterns, such as in a star or snowflake schema. Furthermore, Tsunami is designed to handle data that resides fully in memory, whereas large datasets nowadays are often stored on disk or in cloud objects stores such as Amazon S3. Therefore, in Chapter 4 we introduce **MTO** (Multi-Table Optimizer) [45], the first instance-optimized storage layout framework for jointly optimizing the storage layouts of all tables in disk-based or cloud-based multi-table datasets. Our key idea is to pass additional information about joins through *join-induced predicates*, to jointly optimize the layout for all tables, simultaneously. Furthermore, existing instance-optimized layout techniques must re-optimize the entire layout in response to changes in the query workload. In contrast, MTO gracefully responds to workload changes through partial layout reorganization. Experiments on a commercial cloud-based analytics service show that MTO achieves up to 93% reduction in data accessed and 75% reduction in end-to-end query times compared to state-of-the-art data layout strategies.

1.2.3 Synthesis Into a Complete System

Instance-optimized physical design components have largely been designed and evaluated in isolation, and there have only been a few efforts to integrate them into an end-to-end system. It is unclear how multiple instance-optimized components would work together in concert. In fact, it is easy to imagine a number of learned components destructively interfering with each other. Is it possible to build a system that autonomously custom-tailors its major components to the user's requirements, approaching the performance of a bespoke system but with similar ease of use as a general-purpose system?

To the best of our knowledge, there is no end-to-end data system built with instance-optimization as a foundational design principle. As a case study and a first attempt at synthesizing the rich space of research on instance-optimized database components, in Chapter 5 we introduce **SageDB** [44], an instance-optimized data analytics system, and show how two carefully selected components can work together in practice. These instance-optimized components are (1) data storage layouts combined with data replication, which draws on the ideas of Chapters 3 and 4, and (2) partial materialized views, which are a generalization of traditional materialized views with more degrees of freedom. These techniques minimize I/O when scanning data from disk and maximize computation reuse through intelligent pre-materialization of partial results. For a given dataset and workload, SageDB uses a global optimization algorithm to automatically and simultaneously configure all instance-optimized components. Our SageDB prototype is a single-node database that stores data on disk, backs up data on the cloud, and uses a parallelized pipeline-based execution engine. Our prototype outperforms a commercial cloud-based analytics system by up to 3× on end-to-end query workloads and up to 250× on individual queries.

1.2.4 Opportunities and Limitations

While the work presented in this thesis demonstrates the opportunities of instance-optimized database systems, more work remains before instance-optimization techniques can be widely adopted in production-ready systems. We will describe these limitations and the potential for future work in detail in Section 6.2. Here, we highlight some key limitations at a high level.

While ALEX shows how learned indexes can thrive in dynamic environments, it is nonetheless still a prototype. Compared to the B+ Tree, which is often preferred because of its suitability for data on persistent storage and its support of concurrency, ALEX's design is optimized for data stored in memory, and it is single-threaded. ALEX has served as the blueprint for follow-on work, by both ourselves and others, that addresses these limitations. APEX [113] adapts ALEX to perform well for data that is stored on persistent memory, and ALEX+ [191] is a concurrent version of ALEX that employs a simplified version of APEX's concurrency protocol.

Together, Tsunami and MTO improve both the performance and scope of instance-optimized data storage layouts. However, one limitation of both techniques, and of existing instance-optimized data layouts techniques more broadly, is that since they optimize their design for a specific dataset and workload, they may perform poorly for datasets and workloads that frequently change. For highly dynamic environments, it may not make sense to even use instance-optimized layouts.

There is also room to further improve SageDB's design and performance. First, while we aimed to automate SageDB's operation as much as possible, the user still has one important responsibility: deciding when to manually trigger SageDB's optimization algorithms. Ideally, even this responsibility should be removed from the user, and SageDB's optimization functionality should be fully autonomous. Second, as was the case with instance-optimized layouts, SageDB is not designed to maintain high performance for highly dynamic datasets and workloads. Furthermore, SageDB currently only synthesizes instance-optimized components related to physical design, but there has also been significant work on instance-optimized query optimizers, cardinality estimators, and cost models that have not yet been incorporated into SageDB. These limitations point to important directions for future work.

Chapter 2

ALEX: An Updatable Learned Index

Recent work by Kraska et al. [97], which we will refer to as the Learned Index, proposes to replace a standard database index with a hierarchy of machine learning (ML) models. Given a key, an intermediate node in the hierarchy is a model to predict the child model to use, and a leaf node in this hierarchy is a model to predict the location of the key in a densely packed array (Fig. 2-1). The models for this Learned Index are trained from the data. Their key insight is that using (even simple) models that adapt to the data distribution to make a "good enough" guess of a key's actual location significantly improves performance. However, their solution can only handle lookups on read-only data, with no support for update operations. This critical drawback makes the Learned Index unusable for dynamic, read-write workloads, common in practice.

In this chapter, we start by asking ourselves the following research question: *Can we design a new high performance index for dynamic workloads that effectively combines the core insights from the Learned Index with proven storage & indexing techniques to deliver great performance in both time and space?* Our answer is a new in-memory index structure called ALEX, a fully dynamic data structure that simultaneously provides efficient support for point lookups, short range queries, inserts, updates, deletes, and bulk loading. This mix of operations is commonplace in online transaction processing (OLTP) workloads [30, 179, 164] and is also supported by B+Trees [159].

Implementing writes with high performance requires a careful design of the underlying data structure that stores records. [97] uses a sorted, densely packed array which works well for static datasets but can result in high costs for shifting records if new records are inserted. Furthermore, the prediction accuracy of the models can deteriorate as the data distribution changes over time, requiring repeated retraining. To address these challenges, we make the following technical contributions in this chapter:

- **Storage layout optimized for models:** Similar to a B+Tree, ALEX builds a tree, but allows different nodes to grow and shrink at different rates. To store records in a data node, ALEX uses an array with gaps, a *Gapped Array*, which (1) amortizes the cost of shifting the keys for each insertion because gaps can absorb inserts, and (2) allows more accurate placement of data using *model-based*

inserts to ensure that records are located closely to the predicted position when possible. For efficient search, gaps are actually filled with adjacent keys.

- **Search strategy optimized for models:** ALEX exploits model-based inserts combined with *exponential search* starting from the predicted position. This always beats binary search when models are accurate.

- **Keeping models accurate with dynamic data distributions and workloads:** ALEX provides robust performance even when the data distribution is skewed or dynamically changes after index initialization. ALEX achieves this by exploiting adaptive expansion, and node splitting mechanisms, paired with selective model retraining, which is triggered by intelligent policies based on simple cost models. Our cost models take the actual workload into account and thus can effectively respond to dynamic changes in the workload. ALEX achieves all the above benefits without needing to hand-tune parameters for each dataset or workload.

- **Detailed evaluation:** We present the results of an extensive experimental analysis with real-life datasets and varying read-write workloads and compare against state of the art indexes that support range queries.

On read-only workloads, ALEX beats the Learned Index by up to 2.2× on performance with up to 15× smaller index size. Across the spectrum of read-write workloads, ALEX beats B+Tree by up to 4.1× while never performing worse, with up to 2000× smaller index size. ALEX also beats an ML-enhanced B+Tree and the memory-optimized Adaptive Radix Tree, scales to large data sizes, and is robust to data distribution shift.

ALEX is a key step towards making learned indexes practical for a broader class of database workloads with dynamic updates. The initial design presented in this chapter focused on in-memory datasets and single-threaded execution. After its initial publication, follow-up work has addressed how to adapt the ALEX design to data stored on persistent storage mediums [113] and multi-threaded execution [191].

In the remainder of this chapter, we give background (Section 2.1), present the architecture of ALEX (Section 2.2), describe the operations on ALEX (Section 2.3), present an analysis of ALEX performance (Section 2.4), present experimental results (Section 2.5), review related work (Section 2.6), discuss extensions to our design (Section 2.7), and conclude (Section 2.8).

2.1 Background

2.1.1 Traditional B+Tree Indexes

B+Tree is a classic range index structure. It is a height-balanced tree which stores either the data (primary index) or pointers to the data (secondary index) at the leaf level, in a sorted order to facilitate range queries.

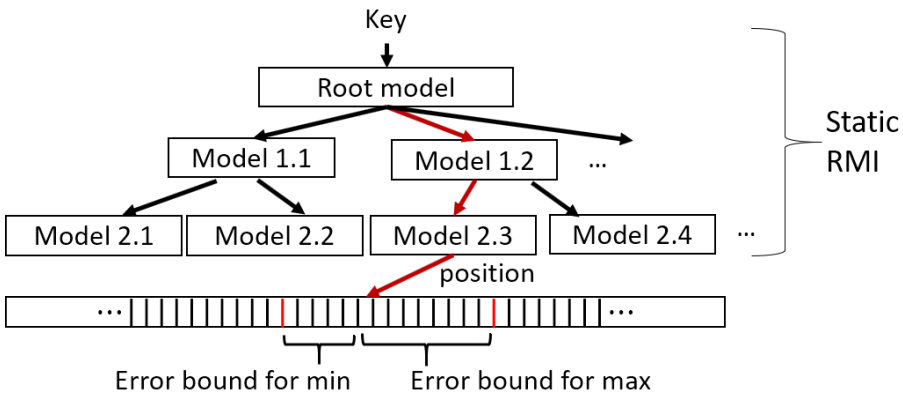

Figure 2-1: Learned Index by Kraska et al.

A B+Tree lookup operation can be broken down into two steps: (1) traverse to leaf, and (2) search within the leaf. Starting at the root, traverse to leaf performs comparisons with the keys stored in each node, and branches via stored pointers to the next level. When the tree is deep, the number of comparisons and branches can be large, leading to many cache misses. Once traverse to leaf identifies the correct leaf page, typically a binary search is performed to find the position of the key within the node, which might incur additional cache misses.

The B+Tree is a dynamic data structure that supports inserts, updates, and deletes; is robust to data sizes and distributions; and is applicable in many different scenarios, including in-memory and on-disk. However, the generality of B+Tree comes at a cost. In some cases knowledge of the data helps improve performance. As an extreme example, if the keys are consecutive integers, we can store the data in an array and perform lookup in O(1) time. A B+Tree does not exploit such knowledge. Here, "learning" from the input data has an edge.

2.1.2 The Case for Learned Indexes

Kraska et al. [97] observed that B+Tree indexes can be thought of as models. Given a key, they predict the location of the key within a sorted array (logically) at the leaf level. If indexes are models, they can be learned using traditional ML techniques by learning the cumulative distribution function (CDF) of the input data. The resulting *Learned Index* is optimized for the specific data distribution.

Another insight from Kraska et al. is that a single ML model learned over the entire data is not accurate enough because of the complexity of the CDF. To overcome this, they introduce the *recursive model index* (*RMI*) [97]. RMI is a hierarchy of models, with a *static* depth of two or three, where a higher-level model picks the model at the next level, and so on, with the leaf-level model making the final prediction for the position of the key in the data structure (Fig. 2-1). Logically, the RMI replaces the internal B+Tree nodes with models. The effect is that comparisons and branches in internal B+Tree nodes during traverse to leaf are replaced by model inferences in a Learned Index.

In [97], the keys are stored in an in-memory sorted array. Given a key, the leaf-level model predicts the position (array index) of the key. Since the model is not perfect, it could make a wrong prediction. The insight is that if the leaf model is accurate, a local search surrounding the predicted location is faster than a binary search on the entire array. To support local search, [97] keeps *min* and *max* error bounds for each model in RMI and performs binary search within these bounds.

Last, each model in RMI can be a different type of model. Both linear regression and neural network based models are considered in [97]. There is a trade-off between model accuracy and model complexity. The root of the RMI is tuned to be either a neural network or a linear regression, depending on which provides better performance, while the simplicity and the speed of computation for linear regression model is beneficial at the non-root levels. A linear regression model can be represented as $y = \lfloor a * x + b \rfloor$, where x is the key and y is the predicted position. A linear regression model needs to store just two parameters a and b, so storage overhead is low. The inference with a single linear regression model requires only one multiplication, one addition and one rounding, which are fast to execute on modern processors.

Unlike B+Tree, which could have many internal levels, RMI uses two or three levels. Also, the storage space required for models (two or four 8-byte double values per model) is much smaller than the storage space for internal nodes in B+Tree (which store keys and pointers). A Learned Index can be an order of magnitude smaller in main memory storage (vs. internal B+Tree nodes), while outperforming a B+Tree in lookup performance by a factor of up to three [97].

The main drawback of the Learned Index is that it does not support any modifications, including inserts, updates, or deletes. Let us demonstrate a naïve insertion strategy for such an index. Given a key k to insert, we first use the model to find the insertion position for k. Then we create a new array whose length is one plus the length of the old array. Next, we copy the data from the old array to the new array, where the elements on the right of the insertion position are shifted to the right by one position. We insert k at the insertion position of the new array. Finally, we update the models to reflect the change in the data distribution. Such a strategy has a linear time complexity with respect to the data size, which is unacceptable in practice. Kraska et al. suggest building delta-indexes to handle inserts [97], which is complementary to our strategy. In this chapter, we describe an alternative data structure to make modifications in a learned index more efficient.

2.2 ALEX Overview

The ALEX design (Fig. 2-2) takes advantage of two key insights. First, we propose a careful space-time trade-off that not only leads to an updatable data structure, but is also faster for lookups. To explore this trade-off, ALEX supports a *Gapped Array (GA)* layout for the leaf nodes, which we present in Section 2.2.2. Second, the Learned Index supports static RMI (SRMI) only, where the number of levels and the number of models in each level is fixed at initialization. SRMI performs poorly on inserts if the data distribution is difficult to model. ALEX can be updated dynamically and

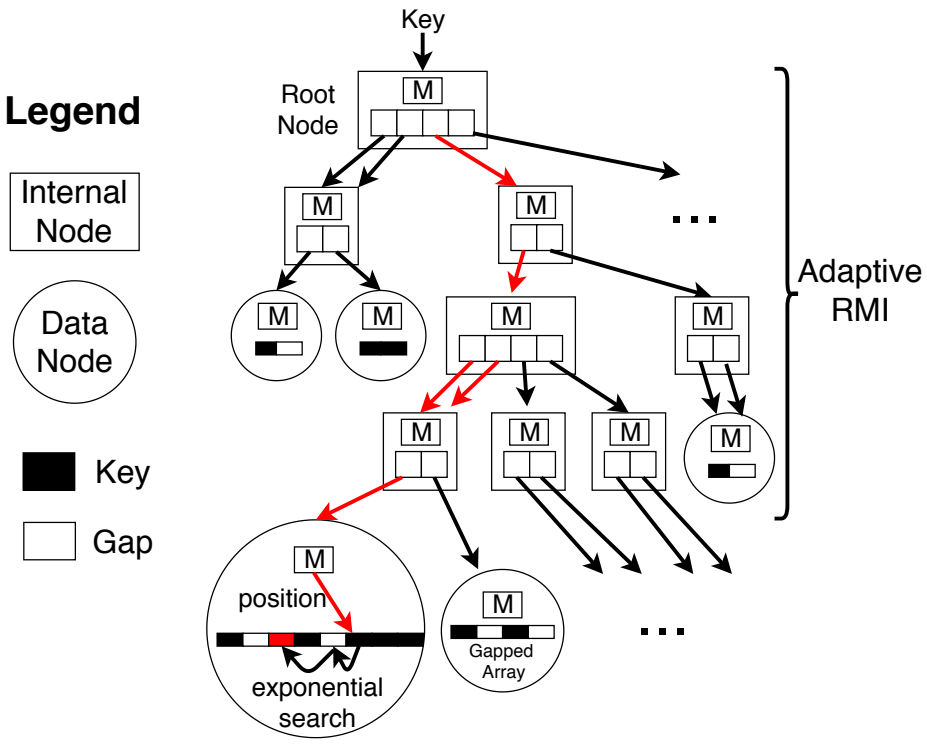

Figure 2-2: ALEX Design

efficiently at runtime and uses linear cost models that predict the latency of lookup and insert operations based on simple statistics measured from an RMI. ALEX uses these cost models to initialize the RMI structure and to dynamically adapt the RMI structure based on the workload.

ALEX aims to achieve the following goals w.r.t. the B+Tree and Learned Index. (1) Insert time should be competitive with B+Tree, (2) lookup time should be faster than B+Tree and Learned Index, (3) index storage space should be smaller than B+Tree and Learned Index (4) data storage space (leaf level) should be comparable to dynamic B+Tree. In general, data storage space will overshadow index storage space, but the space benefit from smaller index storage space is still important because it allows more indexes to fit into the same memory budget. The rest of this section describes how our ALEX design achieves these goals.

2.2.1 Design Overview

ALEX is an in-memory, updatable learned index. ALEX has a number of differences from the Learned Index [97].

The first difference lies in the data structure used to store the data at the leaf level. Like B+Tree, ALEX uses a *node per leaf*. This allows the individual nodes to expand and split more flexibly and also limits the number of shifts required during an insert. In a typical B+Tree, every leaf node stores an array of keys and payloads and has "free space" at the end of the array to absorb inserts. ALEX uses a similar design but more carefully chooses how to use the free space. The insight is that by

introducing gaps that are strategically placed between elements of the array, we can achieve faster insert and lookup times. As shown in Fig. 2-2, ALEX uses a Gapped Array (GA) layout for each data node, which we describe in Section 2.2.2.

The second difference is that ALEX uses exponential search to find keys at the leaf level to correct mispredictions of the RMI, as shown in Fig. 2-2. In contrast, [97] uses binary search within the error bounds provided by the models. We experimentally verified that exponential search without bounds is faster than binary search with bounds (Section 2.5.3). This is because if the models are good, their prediction is close enough to the correct position. Exponential search also removes the need to store error bounds in the models of the RMI.

The third difference is that ALEX inserts keys into data nodes at the position where the models predict that the key should be. We call this *model-based insertion*. In contrast, the Learned Index produces an RMI on an array of records without changing the position of records in the array. Model-based insertion has better search performance because it reduces model misprediction errors.

The fourth difference is that ALEX dynamically adjusts the shape and height of the RMI depending on the workload. We describe the design of initializing and dynamically growing the RMI structure in Section 2.3.

The final difference is that ALEX has no parameters that need to be re-tuned for each dataset or workload, unlike the Learned Index, in which the number of models must be tuned. ALEX automatically bulk loads and adjusts the structure of RMI to achieve high performance by using a cost model.

2.2.2 Node Layout

Data Nodes

Like a B+Tree, the leaf nodes of ALEX store the data records and thus are referred to as *data nodes*, shown as circles in Fig. 2-2. A data node stores a linear regression model (two double values for slope and intercept), which maps a key to a position, and two Gapped Arrays (described below), one for *keys* and one for *payloads*. We show only the keys array in Fig. 2-2. By default, both keys and payloads are fixed-size. (Note that payloads could be records or pointers to *variable-sized* records, stored in separately allocated spaces in memory). We also impose a *max node size* for practical reasons (see details in Section 2.3).

ALEX uses a *Gapped Array* layout which uses model-based inserts to distribute extra space between the elements of the array, thereby achieving faster inserts and lookups. In contrast, B+Tree places all the gaps at the end of the array. Gapped Arrays fill the gaps with the closest key to the right of the gap, which helps maintain exponential search performance. In order to efficiently skip gaps when scanning, each data node maintains a bitmap which tracks whether each location in the node is occupied by a key or is a gap. The bitmap is fast to query and has low space overhead compared to the Gapped Array. We compare Gapped Array to an existing gapped data structure called Packed Memory Array [16] in Appendix A.7.

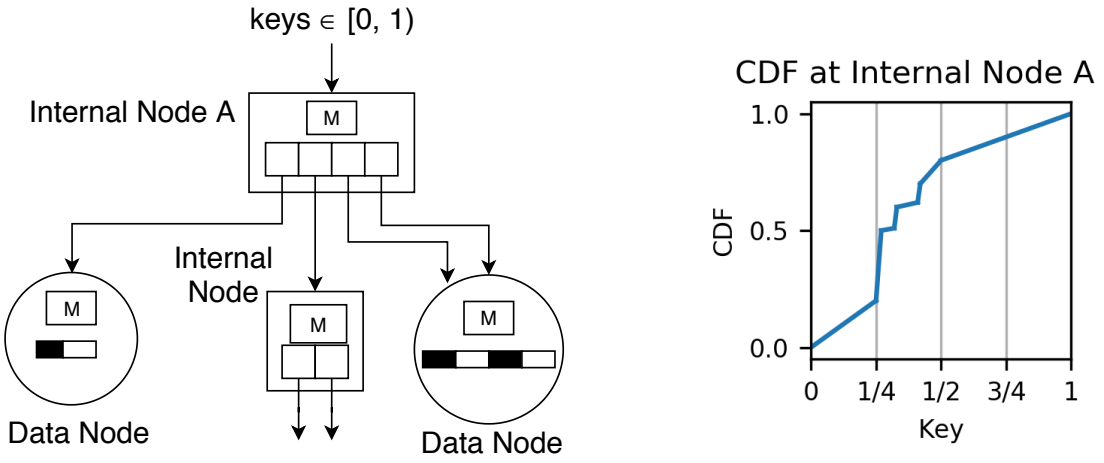

Figure 2-3: Internal nodes allow different resolutions in different parts of the key space $[0, 1)$.

Internal Nodes

We refer to all the nodes which are part of the RMI structure as *internal nodes*, shown as rectangles in Fig. 2-2. Internal nodes store a linear regression model and an array containing pointers to children nodes. Like a B+Tree, internal nodes direct traversals down the tree, but unlike B+Tree, internal nodes in ALEX use models to "compute" the location, in the pointers array, of the next child pointer to follow. Similar to data nodes, we impose a *max node size*.

The internal nodes of ALEX serve a conceptually different purpose than those of the Learned Index. Learned Index's internal nodes have models that are fit to the data; an internal node with a perfect model partitions keys equally to its children, and an RMI with perfect internal nodes results in an equal number of keys in each data node. However, the goal of the RMI structure is not to produce equally sized data nodes, but rather data nodes whose key distributions are roughly linear, so that a linear model can be accurately fit to its keys.

Therefore, the role of the internal nodes in ALEX is to provide a flexible way to partition the key space. Suppose internal node A in Fig. 2-3 covers the key space $[0, 1)$ and has four child pointers. A Learned Index would assign a node to each of these pointers, either all internal nodes or all data nodes. However, ALEX more flexibly partitions the space. Internal node A assigns the key spaces $[0, 1/4)$ and $[1/2, 1)$ to data nodes (because the CDF in those spaces are linear), and assigns $[1/4, 1/2)$ to another internal node (because the CDF is non-linear and the RMI requires more resolution into this key space). As shown in the figure, multiple pointers can point to the same child node; this is useful for handling inserts (Section 2.3.3). We restrict the number of pointers in every internal node to always be a power of 2. This allows nodes to split without retraining its subtree (Section 2.3.3).

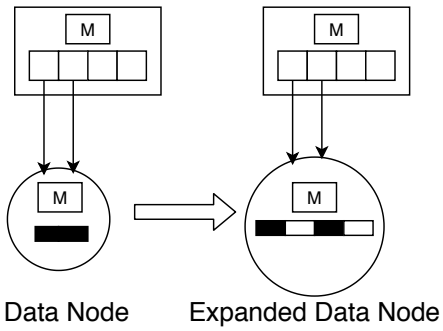

Figure 2-4: Node Expansion

2.3 ALEX Algorithms

In this section, we describe the algorithms for lookups, inserts (including how to dynamically grow the RMI and the data nodes), deletes, out of bounds inserts, and bulk load.

2.3.1 Lookups and Range Queries

To look up a key, starting at the root node of the RMI, we iteratively use the model to "compute" a location in the pointers array, and we follow the pointer to a child node at the next level, until we reach a data node. By construction, the internal node models have perfect accuracy, so there is no search involved in the internal nodes. We use the model in the data node to predict the position of the search key in the *keys* array, doing exponential search if needed to find the actual position of the key. If a key is found, we read the corresponding value at the same position from the *payloads* array and return the record. Else, we return a null record. We visually show (using red arrows) a lookup in Fig. 2-2. A range query first performs a lookup to find the position and data node of the first key whose value is not less than the range's start value, then scans forward until reaching the range's end value, using the node's bitmap to skip over gaps and if necessary using pointers stored in the node to jump to the next data node.

2.3.2 Insert in non-full Data Node

For the insert algorithm, the logic to reach the correct data node (i.e., TraverseToLeaf) is the same as in the lookup algorithm described above. In a non-full data node, to find the insertion position for a new element, we use the model in the data node to predict the insertion position. If the predicted position is not correct (if inserting there would not maintain sorted order), we do exponential search to find the correct insertion position. If the insertion position is a gap, then we insert the element into the gap and are done. Else, we make a gap at the insertion position by shifting the elements by one position in the direction of the closest gap. We then insert the element into the newly created gap. The Gapped Array achieves $O(\log n)$ insertion time with high probability [15].

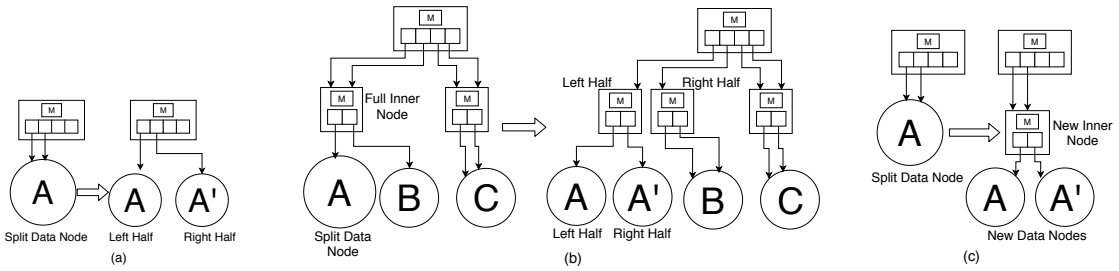

Figure 2-5: Node Splits

2.3.3 Insert in full Data Node

When a data node becomes full, ALEX uses two mechanisms to create more space: expansions and splits. ALEX relies on simple cost models to pick between different mechanisms. Below, we first define the notion of "fullness," then describe the expansion and split mechanisms, and the cost models. We then present the insertion algorithm that combines the mechanisms with the cost models. Algorithm 1 summarizes the procedure for inserting into a data node.

Criteria for Node Fullness

ALEX does not wait for a data node to become 100% full, because insert performance on a Gapped Array will deteriorate as the number of gaps decreases. We introduce lower and upper density limits on the Gapped Array: $d_l, d_u \in (0, 1]$, with the constraint that $d_l < d_u$. Density is defined as the fraction of positions that are filled by elements. A node is full if the next insert results in exceeding d_u. By default we set $d_l = 0.6$ and $d_u = 0.8$ to achieve average data storage utilization of 0.7, similar to B+Tree [67], which in our experience always produces good results and did not need to be tuned. In contrast, B+Tree nodes typically have $d_l = 0.5$ and $d_u = 1$. Section 2.4 presents a theoretical analysis of how the density of the Gapped Array provides a way to trade off between the space and the lookup performance for ALEX.

Node Expansion Mechanism

To expand a data node that contains n keys, we allocate a new larger Gapped Array with n/d_l slots. We then either scale or retrain the linear regression model, and then do model-based inserts of all the elements in this new larger node using the scaled or retrained model. After creation, the new data node is at the lower density limit d_l. Fig. 2-4 shows an example data node expansion where the Gapped Array inside the data node is expanded from two slots on the left to four slots on the right.

Node Split Mechanism

To split a data node in two, we allocate the keys to two new data nodes, such that each new node is responsible for half of the key space of the original node. ALEX supports two ways to split a node:

(1) *Splitting sideways* is conceptually similar to how a B+Tree uses splits. There are two cases: (a) If the parent internal node of the split data node is not yet at the *max node size*, we replace the parent node's pointers to the split data node with pointers to the two new data nodes. The parent internal node's pointers array might have redundant pointers to the split data node (Fig. 2-3). If so, we give half of the redundant pointers to each of the two new nodes. Else, we create a second pointer to the split data node by doubling the size of the parent node's pointers array and making a redundant copy for every pointer, and then give one of the redundant pointers to each of the two new nodes. Fig. 2-5a shows an example of a sideways split that does not require an expansion of the parent internal node. (b) If the parent internal node has reached *max node size*, then we can choose to split the parent internal node, as we show in Fig. 2-5b. Note that by restricting all the internal node sizes to be powers of 2, we can always split a node in a "boundary preserving" way, and thus require no retraining of any models below the split internal node. Note that the split can propagate all the way to the root node, just like in a B+Tree.

(2) *Splitting down* converts a data node into an internal node with two child data nodes, as we show in Fig. 2-5c. The models in the two child data nodes are trained on their respective keys. B+Tree does not have an analogous splitting down mechanism.

Cost Models

To make decisions about which mechanism to apply (expansion or various types of splits), ALEX relies on simple linear cost models that predict average lookup time and insert time based on two simple statistics tracked at each data node: (a) average number of exponential search iterations, and (b) average number of shifts for inserts. Lookup performance is directly correlated with (a) while insert performance is directly correlated with (a) and (b) (since an insert first needs to do a lookup to find the correct insertion position). These *intra-node* cost models predict the time to perform operations within a data node.

These two statistics are not known when creating a data node. To find the *expected cost* of a new data node, we compute the expected value of these statistics under the assumption that lookups are done uniformly on the existing keys, and inserts are done according to the existing key distribution. Specifically, (a) is computed as the average base-2 logarithm of model prediction error for all keys; (b) is computed as the average distance to the closest gap in the Gapped Array for all existing keys. These expected values can be computed without creating the data node. If the data node is created using a subset of keys from an existing data node, we can use the empirical ratio of lookups vs. inserts to weight the relative importance of the two statistics for computing the expected cost.

In addition to the intra-node cost model, ALEX uses a *TraverseToLeaf* cost model to predict the time for traversing from the root node to a data node. The TraverseToLeaf cost model uses two statistics: (1) the depth of the data node being traversed to, and (2) the total size (in bytes) of all inner nodes and data node metadata (i.e., everything except for the keys and payloads). These statistics capture the cost of traversal: deeper data nodes require more pointer chases to find, and larger size

will decrease CPU cache locality, which slows down the traversal to a data node. We provide more details about the cost models and show their low usage overhead in Appendix A.6.

Insertion Algorithm

As lookups and inserts are done on the data node, we count the number of exponential search iterations and shifts per insert. From these statistics, we compute the *empirical cost* of the data node using the intra-node cost model. Once the data node is full, we compare the expected cost (computed at node creation time) to the empirical cost. If they do not deviate significantly, then we conclude that the model is still accurate, and we perform node expansion (if the size after expansion is less than the *max node size*), scaling the model instead of retraining. The models in the internal nodes of the RMI are not retrained or rescaled. We define significant *cost deviation* as occurring when the empirical cost is more than 50% higher than the expected cost. In our experience, this cost deviation threshold of 50% always produces good results and did not need to be tuned.

Otherwise, if the empirical cost has deviated from the expected cost, we must either (i) expand the data node and retrain the model, (ii) split the data node sideways, or (iii) split the data node downwards. We select the action that results in lowest expected cost, according to our intra-node cost model. For simplicity, ALEX always splits a data node in two. The data node could conceptually split into any power of 2, but deciding the optimal fanout can be time-consuming, and we experimentally verified that a fanout of 2 is best according to the cost model in most cases.

Why would empirical cost deviate from expected cost?

This often happens when the distribution of keys that are inserted does not follow the distribution of existing keys, which results in the model becoming inaccurate. An inaccurate model may lead to long contiguous regions without any gaps. Inserting into these *fully-packed regions* requires shifting up to half of the elements within it to create a gap, which in the worst case takes $O(n)$ time. Performance may also degrade simply due to random noise as the node grows larger or due to changing access patterns for lookups.

2.3.4 Delete, update, and other operations

To delete a key, we do a lookup to find the location of the key, and then remove it and its payload. Deletes do not shift any existing keys, so deletion is a strictly simpler operation than inserts and does not cause model accuracy to degrade. If a data node hits the lower density limit d_l due to deletions, then we contract the data node (i.e., the opposite of expanding the data node) in order to avoid low space utilization. Additionally, we can use intra-node cost models to determine that two data nodes should merge together and potentially grow upwards, locally decreasing

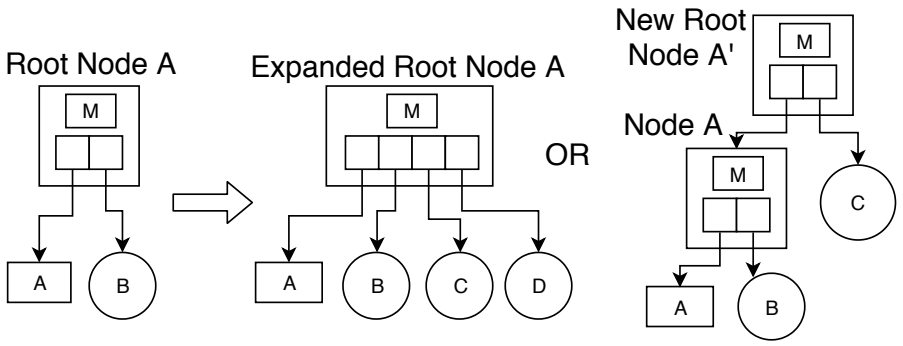

Figure 2-6: Splitting the root

the RMI depth by 1. However, for simplicity we do not implement these merging operations.

Updates that modify the key are implemented by combining an insert and a delete. Updates that only modify the payload will look up the key and write the new value into the payload. Like B+Trees, we can merge two ALEX indexes or find the difference between two ALEX indexes by iterating over their sorted keys in tandem and bulk loading a new ALEX index.

2.3.5 Handling out of bounds inserts

A key that is lower or higher than the existing key space would be inserted into the the left-most or right-most data node, respectively. A series of out-of-bounds inserts, such as an append-only insert workload, would result in poor performance because that data node has no mechanism to split the out-of-bounds key space. Therefore, ALEX has two ways to smoothly handle out-of-bounds inserts. Assume that the out-of-bounds inserts are to the right (e.g., inserted keys are increasing); we apply analogous strategies when inserts are to the left.

First, when an insert that is outside the existing key space is detected, ALEX will *expand the root node*, thereby expanding the key space, shown in Fig. 2-6. We expand the size of the child pointers array to the right. Existing pointers to existing children are not modified. A new data node is created for every new slot in the expanded pointers array. In case this expansion would result in the root node exceeding the max node size, ALEX will create a new root node. The first child pointer of the new root node will point to the old root node, and a new data node is created for every other pointer slot of the new root node. At the end of this process, the out-of-bounds key will fall into one of the newly created data nodes.

Second, the right-most data node of ALEX detects append-only insertion behavior by maintaining the value of the maximum key in the node and keeping a counter for how many times an insert exceeds that maximum value. If most inserts exceed the maximum value, that implies append-only behavior, so the data node expands to the right without doing model-based re-insertion; the expanded space is kept initially empty in anticipation of more append-like inserts.

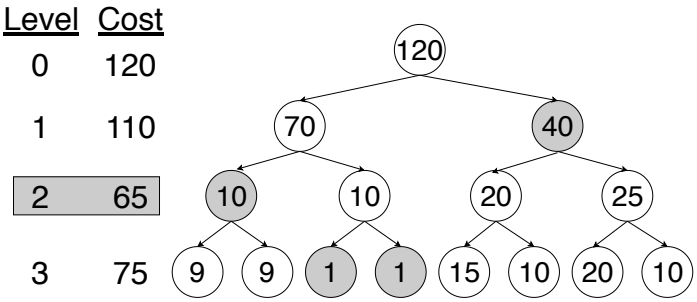

Figure 2-7: Fanout Tree

2.3.6 Bulk Load

ALEX supports a bulk load operation, which is used in practice to index large amounts of data at initialization or for rebuilding an index. Our goal is to find an RMI structure with minimum cost, defined as the expected average time to do an operation (i.e., lookup or insert) on this RMI. Any ALEX operation is composed of TraverseToLeaf to the data node followed by an intra-node operation, so RMI cost is modeled by combining the TraverseToLeaf and intra-node cost models.

Bulk Load Algorithm

Using the cost models, we grow an RMI downwards greedily, starting from the root node. At each node, we independently make a decision about whether the node should be a data node or an internal node, and in the latter case, what the fanout should be. The fanout must be a power of 2, and child nodes will equally divide the key space of the current node. Note that we can make this decision locally for each node because we use linear cost models, so decisions will have a purely additive effect on the overall cost of the RMI. If we decide the node should be an internal node, we recurse on each of its child nodes. This continues until all the data is loaded in ALEX.

The Fanout Tree

As we grow the RMI, the main challenge is to determine the best fanout at each node. We introduce the concept of a *fanout tree* (FT), which is a complete binary tree. An FT will help decide the fanout for a single RMI node; in our bulk loading algorithm, we construct an FT each time we want to decide the best fanout for an RMI node. A fanout of 1 means that the RMI node should be a data node.

Fig. 2-7 shows an example FT. Each FT node represents a possible child of the RMI node. If the key space of the RMI node is $[0, 1)$, then the i-th FT node on a level with n children represents a child RMI node with key space $[i/n, (i+1)/n)$. Each FT node is associated with the expected cost of constructing a data node over its key space, as predicted by the intra-node cost models. Our goal is to find a set of FT nodes that cover the entire key space of the RMI node with minimum overall cost. The overall cost of a covering set is the sum of the costs of its FT nodes, as well as the TraverseToLeaf cost due to model size (e.g., going a level deeper in the FT means

the RMI node must have twice as many pointers). This covering set determines the optimal fanout of the RMI node (i.e., the number of child pointers) as well as the optimal way to allocate child pointers.

We use the following method to find a low-cost covering set: (1) Starting from the FT root, grow entire levels of the FT at a time, and compute the cost of using each level as the covering set. Continue doing so until the costs of each successive level start to increase. In Fig. 2-7, we find that level 2 has the lowest combined cost, and we do not keep growing after level 3. In concept, a deeper level might have lower cost, but computing the cost for each FT node is expensive. (2) Starting from the level of the FT with lowest combined cost, we start merging or splitting FT nodes locally. If the cost of two adjacent FT nodes is higher than the cost of its parent, then we merge (e.g., the nodes with cost 20 and 25 are merged to one with cost 40); this might happen when the two nodes have very few keys, or when their distributions are similar. In the other direction, if the cost of a FT node is higher than the cost of its two children, we split the FT node (e.g., the node with cost 10 is split into two nodes each with cost 1); this might happen when the two halves of the key space have different distributions. We continue with this process of merging and splitting adjacent nodes locally until it is no longer possible. We return the resulting covering set of FT nodes.

2.4 Analysis of ALEX

In this section, we provide bounds on the RMI depth and complexity analysis. Bounds on the performance of model-based search are found in Appendix A.8.

2.4.1 Bound on RMI depth

In this section we present a worst-case bound on maximum RMI depth and describe how to achieve it. Note that the goal of ALEX is to maximize performance, not to minimize tree depth; though the two are correlated, the latter is simply a proxy for the former (e.g., depth is one input to our cost models). Therefore, this analysis is useful for gaining intuition about RMI depth, but does not reflect worst-case guarantees in practice.

Let m be the maximum node size, defined in number of slots (in the pointers array for internal nodes, in the Gapped Array for data nodes). We constrain node size to be a power of 2: $m = 2^k$. Internal nodes can have up to m child pointers, and data nodes must contain no more than md_u keys. Let all keys to be indexed fall within the key space s. Let p be the minimum number of partitions such that when the key space s is divided into p partitions of equal width, every partition contains no more than md_u keys. Define the root node depth as 0.

Theorem 1. *We can construct an RMI that satisfies the max node size and upper density limit constraints whose depth is no larger than* $\lceil \log_m p \rceil$*—we call this the maximal depth. Furthermore, we can maintain maximal depth under inserts. (Note that* p *might change under inserts.)*

In other words, the depth of the RMI is bounded by the density of the densest subregion of s. In contrast, B+Trees bound depth as a function of the number of keys. Theorem 1 can also be applied to a subspace within s, which would correspond to some subtree within the RMI.

Proof. Constructing an RMI with maximal depth is straightforward. The densest subregion, which spans a key space of size $|s|/p$, is allocated to a data node. The traversal path from the root to this densest region is composed of internal nodes, each with m child pointers. It takes $\lceil \log_m p \rceil$ internal nodes to narrow the key space size from $|s|$ to $|s|/p$. To minimize depth in other subtrees of the RMI, we apply this construction mechanism recursively to the remaining parts of the space s.

Starting from an RMI that satisfies maximal depth, we maintain maximal depth using the mechanisms in Section 2.3.3 under the following policy: (1) Data nodes expand until they reach max node size. (2) When a data node must split due to max node size, it splits sideways to maintain current depth (potentially propagating the split up to some ancestor internal node). (3) When splitting sideways is no longer possible (all ancestor nodes are at max node size), split downwards. By following this policy, RMI only splits downward when p grows by a factor of m, thereby maintaining maximal depth. □

2.4.2 Complexity analysis

Here we provide complexity of lookups and inserts, as well as the mechanisms from Section 2.3.3. Both lookups and inserts do TraverseToLeaf in $\lceil \log_m p \rceil$ time. Within the data node, exponential search for lookups is bounded in the worst case by $O(\log m)$. In the best case, the data node model predicts the key's position perfectly, and lookup takes $O(1)$ time. We show in the next sub-section that we can reduce exponential search time according to a space-time trade-off.

Inserts into a non-full node are composed of a lookup, potentially followed by shifts to introduce a gap for the new key. This is bounded in the worst case by $O(m)$, but since Gapped Array achieves $O(\log m)$ shifts per insert with high probability [15], we expect $O(\log m)$ complexity in most cases. In the best case, the predicted insertion position is correct and is a gap, and we place the key exactly where the model predicts for insert complexity of $O(1)$; furthermore, a later model-based lookup will result in a direct hit in $O(1)$.

There are three important mechanisms in Section 2.3.3, whose costs are defined by how many elements must be copied: (1) Expansion of a data node, whose cost is bounded by $O(m)$. (2) Splitting downwards into two nodes, whose cost is bounded by $O(m)$. (3) Splitting sideways into two nodes and propagating upwards in the path to some ancestor node, whose cost is bounded by $O(m\lceil \log_m p \rceil)$ because every internal node on this path must also split. As a result, the worst-case performance for insert into a full node is $O(m\lceil \log_m p \rceil)$.

2.5 Evaluation

We compare ALEX with the Learned Index, B+Tree, a model-enhanced B+Tree, and Adaptive Radix Tree (ART), using a variety of datasets and workloads. This evaluation demonstrates that:

- On read-only workloads, ALEX achieves up to 4.1×, 2.2×, 2.9×, 3.0× higher throughput and 800×, 15×, 160×, 8000× smaller index size than the B+Tree, Learned Index, Model B+Tree, and ART, respectively.
- On read-write workloads, ALEX achieves up to 4.0×, 2.7×, 2.7× higher throughput and 2000×, 475×, 36000× smaller index size than the B+Tree, Model B+Tree, and ART, respectively.
- ALEX has competitive bulk load times and maintains an advantage over other indexes when scaling to larger datasets and under distribution shift due to data skew.
- Gapped Array and the adaptive RMI structure allow ALEX to adapt to different datasets and workloads.

2.5.1 Experimental Setup

We implement ALEX in C++[1]. We perform our evaluation via single-threaded experiments on an Ubuntu Linux machine with Intel Core i9-9900K 3.6GHz CPU and 64GB RAM. We compare ALEX against four baselines. (1) A standard B+Tree, as implemented in the STX B+Tree [20]. (2) Our best-effort reimplementation of the Learned Index [97], using a two-level RMI with linear models at each node and binary search for lookups.[2] (3) Model B+Tree, which maintains a linear model in every node of the B+Tree, stores each node as a Gapped Array, and uses model-based exponential search instead of binary search, implemented on top of [20]; this shows the benefit of using models while keeping the fundamental B+Tree structure. (4) Adaptive Radix Tree (ART) [107], a trie that adapts to the data which is optimized for main memory indexing, implemented in C [34]. Since ALEX supports all operations common in OLTP workloads, we do not compare to hash tables and dynamic hashing techniques, which cannot efficiently support range queries.

For each dataset and workload, we use grid search to tune the page size for B+Tree and Model B+Tree and the number of models for Learned Index to achieve the best throughput. In contrast, no tuning is necessary for ALEX, unless users place additional constraints. For example, users might want to bound the latency of a single operation. We set a max node size of 16MB to achieve tail latency (99.9th percentile) of around

[1] `https://github.com/microsoft/ALEX`

[2] In private communication with the authors of [97], we learned that the added complexity of using a neural net for the root model usually is not justified by the resulting minor performance gains, which we also independently verified.

Table 2.1: Dataset Characteristics

	longitudes	**longlat**	**lognormal**	**YCSB**
Num keys	1B	200M	190M	200M
Key type	double	double	64-bit int	64-bit int
Payload size	8B	8B	8B	80B
Total size	16GB	3.2GB	3.04GB	17.6GB

2μs per operation, but max node size can be adjusted according to user's desired limits (Fig. 2-15).

Index size of ALEX and Learned Index is the sum of the sizes of all models used in the index and metadata; index size for ALEX also includes internal node pointers. For ALEX, each linear model consists of two 64-bit doubles which represent the slope and intercept. Learned Index keeps two additional integers per model that represent the error bounds. The index size of B+Tree and Model B+Tree is the sum of the sizes of all inner nodes, which for Model B+Tree includes the models in each node. The index size of ART is the sum of inner node sizes minus the total size of keys, since keys are encoded into the inner nodes. The data size of ALEX is the sum of the sizes of the arrays containing the keys and payloads, including gaps, as well as the bitmap in each data node. The data size of B+Tree is the sum of the sizes of all leaf nodes. At initialization, the Gapped Arrays in data nodes are set to have 70% space utilization, comparable to B+Tree leaf node space utilization [67].

Datasets

We run all experiments using 8-byte keys from some dataset and randomly generated fixed-size payloads. We evaluate ALEX on 4 datasets, whose characteristics and CDFs are shown in Table 2.1 and Fig. 2-8. The *longitudes* dataset consists of the longitudes of locations around the world from Open Street Maps [13]. The *longlat* dataset consists of compound keys that combine longitudes and latitudes from Open Street Maps by applying the transformation $k = 180 \cdot \text{floor}(\text{longitude}) + \text{latitude}$ to every pair of longitude and latitude. The resulting distribution of keys k is highly non-linear. The *lognormal* dataset has values generated according to a lognormal distribution with $\mu = 0$ and $\sigma = 2$, multiplied by 10^9 and rounded down to the nearest integer. The *YCSB* dataset has values representing user IDs generated according to the YCSB Benchmark [30], which are uniformly distributed across the full 64-bit domain, and uses an 80-byte payload. These datasets do not contain duplicate values. Unless otherwise stated, these datasets are randomly shuffled to simulate a uniform dataset distribution over time.

Workloads

Our primary metric for evaluating ALEX is average throughput. We evaluate throughput for five workloads: (1) a read-only workload, (2) a read-heavy workload with 95% reads and 5% inserts, (3) a write-heavy workload with 50% reads and 50% inserts, (4)

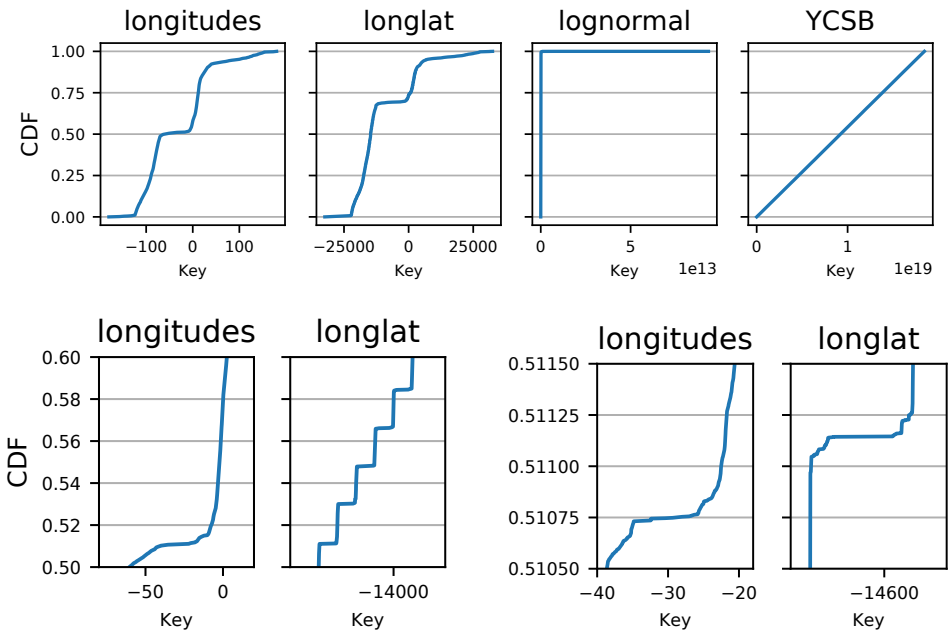

Figure 2-8: Dataset CDFs, and zoomed-in CDFs.

Table 2.2: ALEX Statistics after Bulk Load

	longitudes	**longlat**	**lognormal**	**YCSB**
Avg depth	1.01	1.56	1.80	1
Max depth	2	4	3	1
Num inner nodes	55	1718	24	1
Num data nodes	4450	23257	757	1024
Min DN size	672B	16B	224B	12.3MB
Median DN size	161KB	39.6KB	2.99MB	12.3MB
Max DN size	5.78MB	8.22MB	14.1MB	12.3MB

a short range query workload with 95% reads and 5% inserts, and (5) a write-only workload, to complete the read-write spectrum. For the first three workloads, reads consist of a lookup of a single key. For the short range workload, a read consists of a key lookup followed by a scan of the subsequent keys. The number of keys to scan is selected randomly from a uniform distribution with a maximum scan length of 100. For all workloads, keys to look up are selected randomly from the set of existing keys in the index according to a Zipfian distribution. The first four workloads roughly correspond to Workloads C, B, A, and E from the YCSB benchmark [30], respectively. For a given dataset, we initialize an index with 100 million keys. We then run the workload for 60 seconds, inserting the remaining keys. We report the throughput of operations completed in that time, where operations are either inserts or reads. For the read-write workloads, we interleave the operations: for the read-heavy workload and short range workload, we perform 19 reads/scans, then 1 insert, then repeat the cycle; for the write-heavy workload, we perform 1 read, then 1 insert, then repeat the cycle.

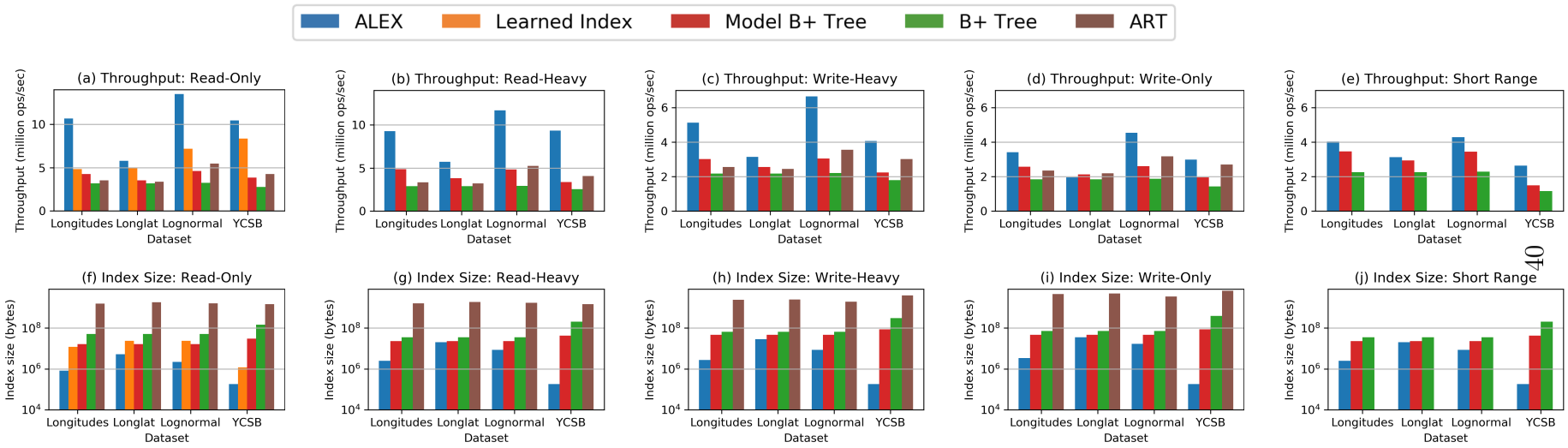

Figure 2-9: ALEX vs. Baselines: Throughput & Index Size. Throughput includes model retraining time.

2.5.2 Overall Results

Read-only Workloads

For read-only workloads, Figs. 2-9a and 2-9f show that ALEX achieves up to 4.1×, 2.2×, 2.9×, 3.0× higher throughput and 800×, 15×, 160×, 8000× smaller index size than the B+Tree, Learned Index, Model B+Tree, and ART, respectively.

On the longlat and YCSB datasets, ALEX performance is similar to Learned Index. The longlat dataset is highly non-uniform, so ALEX is unable to achieve high performance, even with adaptive RMI. The YCSB dataset is nearly uniform, so the optimal allocation of models is uniform; ALEX adaptively finds this optimal allocation, and Learned Index allocates this way by nature, so the resulting RMI structures are similar. On the other two datasets, ALEX has more performance advantage over Learned Index, which we explain in Section 2.5.3.

In general, Model B+Tree outperforms B+Tree while also having smaller index size, because the tuned page size of Model B+Tree is always larger than those of B+Tree. The benefit of models in Model B+Tree is greatest when the key distribution within each node is more uniform, which is why Model B+Tree has least benefit on non-uniform datasets like longlat.

The index size of ALEX is dependent on how well ALEX can model the data distribution. On the YCSB dataset, ALEX does not require a large RMI to accurately model the distribution, so ALEX achieves small index size. However, on datasets that are more challenging to model such as longlat, ALEX has a larger RMI with more nodes. ALEX has smaller index size than the Learned Index, even when throughput is similar, for two reasons. First, ALEX uses model-based inserts to obtain better predictive accuracy for each model, which we show in Section 2.5.3, and therefore achieves high throughput while using relatively fewer models. Second, ALEX adaptively allocates data nodes to different parts of the key space and does not use any more models than necessary (Fig. 2-3), whereas Learned Index fixes the number of models and ends up with many redundant models. The index size of ART is higher than all other indexes. [107] claims that ART uses between 8 and 52 bytes to store each key, which is in agreement with the observed index sizes.

Table 2.2 shows ALEX statistics after bulk loading, including data node (DN) sizes. The root has depth 0. Average depth is averaged over keys. The max depth of the tuned B+Tree is 4 on the YCSB dataset and 5 on the other datasets. Datasets that are easier to model result in fewer nodes. For uniform datasets like YCSB, the data node sizes are also uniform.

Read-Write Workloads

For read-write workloads, Figs. 2-9b to 2-9d and 2-9g to 2-9i show that ALEX achieves up to 4.0×, 2.7×, 2.7× higher throughput and 2000×, 475×, 36000× smaller index size than the B+Tree, Model B+Tree, and ART, respectively. The Learned Index has insert time orders of magnitude slower than ALEX and B+Tree, so we do not include it in these benchmarks.

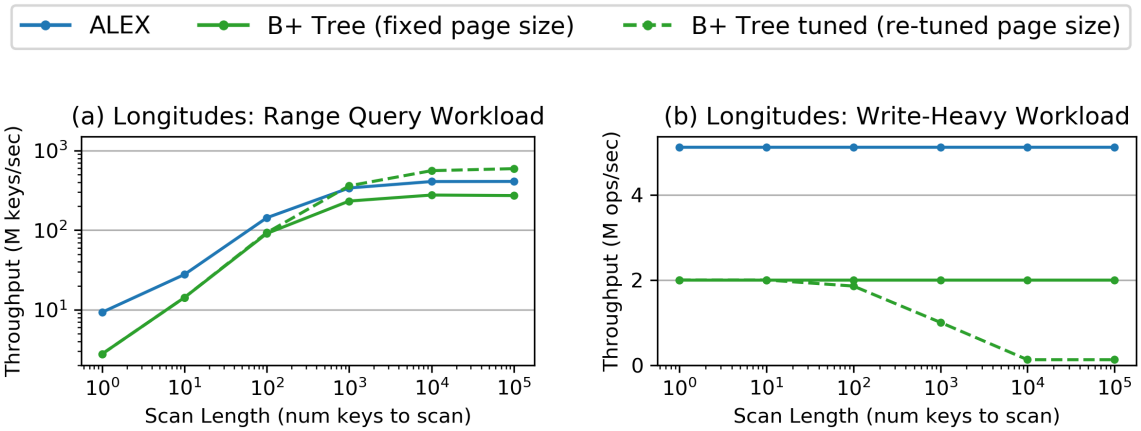

Figure 2-10: (a) When scan length exceeds 1000 keys, ALEX is slower on range queries than a B+Tree whose page size is re-tuned for different scan lengths. (b) However, throughput of the re-tuned B+Tree suffers for other operations, such as point lookups and inserts in the write-heavy workload.

The relative performance advantage of ALEX over baselines decreases as the workload skews more towards writes, because all indexes must pay the cost of copying when splitting/expanding nodes. Copying has an especially big impact for YCSB, for which payloads are 80 bytes. ART achieves comparable throughput to ALEX on the write-only workload for YCSB because ART does not keep payloads clustered, so it avoids the high cost of copying 80-byte payloads. Note that ALEX could similarly avoid copying large payloads by storing unclustered payloads separately and keeping a pointer with every key; however, this would impact scan performance. On datasets that are challenging to model such as longlat, ALEX only achieves comparable write-only throughput to Model B+Tree and ART, but is still faster than B+Tree.

Range Query Workloads

Figs. 2-9e and 2-9j show that ALEX maintains its advantage over B+Tree on the short range workload, achieving up to 2.27×, 1.77× higher throughput and 1000×, 230× smaller index size than B+Tree and Model B+Tree, respectively. However, the relative throughput benefit decreases, compared to Fig. 2-9b. This is because as scan time begins to dominate overall query time, the speedups that ALEX achieves on lookups become less apparent. The ART implementation from [34] does not support range queries; we suspect range queries on ART would be slower than for the other indexes because ART does not cluster payloads, leading to poor scan locality. Appendix A.5.1 shows that ALEX continues to outperform other indexes on a workload that mixes inserts, point lookups, and short range queries.

To show how performance varies with range query selectivity, we compare ALEX against two B+Tree configurations with increasingly larger range scan length over the longitudes dataset (Fig. 2-10a). In the first B+Tree configuration, we use the optimal B+Tree page size on the write-heavy workload (Fig. 2-9c), which is 1KB (solid green line). In the second B+Tree configuration, we tune the B+Tree page size for each

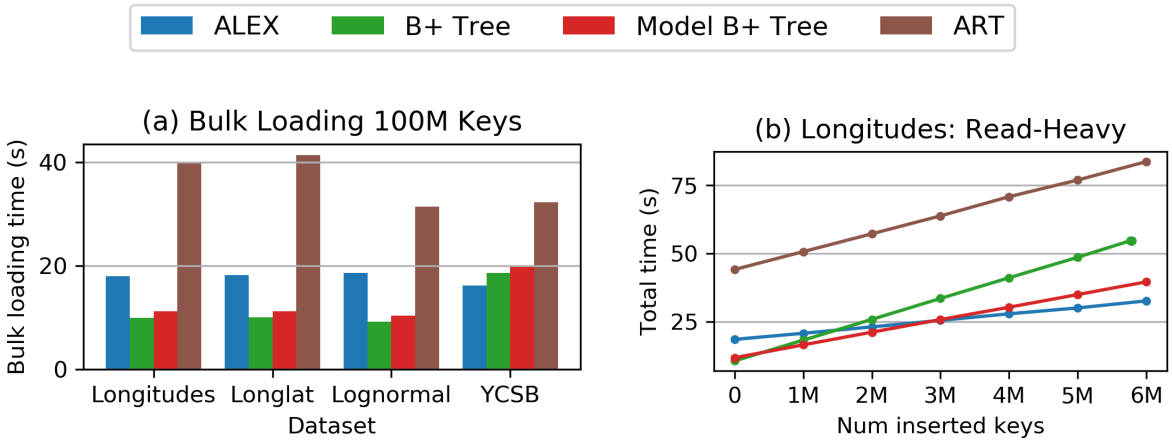

Figure 2-11: ALEX takes 50% more than time than B+Tree to bulk load on average, but quickly makes up for this by having higher throughput.

different scan length (dashed green line).

Unsurprisingly, Fig. 2-10a shows as scan length increases, the throughput in terms of keys scanned per second increases for all indexes due to better locality and a smaller fraction of time spent on the initial point lookup. Furthermore, ALEX outperforms the 1KB-page B+Tree for all scan lengths due to ALEX's larger nodes; median ALEX data node size is 161KB on the longitudes dataset (Table 2.2), which benefits scan locality—scanning larger contiguous chunks of memory leads to better prefetching and fewer pointer chases. This makes up for the Gapped Array's overhead.

However, if we re-tune the B+Tree page size for each scan length (dashed green line), the B+Tree outperforms ALEX when scan length exceeds 1000 keys because past this point, the overhead of Gapped Array outpaces ALEX's scan locality advantage from having larger node sizes. However, this comes at the cost of performance on other operations: Fig. 2-10b shows that if we run the re-tuned B+Tree on the write-heavy workload, which includes both point lookups and inserts, its performance would begin to decline when scan length exceeds 100 keys. In particular, larger B+Tree pages lead to a higher number of search iterations for lookups and shifts for inserts; ALEX avoids both of these problems for large data nodes by using Gapped Arrays with model-based inserts. We show in Appendix A.5 that this behavior also occurs on the other three datasets.

Bulk Loading

We compare the time to initialize each index with bulk loading, which includes the time to sort keys. Fig. 2-11a shows that on average, ALEX only takes 50% more time to bulk load than B+Tree, and in the worst case is only 2× slower than B+Tree. On the YCSB dataset, B+Tree and Model B+Tree take longer to bulk load due to the larger payload size, but bulk loading ALEX remains efficient due to its simple structure (Table 2.2). Model B+Tree is slightly slower to bulk load than B+Tree due to the overhead of training models for each node. ART is slower to bulk load than B+Tree, Model B+Tree, and ALEX.

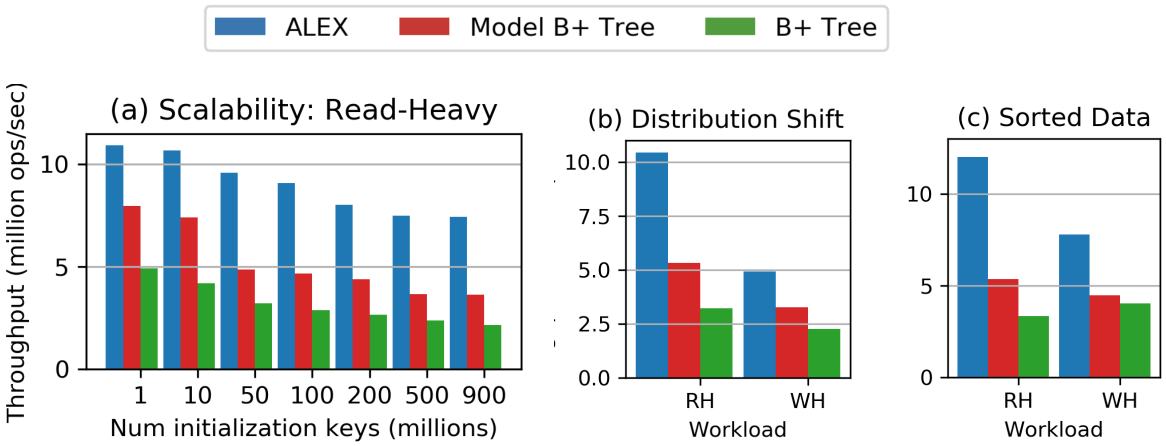

Figure 2-12: ALEX maintains high throughput when scaling to large datasets and under data distribution shifts. (RH = Read-Heavy, WH = Write-Heavy)

ALEX can quickly make up for its slower bulk loading time than B+Tree by having higher throughput performance. Fig. 2-11b shows that when running the read-heavy workload on the longitudes dataset, ALEX's total time usage (bulk loading plus workload) drops below all other indexes after only 3 million inserts. We provide a more detailed bulk loading evaluation in Appendix A.1.

Scalability

ALEX performance scales well to larger datasets. We again run the read-heavy workload on the longitudes dataset, but instead of initializing the index with 100 million keys, we vary the number of initialization keys. Fig. 2-12a shows that as the number of indexed keys increases, ALEX maintains higher throughput than B+Tree and Model B+Tree. In fact, as dataset size increases, ALEX throughput decreases at a surprisingly slow rate. This occurs because ALEX adapts its RMI structure in response to the incoming data.

Dataset Distribution Shift

ALEX is robust to dataset distribution shift. We initialize the index with the 50 million smallest keys and run read-write workloads by inserting the remaining keys in random order. This simulates distribution shift because the keys we initialize with come from a completely disjoint domain than the keys we subsequently insert with. Fig. 2-12b shows that ALEX maintains up to 3.2× higher throughput than B+Tree in this scenario. ALEX is also robust to adversarial patterns such as sequential inserts in sorted order, in which new keys are always larger than the maximum key currently indexed. Fig. 2-12c shows that when we initialize with the 50 million smallest keys and insert the remaining keys in ascending sorted order, ALEX has up to 3.6× higher throughput than B+Tree. Appendix A.4 further shows that ALEX is robust to radically changing key distributions.

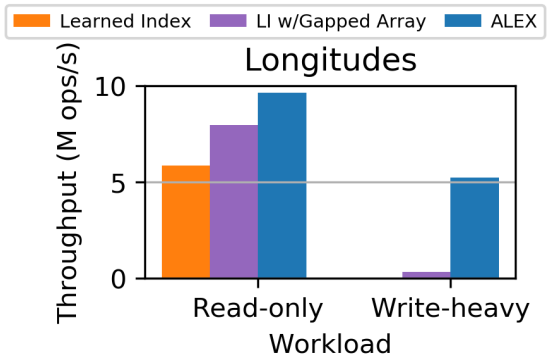

Figure 2-13: Impact of Gapped Array and adaptive RMI.

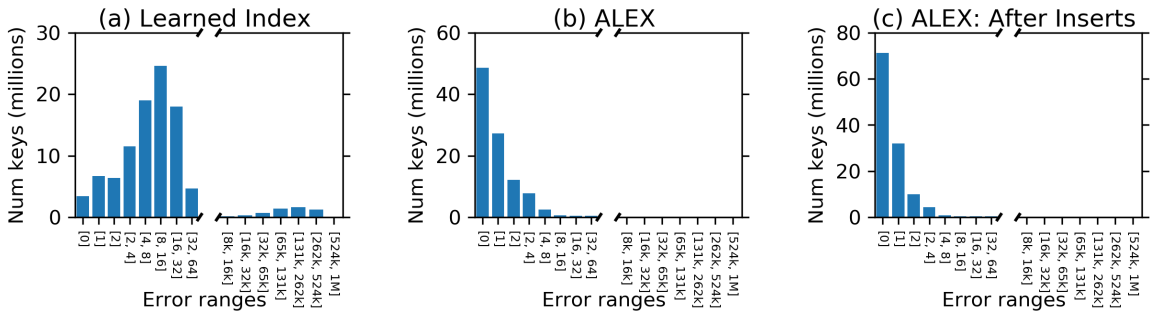

Figure 2-14: ALEX achieves smaller prediction error than the Learned Index.

2.5.3 Drilldown into ALEX Design Trade-offs

In this section, we delve deeper into how node layout and adaptive RMI help ALEX achieve its design goals.

Part of ALEX's advantage over Learned Index comes from using model-based insertion with Gapped Arrays in the data nodes, but most of ALEX's advantage for dynamic workloads comes from the adaptive RMI. To demonstrate the effects of each contribution, Fig. 2-13 shows that taking a 2-layer Learned Index and replacing the single dense array of values with a Gapped Array per leaf (LI w/Gapped Array) already achieves significant speedup over Learned Index for the read-only workload. However, a Learned Index with Gapped Arrays achieves poor performance on read-write workloads due to the presence of fully-packed regions which require shifting many keys for each insert. ALEX's ability to adapt the RMI structure to the data is necessary for good insert performance.

During lookups, the majority of the time is spent doing local search around the predicted position. Smaller prediction errors directly contribute to decreased lookup time. To analyze the prediction errors of the Learned Index and ALEX, we initialize an index with 100 million keys from the longitudes dataset, use the index to predict the position of each of the 100 million keys, and track the distance between the predicted position and the actual position. Fig. 2-14a shows that the Learned Index has prediction error with mode around 8-32 positions, with a long tail to the right.

Table 2.3: Data Node Actions When Full (Write-Heavy)

	longitudes	**longlat**	**lognormal**	**YCSB**
Expand + scale	26157	113801	2383	1022
Expand + retrain	219	2520	2	1026
Split sideways	79	2153	7	0
Split downwards	0	230	0	0
Total times full	26455	118704	2392	2048

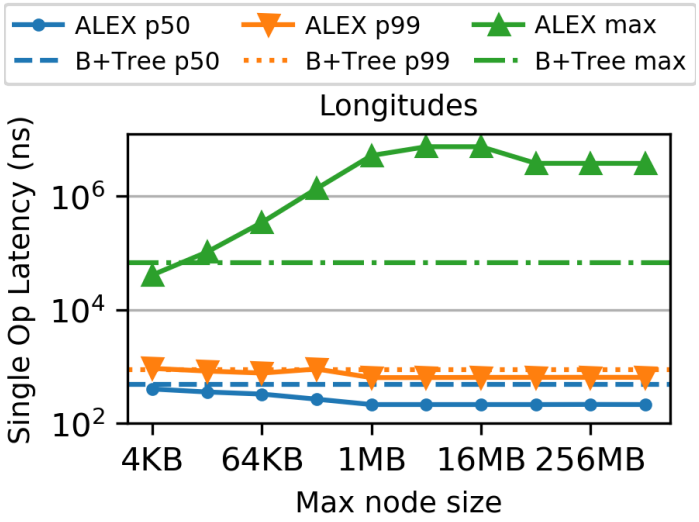

Figure 2-15: Latency of a single operation.

On the other hand, ALEX achieves much lower prediction error by using model-based inserts. Fig. 2-14b shows that after initializing, ALEX often has no prediction error, the errors that do occur are often small, and the long tail of errors has disappeared. Fig. 2-14c shows that even after 20 million inserts, ALEX maintains low prediction errors.

Once a data node becomes full, one of four actions happens: if there is no cost deviation, then (1) the node is expanded and the model is scaled. Otherwise, the node is either (2) expanded and its model retrained, (3) split sideways, or (4) split downwards. Table 2.3 shows that in the vast majority of cases, the data node is simply expanded and the model scaled, which implies that models usually remain accurate even after inserts, assuming no radical distribution shift. The number of occurrences of a data node becoming full is correlated with the number of data nodes (Table 2.2). On YCSB, expansion with model retraining is more common because the data nodes are large, so cost deviation often results simply from randomness.

Users can adjust the max node size to achieve target tail latencies, if desired. In Fig. 2-15, we run the write-heavy workload on the longitudes dataset, measuring the latency for every operation. As we increase the max node size, median and even p99 latency of ALEX decreases, because ALEX has more flexibility to build a better-performing RMI (e.g., ability to have higher internal node fanout). However,

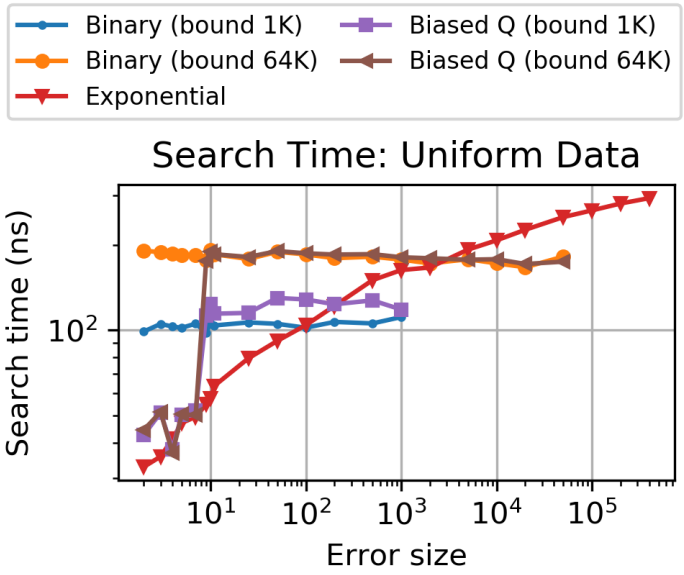

Figure 2-16: Exponential vs. other search methods.

maximum latency increases, because an insert that triggers an expansion or split of a large node is slow. If the user has strict latency requirements, they can decrease the max node size accordingly. After increasing the max node size beyond 64MB, latencies do not change because ALEX never decides to use a node larger than 64MB.

Search Method Comparison

In order to show the trade-off between exponential search and other search methods, we perform a microbenchmark on synthetic data. We create a dataset with 100 million perfectly uniformly distributed doubles. We then perform searches for 10 million randomly selected values from this dataset. We use three search methods: binary search and biased quaternary search (proposed in [97] to take advantage of accurate predictions), each evaluated with two different error bound sizes, as well as exponential search. For each lookup, the search method is given a predicted position that has some synthetic amount of error in the distance to the actual position value. Fig. 2-16 shows that the search time of exponential search increases proportionally with the logarithm of error size, whereas the binary search methods take a constant amount of time, regardless of error size. This is because binary search must always begin search within its error bounds, and cannot take advantage of cases when the error is small. Therefore, exponential search should outperform binary search if the prediction error of the RMI models in ALEX is small. As we showed in Section 2.5.3, ALEX maintains low prediction errors through model-based inserts. Therefore, ALEX is well suited to take advantage of exponential search. Biased quaternary search is competitive with exponential search when error is below σ (we set $\sigma = 8$ for this experiment; see [97] for details) because search can be confined to a small range, but performs similarly to binary search when error exceeds σ because the full error bound must be searched. We

prefer exponential search to biased quaternary search due to its smoother performance degradation and simplicity of implementation (e.g., no need to tune σ).

2.6 Related Work

Learned Index Structures: The most relevant work is the Learned Index [97], discussed in Section 2.1.2. Learned Index has similarities to prior work that explored how to compute down a tree index. Tries [93] use key prefixes instead of B+Tree splitters. Masstree [123] and Adaptive Radix Tree [107] combine the ideas of B+Tree and trie to reduce cache misses. Plop-hashing [101] uses piecewise linear order-preserving hashing to distribute keys more evenly over pages. Digital B-tree [111] uses bits of a key to compute down the tree more flexibly. [112] proposes to partially expand the space instead of always doubling when splitting in B+Tree. [65] proposes the idea of interpolation search within B+Tree nodes; this idea was revisited in [140]. The interpolation-based search algorithms in [186] can complement ALEX's search strategy. Hermit [192] creates a succinct tree structure for secondary indexes.

Other works propose replacing the leaf nodes of a B+Tree with other data structures in order to compress the index, while maintaining search and update performance. FITing-tree [60] uses linear models in its leaf nodes, while BF-tree [10] uses bloom filters in its leaf nodes.

All these works share the idea that using extra computation or data structures can make search faster by reducing the number of binary search hops and corresponding cache misses, while allowing larger node sizes and hence a smaller index size. However, ALEX is different in several ways: (1) We use a model to split the key space, similar to a trie, but no search is required until we reach the leaf level. (2) ALEX's accurate linear models enable larger node sizes without sacrificing search and update performance. (3) Model-based insertion reduces the impact of model's misprediction. (4) ALEX's cost models automatically adjust the index structure to dynamic workloads.

Memory Optimized Indexes: There is a large body of work on optimizing tree index structures for main memory by exploiting hardware features such as CPU cache, multi-core, SIMD, and prefetching. CSS-trees [161] improve B+Tree's cache behavior by matching index node size to CPU cache-line size and eliminating pointers in index nodes by using arithmetic operations to find child nodes. CSB$^+$-tree [162] extends the static CSS-trees by supporting incremental updates without sacrificing CPU cache performance. [71] evaluates the effect of node size on the performance of CSB$^+$-tree analytically and empirically. pB+-tree [28] uses larger index nodes and relies on prefetching instructions to bring index nodes into cache before nodes are accessed. In addition to optimizing for cache performance, FAST [87] further optimizes searches within index nodes by exploiting SIMD parallelism.

ML in other DB components: Machine learning has been used to improve cardinality estimation [90, 51], query optimization [125], workload forecasting [117], multi-dimensional indexing [138], and data partitioning [194]. SageDB [95] envisions a database system in which every component is replaced by a learned component. These studies show that the use of machine learning enables workload-specific optimizations,

which also inspired our work.

2.7 Discussion & Future Work

In this section, we discuss our design and possible future extensions.

Role of Machine Learning. Similar to the original learned indexes, machine learning (ML) currently plays a limited, but important role in the design of ALEX. ALEX uses simple linear regression models, at all levels of the RMI. We found linear regression models to strike the right balance between computation overhead vs. prediction accuracy. We have also found these to work better than the even simpler, pure interpolation search [140]. Polynomial models, piecewise linear splines, or even a hybrid of ML models and B+Tree in the RMI structure, as originally proposed by [97] might be worth exploring, but they have not been the focus of this chapter.

Concurrency Control. To use ALEX in a database system requires concurrency control for handling updates with concurrent lookups. For lookups, without Adaptive RMI, only a shared lock on the leaf data node is needed in ALEX. With adaptive RMI, to protect against concurrent modifications of the RMI structure, lookups can use lock-coupling (or crabbing) [66] while traversing the RMI to a leaf data node. Similarly, for inserts, without adaptive RMI, an exclusive lock on the leaf data node is sufficient. For cases which require an expansion, we need to hold an exclusive lock on the leaf data node and the corresponding leaf level model in the RMI, since the model needs to be retrained. With adaptive RMI with node splitting on inserts, the structure of the RMI can be modified as well. Since this is very similar to splits in a B+Tree, we believe lock-coupling [66] can be applied in ALEX, in this case as well.

Data Skew. Data skew is quite common in real-life workloads and hence it is important for any index structure to be able to deal with it. Fig. 2-12b shows that ALEX is able to handle some data skew gracefully. However, it is also easy to construct an adversarial workload where ALEX's performance degrades significantly, as shown in Fig. 2-12c. Future work could explore even better node layouts for ALEX, for example the adaptive PMA [16] could, in theory, prevent the adversarial case shown in Fig. 2-12c. Better models, or different adaptability strategies for the RMI are other possible directions to pursue.

Secondary Indexes. Handling secondary indexes is straight-forward in ALEX. Similar to a B+Tree, instead of storing actual data at the leaf level, ALEX can store a pointer to the data. The difficulty is in dealing with duplicate keys, which ALEX currently does not support.

Secondary Storage. Handling secondary storage, for data that does not fit in-memory is another important practical requirement. ALEX uses a node per leaf layout, which could be mapped to disk pages, and hence is secondary storage friendly. A simple extension of ALEX could store a pointer to a leaf data page in secondary storage, for every leaf node. However, as observed in [97], supporting secondary storage may require: changes to model training, introducing an additional translation table, or using more complex models.

2.8 Conclusion

We build on the excitement of learned indexes by proposing ALEX, a new updatable learned index that effectively combines the core insights from the Learned Index with proven storage and indexing techniques. Specifically, we propose a Gapped Array node layout that uses model-based inserts and exponential search, combined with an adaptive RMI structure driven by simple cost models, to achieve high performance and low memory footprint on dynamic workloads. Our in-depth experimental results show that ALEX not only consistently beats B+Tree across the read-write workload spectrum, it even beats the existing Learned Index, on all datasets, by up to 2.2× with read-only workloads.

We believe this chapter presents important learnings to our community and opens avenues for future research in this area. We intend to pursue open theoretical problems about ALEX performance, supporting secondary storage for larger than memory datasets, and new concurrency control techniques tailored to the ALEX design.

Algorithm 1 *Gapped Array* Insertion

```
1: struct Node { keys[] (Gapped Array); num_keys; d_u, d_l;      model: key→ [0, keys.size);
   }
2: procedure INSERT(key)
3:     if num_keys / keys.size >= d_u then
4:         if expected cost ≈ empirical cost then
5:             Expand(retrain=False)
6:         else
7:             Action with lowest cost /* described in Sec. 2.3.3 */
8:         end if
9:     end if
10:    predicted_pos = model.predict(key)
11:    /* check for sorted order */
12:    insert_pos = CorrectInsertPosition(predicted_pos)
13:    if keys[insert_pos] is occupied then
14:        MakeGap(insert_pos) /* described in text */
15:    end if
16:    keys[insert_pos] = key
17:    num_keys++
18: end procedure
19: procedure EXPAND(retrain)
20:    expanded_size = num_keys * 1/d_l
21:    /* allocate a new expanded array */
22:    expanded_keys = array(size=expanded_size)
23:    if retrain == True then
24:        model = /* train linear model on keys */
25:    else
26:        /* scale existing model to expanded array */
27:        model *= expanded_size / keys.size
28:    end if
29:    for key : keys do
30:        ModelBasedInsert(key)
31:    end for
32:    keys = expanded_keys
33: end procedure
34: procedure MODELBASEDINSERT(key)
35:    insert_pos = model.predict(key)
36:    if keys[insert_pos] is occupied then
37:        insert_pos = first gap to right of predicted_pos
38:    end if
39:    keys[insert_pos] = key
40: end procedure
```

Chapter 3

Tsunami: An Instance-Optimized Storage Layout for In-Memory Data

While the previous chapter focused on a instance-optimized learned index for operational workloads that consist of a mix of point lookups, short range queries, inserts, updates, and deletes, the remainder of this thesis will focus on instance-optimized physical designs for analytic workloads. In this chapter and the following chapter, we discuss instance-optimized layouts for improving the performance of scanning and filtering.

Filtering through data is the foundation of any analytical database engine, and several advances over the past several years specifically target database filter performance. For example, column stores [54] delay or entirely avoid accessing columns (i.e., dimensions) which are not relevant to a query, and they often sort the data by a single dimension in order to skip over records that do not match a query filter over that dimension.

If data has to be filtered by more than one dimension, secondary indexes can be used. Unfortunately, their large storage overhead and the latency incurred by chasing pointers make them viable only when the predicate on the indexed dimension has a very high selectivity. An alternative approach is to use (clustered) *multi-dimensional* indexes; these may be tree-based data structures (e.g., k-d trees, R-trees, or octrees) or a specialized sort order over multiple dimensions (e.g., a space-filling curve like Z-ordering or hand-picked hierarchical sort). Many state-of-the-art analytical database systems use multi-dimensional indexes or sort orders to improve the scan performance of queries with predicates over several columns [9, 41, 76].

However, multi-dimensional indexes have significant drawbacks. First, these techniques are hard to tune and require an admin to carefully pick which dimensions to index, if any at all, and the order in which they are indexed. This decision must be revisited every time the data or workload changes, requiring extensive manual labor to maintain performance. Second, there is no single approach (even if tuned correctly) that dominates all others [138].

To address the shortcomings of traditional indexes, recent work has proposed the idea of *learned* multi-dimensional indexes [138, 194, 108, 187, 42]. In particular, Flood [138] is a in-memory multi-dimensional index that automatically optimizes

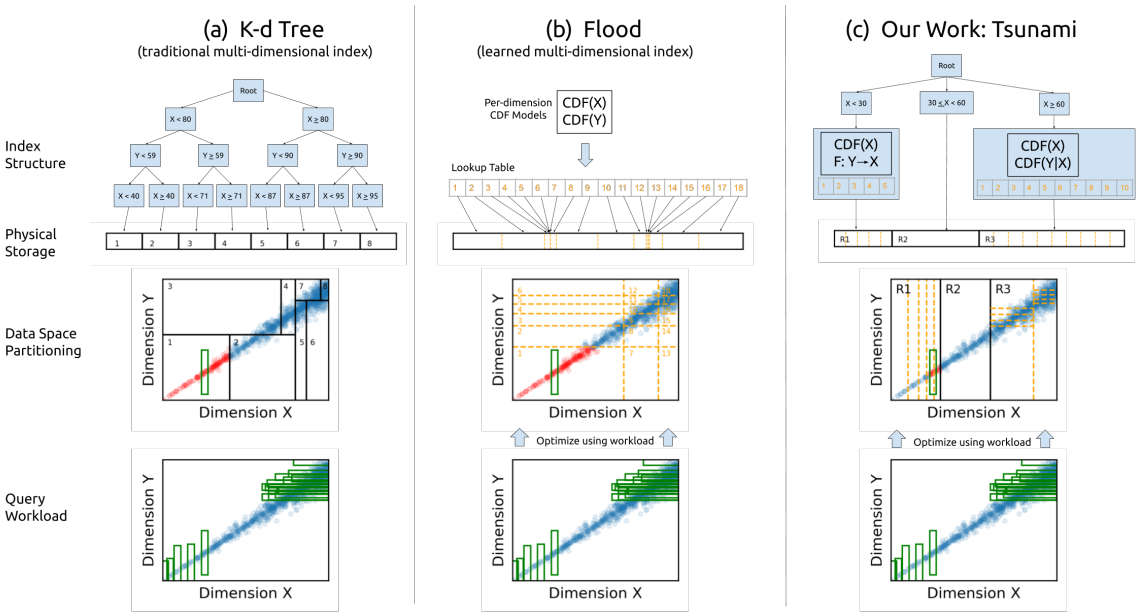

Figure 3-1: Indexes must identify the points that fall in the green query rectangle. To do so, they scan the points in red. (a) K-d tree guarantees equally-sized regions but is not optimized for the workload. (b) Flood is optimized using the workload but its structure is not expressive enough to handle query skew, and cells are unequally sized on correlated data. (c) Tsunami is optimized using the workload, is adaptive to query skew, and maintains equally-sized cells within each region.

its structure to achieve high performance on a particular dataset and workload. In contrast to traditional multi-dimensional indexes, such as the k-d tree, which are created entirely based on the data (see Fig. 3-1a), Flood divides each dimension into some number of partitions based on the observed data and workload (see Fig. 3-1b, explained in detail in Chapter 3.1). The Cartesian product of the partitions in each dimension form a grid. Furthermore, to reduce the index size, Flood uses models of the CDF of each dimension to locate the data.

However, Flood faces a number of limitations in real-world scenarios. First, Flood's grid cannot efficiently adapt to *skewed* query workloads in which query frequencies and filter selectivities vary across the data space. Second, if dimensions are correlated, then Flood cannot maintain uniformly sized grid cells, which degrades performance and memory usage.

To address these limitations, we propose Tsunami, an in-memory read-optimized learned multi-dimensional index that extends the ideas of Flood with new data structures and optimization techniques. First, Tsunami achieves high performance on skewed query workloads by using a lightweight decision tree, called a Grid Tree, to partition space into non-overlapping regions in a way that reduces query skew. Second, Tsunami achieves high performance on correlated datasets by indexing each region using an Augmented Grid, which uses two techniques—*functional mappings* and *conditional CDFs*—to efficiently capture information about correlations.

While recent work explored how correlation can be exploited to reduce the size of

secondary indexes [193, 88], our work goes much further. We demonstrate not only how to leverage correlation to achieve faster and more compact multi-dimensional indexes (in which the data is organized based on the index) rather than secondary indexes, but also how to integrate the optimization for query skew and data correlation into a full end-to-end solution. Tsunami automatically optimizes the data storage organization as well as the multi-dimensional index structure based on the data and workload.

Like Flood [138], Tsunami is a clustered in-memory read-optimized index over an in-memory column store. In-memory stores are increasingly popular due to lower RAM prices [86] and our focus on reads reflects the current trend towards avoiding in-place updates in favor of incremental merges (e.g., RocksDB [164]). We envision that Tsunami could serve as the building block for a multi-dimensional in-memory key-value store or be integrated into commercial in-memory (offline) analytics accelerators like Oracle's Database In-Memory (DBIM) [151].

In summary, we make the following contributions:

1. We design and implement Tsunami, an in-memory read-optimized learned multi-dimensional index that self-optimizes to achieve high performance and robustness to correlated datasets and skewed workloads.
2. We introduce two data structures, the Grid Tree and the Augmented Grid, along with new optimization procedures that enable Tsunami to tailor its index structure and data organization strategy to handle data correlation and query skew.
3. We evaluate Tsunami against Flood, the original in-memory learned multi-dimensional index, as well as a number of traditional non-learned indexes, on a variety of workloads over real datasets. We show that Tsunami is up to 6× and 11× faster than Flood and the fastest optimally-tuned non-learned index, respectively. Tsunami is also adaptable to workload shift, and scales across data size, query selectivity, and dimensionality.

In the remainder of this chapter, we give background (Chapter 3.1), present an overview of Tsunami (Chapter 3.2), introduce its two core components—Grid Tree (Chapter 3.3) and Augmented Grid (Chapter 3.4), present experimental results (Chapter 3.5), review related work (Chapter 3.6), propose future work (Chapter 3.7), and conclude (Chapter 3.8).

3.1 Background

Tsunami is an in-memory clustered multi-dimensional index for a single table. Tsunami aims to increase the throughput performance of analytics queries by decreasing the time needed to filter records based on range predicates. Tsunami supports queries such as:

```
SELECT SUM(R.X)
FROM MyTable
WHERE (a ≤ R.Y ≤ b) AND (c ≤ R.Z ≤ d)
```

where `SUM(R.X)` can be replaced by any aggregation. Records in a d-dimensional table can be represented as points in d-dimensional data space. For the rest of this

chapter, we use the terms *record* and *point* interchangeably. To place Tsunami in context, we first describe the k-d tree as an example of a traditional non-learned multi-dimensional index, and Flood, which originally proposed the idea of learned in-memory multi-dimensional indexing.

3.1.1 K-d Tree: A Traditional Non-Learned Index

The k-d tree [17] is a binary space-partitioning tree that recursively splits d-dimensional space based on the median value along each dimension, until the number of points in each leaf region falls below a threshold, called the page size. Fig. 3-1a shows a k-d tree over 2-dimensional data that has 8 leaf regions. The points within each region are stored contiguously in physical storage (e.g., a column store). By construction, the leaf regions have a roughly equal number of points. To process a query (i.e., identify all points that match the query's filter predicates), the k-d tree traverses the tree to find all leaf regions that intersect the query's filter, then scans all points within those regions to identify points that match the filter predicates.

The k-d tree structure is constructed based on the data distribution but *independently* of the query workload. That is, regardless of whether a region of the space is never queried or whether queries are more selective in some dimensions than others, the k-d tree would still build an index over all data points with the same page size and index overhead. While other traditional multi-dimensional indexes split space in different ways [129, 14, 143, 198], they all share the property that the index is constructed *independent* of the query workload.

3.1.2 Flood: A Learned Index

In contrast, Flood [138] optimizes its layout based on the workload (Fig. 3-1b). We first introduce how Flood works, then explain its two key advantages over traditional indexes, then discuss its limitations.

Given a d-dimensional dataset, Flood first constructs compact models of the CDF of each dimension. The choice of modeling technique is orthogonal; Flood uses a Recursive Model Index [98], but one could also use a histogram or linear regression. Flood uses these models to divide the domain of each dimension into equally-sized *partitions*: let p_i be the number of partitions in each dimension $i \in [0, d)$. Then a point whose value in dimension i is x is placed into the $\lfloor CDF_i(x) \cdot p_i \rfloor$-th partition of dimension i. This guarantees that each partition in a given dimension has an equal number of points. When combined, the partitions of each dimension form a d-dimensional grid with $\prod_{i \in [0,d)} p_i$ *cells*, which are ordered. The points within each cell are stored contiguously in physical storage.

Flood's query processing workflow has three steps, shown in Fig. 3-1b: (1) Using the per-dimension CDF models, identify the range of intersecting partitions in each dimension, and take the Cartesian product to identify the set of intersecting cells. (2) For each intersecting cell, identify the corresponding range in physical storage using a lookup table. (3) Scan all the points within those physical storage ranges, and identify the points that match all query filters.

Flood's Strengths

Flood has two key advantages over traditional indexes such as the k-d tree[1]. First, Flood can automatically tune its grid structure for a given query workload by adjusting the number of partitions in each dimension to maximize query performance. For example, in Fig. 3-1b, there are many queries in the upper-right region of the data space that have high selectivity over dimension Y. Therefore, Flood's optimization technique will place more partitions in dimension Y than dimension X, in order to reduce the number of points those queries need to scan. In other words, Flood learns which dimensions to prioritize over others and adjusts the number of partitions accordingly, whereas non-learned approaches do not take the workload into account and treat all dimensions equally.

Flood's second key advantage is its CDF models. The advantage of indexing using compact CDF models, as opposed to a tree-based structure such as a k-d tree, is lower overhead in both space and time: storing d CDF models takes much less space than storing pointers and boundary keys for all internal tree nodes. It is also much faster to identify intersecting grid cells by invoking d CDF models than by pointer chasing to traverse down a tree index.

The combination of these two key advantages allows Flood to outperform non-learned indexes by up to three orders of magnitude while using up to 50× smaller index size [138].

Flood's Limitations

However, Flood has two key limitations. First, Flood only optimizes for the *average* query, which results in degraded performance when queries are not uniform. For example, in Fig. 3-1b there are a few queries in the lower-left region of the data space that, unlike the many queries in the upper-right region, have high selectivity over dimension X. Since these queries are a small fraction of the total workload, Flood's optimization will not prioritize their performance. As a result, Flood will need to scan a large number of points to create the query result (red points in Fig. 3-1b). Flood's uniform grid structure can only optimize for the average selectivity in each dimension and is not expressive enough to optimize for *both* the upper-right queries and lower-left queries independently. The workload in Fig. 3-1 is an example of a skewed workload. Query skew is common in real workloads: for example, queries often hit recent data more frequently than stale data, and operations monitoring systems only query for health metrics that are exceedingly low or high.

Second, Flood's model-based indexing technique can result in unequally-sized cells when data is correlated. In Fig. 3-1b, even though the CDF models guarantee that the three partitions over dimension X have an equal number of points, as do the six partitions over dimension Y, the 18 grid cells are unequally sized. This degrades performance and space usage (Chapter 3.4.1). Correlations are common in real data: for example, the price and distance of a taxi ride are correlated, as are the dates on which a package is shipped and received.

[1]Flood's minor third advantage, the *sort dimension*, is orthogonal to our work.

The goal of our work, Tsunami, is to maintain the two advantages of Flood—optimization based on the query workload and a compact/fast model-based index structure—while also addressing Flood's limitations in the presence of data correlations and query skew.

3.2 Tsunami Design Overview

Tsunami is a learned multi-dimensional index that is robust to data correlation and query skew. We first introduce the index structure and how it is used to process a query. We then provide an overview of the offline procedures we use to automatically optimize Tsunami's structure.

Tsunami Structure. Tsunami is a composition of two independent data structures: the Grid Tree (Chapter 3.3) and the Augmented Grid (Chapter 3.4). The Grid Tree is a space-partitioning decision tree that divides d-dimensional data space into some number of non-overlapping *regions*. In Fig. 3-1c, the Grid Tree divides data space into three regions by splitting on dimension X.

Within each region, there is an Augmented Grid. Each Augmented Grid indexes the points that fall in its region. In Fig. 3-1c, Regions 1 and 3 each have their own Augmented Grid. Region 2 is not given an Augmented Grid because no queries intersect its region. An Augmented Grid is essentially a generalization of Flood's index structure that uses additional techniques to capture correlations. In Fig. 3-1c, the Augmented Grids use $F : Y \to X$ and $CDF(Y|X)$ instead of Flood's $CDF(Y)$ (explained in Chapter 3.4.2).

Tsunami Query Workflow. Tsunami processes a query in three steps: (1) Traverse the Grid Tree to find all regions that intersect the query's filter. (2) In each region, identify the set of intersecting Augmented Grid cells (Chapter 3.4), then identify the corresponding range in physical storage using a lookup table. (3) Scan all the points within those physical storage ranges, and identify the points that match all query filters.

Tsunami Optimization. Tsunami's offline optimization procedure has two steps: (1) Optimize the Grid Tree using the full dataset and sample query workload (Chapter 3.3.3). (2) In each region of the optimized Grid Tree, construct an Augmented Grid that is optimized over only the points and queries that intersect its region (Chapter 3.4.3).

Intuitively, Tsunami separates the two concerns of query skew and data correlations into its two component structures, Grid Tree and Augmented Grid, respectively. Each structure is optimized in a way that addresses its corresponding concern. We now describe each structure in detail.

3.3 Grid Tree

In this section, we first discuss the performance challenges posed by skewed workloads. We then formally define query skew, and we describe Tsunami's solution for mitigating

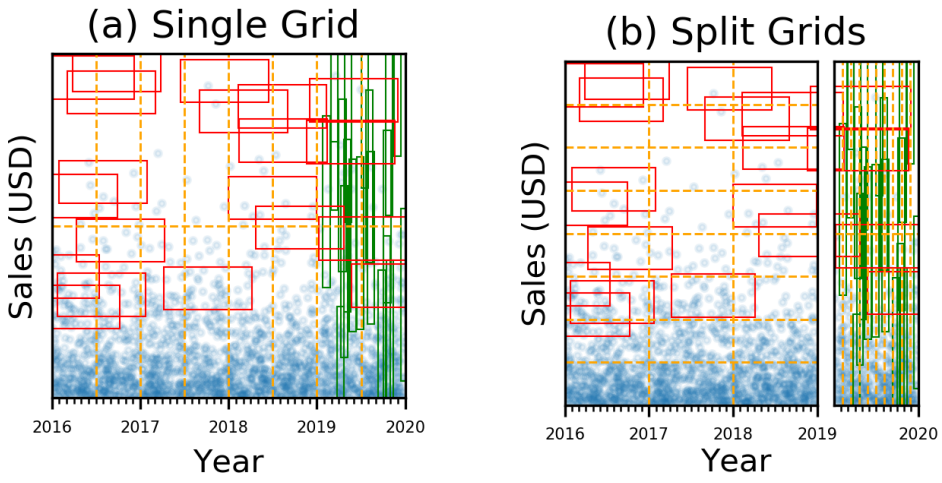

Figure 3-2: A single grid cannot efficiently index a skewed query workload, but a combination of non-overlapping grids can. We use this workload as a running example.

query skew: the Grid Tree.

3.3.1 Challenges of Query Skew

A query workload is skewed if the characteristics of queries (e.g., frequency or selectivity) vary in different parts of the data space. Fig. 3-2a shows an example of sales data from 2016 to 2020. Points are uniformly distributed in time. The query workload is composed of two distinct query "types": the red queries Q_r filter uniformly over one-year spans, whereas the green queries Q_g filter over one-month spans only over the last year. If we were to impose a grid over the data space, we intuitively would want many partitions over the past year in order to obtain finer granularity for Q_g, whereas partitions prior to 2019 should be more widely spaced, because Q_r does not require much granularity in time. However, with a single grid it is not possible accommodate both while maintaining an equal number of points in each partition (Fig. 3-2a).

Instead, we can split the data space into two regions: before 2019 and after 2019 (Fig. 3-2b). Each region has its own grid, and the two grids are independent. The right region can therefore tailor its grid for Q_g by creating many partitions over time. On the other hand, the left region does not need to worry about Q_g at all and places few partitions over time, and can instead add more partitions over the sales dimension. This intuition drives our solution for tackling query skew.

3.3.2 Reducing Query Skew with a Grid Tree

We first formally define query skew. We then describe at a high level how Grid Tree tackles query skew and how to process queries using the Grid Tree. We then describe how to find the optimal Grid Tree for a given dataset and query workload. We use the terminology in Tab. 3.1.

Table 3.1: Terms used to describe the Grid Tree

Term	Description
d	Dimensionality of the dataset
S	d-dimensional data space: $[0, X_0) \times \cdots \times [0, X_{d-1})$
Q	Set of queries
$Uni_i(a, b)$	Uniform distribution over $[a, b)$ in dimension $i \in [0, d)$
$PDF_i(Q, a, b)$	Empirical PDF of queries Q over $[a, b)$ in dimension i
$Hist_i(Q, a, b, n)$	Approximate PDF of queries Q over range $[a, b)$ in dimension i using a histogram with n bins
$EMD(P_1, P_2)$	Earth Mover's Distance between distributions P_1, P_2
$Skew_i(Q, a, b)$	Skew of query set Q over range $[a, b)$ in dimension i

Definition of Query Skew

The skew of a set of queries Q with respect to a range $[a, b)$ in dimension i is

$$Skew_i(Q, a, b) = EMD\left(Uni_i(a, b), PDF_i(Q, a, b)\right)$$

where $Uni_i(a, b)$ is a uniform distribution over $[a, b)$ and $PDF_i(Q, a, b)$ is the empirical PDF of queries in Q over $[a, b)$. Each query contributes a unit mass to the PDF, spread over its filter range in dimension i. EMD is the Earth Mover's Distance, which is a measure of the distance between two probability distributions.

Fig. 3-3a shows the same data and workload as in Fig. 3-2. Fig. 3-3b-c show the PDF of Q_g and Q_r, respectively. The skew is intuitively visualized (though not technically equal to) the shaded area between the PDF and the uniform distribution. Although Q_g is highly skewed over the time dimension, Fig. 3-3d shows that by splitting the time domain at 2019, we can reduce the skew of Q_g because $Skew_{Year}(Q_g, 2016, 2019)$ and $Skew_{Year}(Q_g, 2019, 2020)$ are low.

In concept, $PDF_i(Q, a, b)$ is a continuous probability distribution. However, in practice we approximate $PDF_i(Q, a, b)$ using a histogram: we discretize the range $[a, b)$ into n bins. If a query q's filter range intersects with m contiguous bins, then it contributes $1/m$ mass to each of the bins. Therefore, the total histogram mass will be $|Q|$. We call this histogram $Hist_i(Q, a, b, n)$.

In this context, a probability distribution over a range of histogram bins $[x, y)$, where $0 \le x < y \le n$, is a $(y - x)$-dimensional vector. We can concretely compute skew over the bins $[x, y)$:

$$\begin{aligned}
Uni_i(Q, x, y)[j] &= \frac{\sum_{x \le k < y} Hist_i(Q, a, b, n)[k]}{y - x} \qquad \text{for } x \le j < y \\
PDF_i(Q, x, y)[j] &= Hist_i(Q, a, b, n)[j] \qquad \text{for } x \le j < y \\
Skew_i(Q, x, y) &= EMD\left(Uni_i(Q, x, y), PDF_i(Q, x, y)\right)
\end{aligned}$$

We store the bin boundaries of the histogram, so there is a simple mapping function from a value a to its bin x. Therefore, throughout this section, we will use $Skew_i(Q, a, b)$

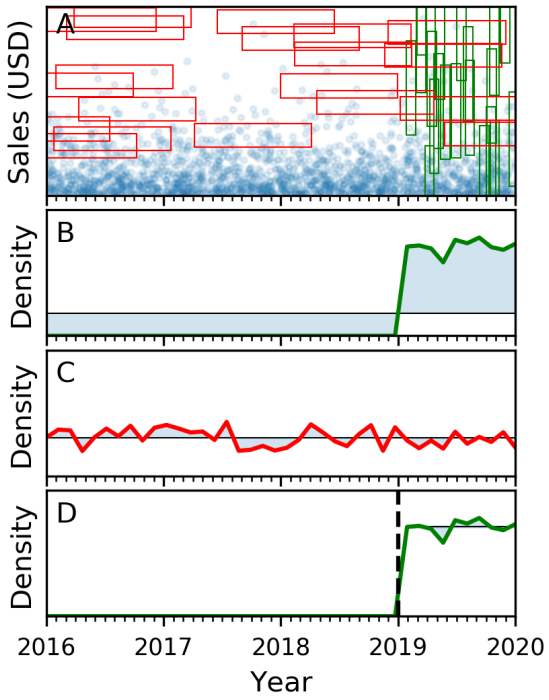

Figure 3-3: Query skew is computed independently for each query type (Q_g and Q_r) and is defined as the statistical distance between the empirical PDF of the queries and the uniform distribution.

and $Skew_i(Q, x, y)$ interchangeably.

Grid Tree Design

Given a query workload that is skewed over a data space, the aim of the Grid Tree is to divide the data space into a number of non-overlapping *regions* so that within each region, there is little query skew.

The Grid Tree is a space-partitioning decision tree, similar to a k-d tree. Each internal node of the Grid Tree divides space based on the values in a particular dimension, called the split dimension d_s. Unlike a k-d tree, which is a binary tree, internal nodes of the Grid Tree can split on more than one value. If an internal node splits on values $V = \{v_1, \ldots, v_k\}$, then the node has $k + 1$ children. To process a query, we traverse the Grid Tree to find all regions that intersect with the query's filter predicates. If there is an index over the points in that region (e.g., an Augmented Grid), then we delegate the query to that index and aggregate the returned results. If there is no index for the region, we simply scan all points in the region.

Note that the Grid Tree is not meant to be an end-to-end index. Instead, the Grid Tree's purpose is to efficiently reduce query skew, while using low memory. This way, the user is free to use any indexing scheme within each region, without worrying about intra-region query skew.

3.3.3 Optimizing the Grid Tree

Given a dataset and sample query workload, our optimization goal is to reduce query skew as much as possible while maintaining a small and lightweight Grid Tree. We present the high-level optimization algorithm, then dive into details. Our procedure is as follows: (1) Group queries in the sample workload into some number of clusters, which we call query *types* (Chapter 3.3.3). (2) Build the Grid Tree in a greedy fashion. Start with a root node that is responsible for the entire data space S. Recursively, for each node N responsible for data space S_N, pick the split dimension $d_s \in [0, d)$ and the set of split values $V = \{v_1, \ldots, v_k\}$ that most reduce query skew (Chapter 3.3.3). d_s and V define $k+1$ non-overlapping sub-spaces of S_N. Assign a child node to each of the $k+1$ sub-spaces and recurse for each child node. If a node N has low query skew (Chapter 3.3.3), or has below a minimum threshold number of intersecting points or queries, then it stops recursing and becomes a leaf node, representing a *region*.

Clustering Query Types

It is not enough to consider the query skew of the entire query set Q as a whole, because queries within this set have different characteristics and therefore are best indexed in different ways. For example, we showed in Fig. 3-2 that Q_g and Q_r are best indexed with different partitioning schemes. Considering all queries as a whole can mask the effects of skew because the skews of different query types can cancel each other out.

Therefore, we cluster queries into *types* that have similar selectivity characteristics. First, queries that filter over different sets of dimensions are automatically placed in different types. For each group of queries that filter over the same set of d' dimensions, we transform each query into a d'-dimensional embedding in which each value is set to the filter selectivity of the query over a particular dimension. We run DBSCAN over the d'-dimensional embeddings with eps set to 0.2 (this worked well for all our experiments and we never tuned it). DBSCAN automatically determines the number of clusters. The choice of clustering algorithm is orthogonal to the Grid Tree design.

Real query workloads have patterns and can usually be divided into types. For example, many analytic workloads are composed of query templates, for which the dimensions filtered and rough selectivity remains constant, but the specific predicate values vary. However, even if there are no patterns in the workload, the Grid Tree is still useful because there can still be query skew over a single query type (i.e., query frequency varies in different parts of data space).

From now on, we assume that if the query set Q is composed of t query types, then we can divide Q into t subsets $Q_1, \ldots, Q_t$. For example, in Fig. 3-3 there are 2 types, Q_r and Q_g. Note that each query can only belong to one query type, but queries in different types are allowed to overlap in data space. We now redefine skew:

$$Skew_i(Q, a, b) = \sum_{1 \le i \le t} Skew_i(Q_t, a, b)$$

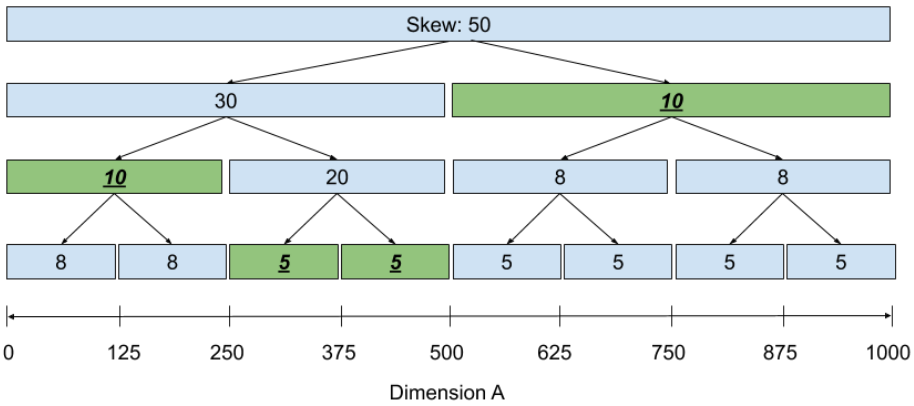

Figure 3-4: Skew tree over the range $[0, 1000)$ with eight leaf nodes. The covering set that achieves lowest combined skew is shaded green. Based on the boundaries of the covering set, we extract the split values $V = \{250, 375, 500\}$.

Selecting the Split Dimension and Values

Given a Grid Tree node N over a data space S_N and a set of queries Q that intersects with S_N, our goal is to find the split dimension d_s and split values over that dimension $V = \{v_1, \ldots, v_k\}$ that achieve the largest reduction in query skew. For a dimension $i \in [0, d)$ and split values V, the reduction in query skew is defined as

$$R_i(Q, 0, X_d, V) = Skew_i(Q, 0, X_d) - \Big[Skew_i(Q, 0, v_1) + Skew_i(Q, v_k, X_d) + \sum_{1 \le i < k} Skew_i(Q, v_i, v_{i+1})\Big]$$

Note that skew reduction is defined *per dimension*, not over all d dimensions simultaneously. Therefore, we independently find the largest skew reduction $Rmax_i = \max_V(R_i)$ for each dimension $i \in [0, d)$ (explained next), then pick the split dimension $d_s = \arg\max_i(Rmax_i)$.

For example, in Fig. 3-3, $Skew_{Sales}$ is already low because both query types are distributed relatively uniformly over Sales, so $Rmax_{\text{Sales}}$ is low. On the other hand, $Skew_{Year}$ is high. We can achieve very large $Rmax_{\text{Year}}$ using $V = \{2019\}$. Therefore, we select $d_s = \text{Year}$ and $V = \{2019\}$.

If $\max_i(Rmax_i)$ is below some minimum threshold (by default 5% of $|Q|$) or if S_N intersects below a minimum threshold of points or queries (by default 1% of the total points or queries in the entire data space), then d_s is rejected and N becomes a leaf Grid Tree node.

We now explain how to find the split values V that maximize R_{d_s} for each candidate split dimension $d_s \in [0, d)$. We introduce a data structure called the *skew tree*, which is simply a tool to help find the optimal V; it is never used when running queries. The skew tree is a balanced binary tree (Fig. 3-4). Each node represents a range over the domain of dimension d_s. The root node represents the entire range $[0, X_{d_s})$, and every node represents the combined ranges of the nodes in its subtree. A skew tree

node whose range is $[a, b)$ will store the value $Skew_{d_s}(Q, a, b)$. In other words, each skew tree node stores the query skew over the range it represents.

Creating the skew tree requires $Hist_{d_s}(Q, 0, X_{d_s})$. By default, we instantiate the histogram with 128 bins. Note that we are unable to compute a meaningful skew over a single histogram bin: $Skew_{d_s}(Q, x, x+1)$ is always zero, because a single bin has no way to differentiate the uniform distribution from the query PDF. Therefore, the skew tree will only have 64 leaf nodes. However, if there are fewer than 128 unique values in dimension d_s, we create a bin for each unique value. In this case, there is truly no skew within each histogram bin, so the skew tree has as many leaf nodes as unique values in d_s, and the skew at each leaf node is 0.

A set of skew tree nodes is called *covering* if their represented ranges do not intersect and the union of their represented ranges is $[0, X_{d_s})$. We want to solve for the covering set with minimum combined query skew. This is simple to do via dynamic programming in two passes over the skew tree nodes: in the first pass, we start from the leaf nodes and work towards the root node, and at each node we annotate the minimum combined query skew achievable over the node's subtree. In the second pass, we start from the root and work towards the leaves, and check if a node's skew is equal to the annotated skew: if so, the node is part of the optimal covering set. The boundaries between the ranges of nodes in the optimal covering set form V.

As a final step, we do a single ordered pass over all the nodes in the covering set, in order of the range they represent, and merge nodes if the query skew of the combined node is not more than a constant factor (by default, 10%) larger than the sum of the individual query skews. For example, in Fig. 3-4 if $Skew_A(Q, 0, 375) < 15 \cdot 1.1$, then the first two nodes of the covering set would be merged, and 250 would be removed as a split value. This step counteracts the fact that the binary tree may split at superfluous points, and it also acts as a regularizer that prevents too many splits.

3.4 Augmented Grid

In this section, we describe the challenges posed by data correlations, and we introduce our solution to address those challenges: the Augmented Grid. Note that the Grid Tree (Chapter 3.3) optimizes only for query skew reduction, and the points within each *region* might still display correlation.

3.4.1 Challenges of Data Correlation

We broadly define a pair of dimensions X and Y to be correlated if they are not independent, i.e., if $CDF(X) \neq CDF(X|Y)$ and vice versa. In the presence of correlated dimensions, it is not possible to impose a grid that has equally-sized cells by partitioning each dimension independently (see Fig. 3-1b). As a result, points will be clustered into a relatively few number of cells, so any query that hits one of those cells will likely scan many more points than necessary.

One way to mitigate this issue is by increasing the number of partitions in each dimension, to form more fine-grained cells. However, increasing the number of cells

Table 3.2: Example skeleton over dimensions X, Y, Z, and all skeletons one "hop" away. Restrictions are explained in Chapter 3.4.2 and Chapter 3.4.2 (e.g., $[X \to Z, Y|X, Z]$ is not allowed).

Ex. skeleton	$[X, Y\|X, Z]$ (i.e., $CDF(X)$, $CDF(Y\|X)$, and $CDF(Z)$)		
One hop away	$[X, Y, Z]$	$[X, Y\|Z, Z]$	$[X, Y \to X, Z]$
	$[X, Y \to Z, Z]$	$[X, Y\|X, Z\|X]$	$[X, Y\|X, Z \to X]$

would counteract the two advantages of grids over trees: (1) Space overhead increases rapidly (e.g., doubling the number of partitions in each dimension increases index size by 2^d). (2) Time overhead also increases, because each cell incurs a lookup table lookup. Therefore, simply making finer-grained grids is not a scalable solution to data correlations.

3.4.2 A Correlation-Aware Grid

Tsunami handles data correlations while maintaining the time and space advantage of grids by augmenting the basic grid structure with new partitioning strategies that allow it to partition dimensions *dependently* instead of independently. We first provide a high level description of the Augmented Grid, then dive into details.

An Augmented Grid is a grid in which each dimension $X \in [0, d)$ is divided into p_X partitions and uses one of three possible strategies for creating its partitions: (1) We can partition X independently of other dimensions, uniformly in $CDF(X)$. This is what Flood does for every dimension. (2) We can remove X from the grid and transform query filters over X into filters over some other dimension $Y \in [0, d)$ using a functional mapping $F : X \to Y$ (Chapter 3.4.2). (3) We can partition X dependent on another dimension $Y \in [0, d)$, uniformly in $CDF(X|Y)$ (Chapter 3.4.2).

A specific instantiation of partitioning strategies for all dimensions is called a *skeleton*. Tab. 3.2 shows an example. We "flesh out" the skeleton by setting the number of partitions in each dimension to create a concrete instantiation of an Augmented Grid. Therefore, an Augmented Grid is uniquely defined by the combination of its skeleton S and number of partitions in each dimension P.

Functional Mappings

A pair of dimensions X and Y is monotonically correlated if as values in X increase, values in Y only move in one direction. Linear correlations are one subclass of monotonic correlations. For monotonically correlated X and Y, we conceptually define a mapping function as a function $F : \mathbb{R}^2 \to \mathbb{R}^2$ that takes a range $[Y_{min}, Y_{max}]$ over dimension Y and maps it to a range $[X_{min}, X_{max}]$ over dimension X with the guarantee that any point whose value in dimension Y is in $[Y_{min}, Y_{max}]$ will have a value in dimension X in $[X_{min}, X_{max}]$. In this case, we call Y the *mapped dimension* and we call X the *target dimension*. For simplicity, we place a restriction: a target dimension cannot itself be a mapped dimension. Similar ideas were proposed in [88, 193].

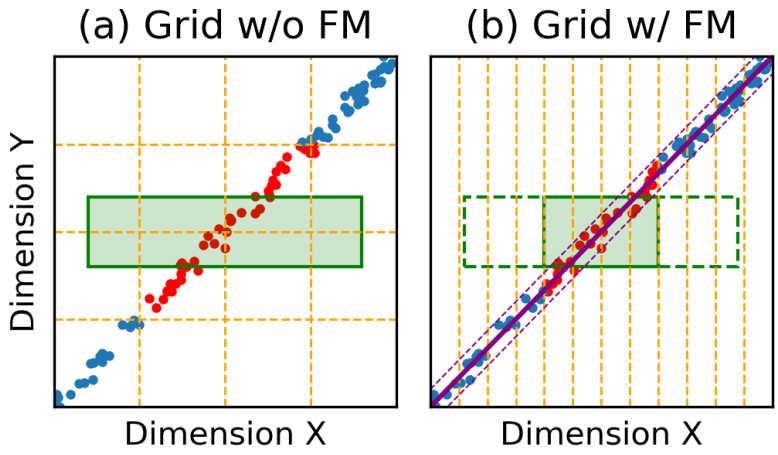

Figure 3-5: Functional mapping creates equally-sized cells and reduces scanned points for tight monotonic correlations. The query is in green, scanned points are red, and the mapping function is purple, with error bounds drawn as dashed lines.

Concretely, we implement the mapping function as a simple linear regression LR trained to predict X from Y, with lower and upper error bounds e_l and e_u. Therefore, a functional mapping is encoded in four floating point numbers and has negligible storage overhead. Given a range $[Y_{min}, Y_{max}]$, the mapping function produces $X_{min} = Y_{min} - e_l$ and $X_{max} = Y_{max} + e_u$. Note that the idea of functional mappings can generalize to all monotonic correlations, as in [193]. However, in our experience the vast majority of monotonic correlations in real data are linear, so we use linear regressions for simplicity.

Given a functional mapping, any range filter predicate $(y_0 \leq Y \leq y_1)$ over dimension Y can be transformed into a semantically equivalent predicate $(x_0 \leq X \leq x_1)$ over dimension X, where $(x_0, x_1) = F(y_0, y_1)$. This gives us the opportunity to completely remove the mapped dimension from the d-dimensional grid, to obtain equally-sized cells. Fig. 3-5 demonstrates the benefits of functional mapping. The grid without functional mapping has unequally-sized cells, which results in many points scanned. On the other hand, the grid with functional mapping has equally-sized cells and is furthermore able to "shrink" the size of the query to a semantically equivalent query by *inducing* a narrower filter over dimension X using the mapping function. This results in fewer points scanned.

Conditional CDFs

Functional mappings are only useful for tight monotonic correlations. Otherwise, the error bounds would be too large for the mapping to be useful. For loose monotonic correlations or generic correlations, we instead use conditional CDFs. For a pair of generically correlated dimensions X and Y, we partition X uniformly in $CDF(X)$ and we partition Y uniformly in $CDF(Y|X)$, resulting in equally-sized cells. In this case, we call X the *base dimension* and Y the *dependent dimension*. For simplicity,

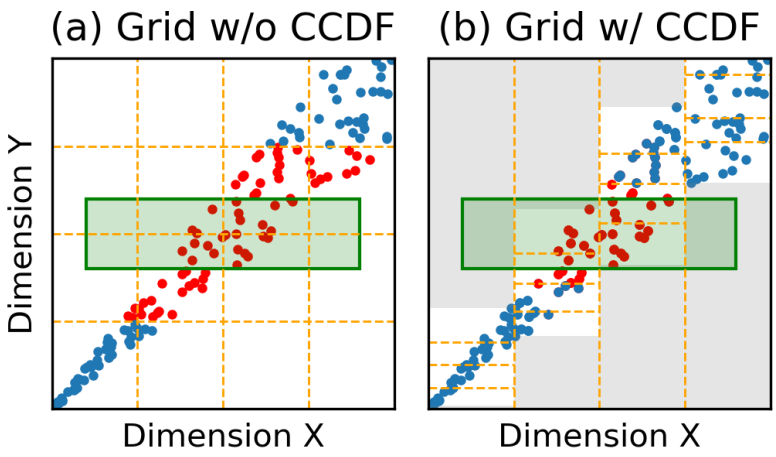

Figure 3-6: Conditional CDFs create equally-sized cells and reduce scanned points for generic correlations. The query is in green, and scanned points are in red.

we place restrictions: a base dimension cannot itself be a mapped dimension or a dependent dimension.

Concretely, if there are p_X and p_Y partitions over X and Y respectively, we implement $CDF(Y|X)$ by storing p_X histograms over Y, one for each partition in X. When a query filters over Y, we first find all intersecting partitions in X, then for each X partition independently invoke $CDF(Y|X)$ to find the intersecting partitions in Y. The storage overhead is proportional to $p_X p_Y$, which is minimal compared to the existing overhead of the grid's lookup table, which is proportional to $\prod_{i\in[0,d)} p_i$.

Fig. 3-6 shows an example of using conditional CDFs. Both grids have $p_X = p_Y = 4$. By partitioning Y using $CDF(Y|X)$, the grid on the right has staggered partition boundaries, which create equally-sized cells and results in fewer points scanned. Additionally, the regions outside the cells (shaded in gray) are guaranteed to have no points, which allows the query to avoid scanning the first and last partitions of X, even though they intersect the query.

3.4.3 Optimizing the Augmented Grid

Given a dataset and sample query workload, our optimization goal is to find the best Augmented Grid, i.e., the settings of the parameters (S, P) that achieves lowest average query time over the sample workload, where S is the skeleton and P is the number of partitions in each dimension.

This optimization problem is challenging in two ways: (1) For a specific setting of (S, P), we cannot know the average query time without actually running the queries, which can be very time-intensive. Therefore, we create a cost model to predict average query time, and we optimize for lowest average *predicted* query time (Chapter 3.4.3). (2) The search space over skeletons is exponentially large. For each dimension, there are $O(d)$ possible partitioning strategies, since there are up to $d-1$ choices for the other dimension in a functional mapping or conditional CDF. Therefore, the search

space of skeletons has size $O(d^d)$. To efficiently navigate the joint search space of (S, P), we use *adaptive gradient descent* (Chapter 3.4.3).

Cost Model

We use a simple analytic linear cost model to predict the runtime of a query q on dataset D and an instantiation of the Augmented Grid with parameters (S, P):

$$\text{Time} = w_0(\#\text{ cell ranges}) + w_1(\#\text{ scanned points})(\#\text{ filtered dims})$$

We now explain each term of this model. A set of adjacent cells in physical storage is called a *cell range*. Instead of doing a lookup on the lookup table for every intersecting cell, we only look up the first and last cell of a cell range. Furthermore, skipping to each new cell range in physical storage likely incurs a cache miss. w_0 represents the time to do a lookup and the cache miss of accessing the range in physical storage.

The w_1 term models the time to scan points (e.g., all red points in previous figures). Since data is stored in a column store, only the dimensions filtered by the query need to be accessed. w_1 represents the time to scan a single dimension of a single point.

Importantly, the features of this cost model can be efficiently computed or estimated: the number of cell ranges is easily computed from q and (S, P). The number of filtered dimensions is obvious from q. The number of scanned points is estimated using q, (S, P), and a sample of D.

Note that we do not model the time to actually perform the aggregation after finding the points that intersect the query rectangle. This is because aggregation is a fixed cost that must be incurred regardless of index choice, so we ignore it when optimizing.

Adaptive Gradient Descent

We find the (S, P) that minimizes average query time, as predicted by the cost model, using adaptive gradient descent (AGD). We first enumerate AGD's high level steps, then provide details for each step. AGD is an iterative algorithm that jointly optimizes S and P:

1. Using heuristics, initialize (S_0, P_0).
2. From (S_0, P_0), take a gradient descent step over P_0 using the cost model as the objective function, which gives us (S_0, P_1).
3. From (S_0, P_1), perform a local search over skeletons to find the skeleton S' that minimizes query time for (S', P_1). Set $S_1 = S'$. It may be that $S' = S_0$, that is, the skeleton does not change in this step.
4. Repeat steps 2 and 3 starting from (S_1, P_1) until we reach a minimum average query time.

In step 1, we first initialize S, then P. We make a best guess at the optimal skeleton using heuristics: for each dimension X, use a functional mapping to dimension Y if the error bound is below 10% of Y's domain. Else, partition using $CDF(X|Y)$ if not doing so would result in more than 25% of cells in the XY grid hyperplane being empty. Else, partition X independently using $CDF(X)$. Given the initial S, we

initialize P proportionally to the average query filter selectivity in each grid dimension (i.e., excluding mapped dimensions).

In step 2, we use the insight that the cost model is relatively smooth in P: changing the number of partitions usually smoothly increases or decreases the cost. Therefore, we take the numerical gradient over P at (S, P) and take a step in the gradient direction.

In step 3, we take advantage of the insight that an incremental change in P is unlikely to cause the skeleton S' to differ greatly from S. Therefore, step 3 will only search over S' that can be created by changing the partitioning strategy for a single dimension in S (e.g., skeletons one "hop" away in Tab. 3.2).

While we could conceivably use black box optimization methods such as simulated annealing to optimize (S, P), AGD takes advantage of the aforementioned insights into the behavior of the optimization and is therefore able to find lower-cost Augmented Grids, which we confirm in Chapter 3.5.6.

3.5 Evaluation

We first describe the experimental setup and then present the results of an in-depth experimental study that compares Tsunami with Flood and several other indexing methods on a variety of datasets and workloads. Overall, this evaluation shows that:

1. Tsunami is consistently the fastest index across tested datasets and workloads. It achieves up to 6× higher query throughput than Flood and up to 11× higher query throughput than the fastest optimally-tuned non-learned index. Furthermore, Tsunami has up to 8× smaller index size than Flood and up to 170× smaller index size than the fastest non-learned index (Chapter 3.5.3).
2. Tsunami can optimize its index layout and reorganize the records quickly for a new query distribution, typically in under 4 minutes for a 300 million record dataset (Chapter 3.5.4).
3. Tsunami's performance advantage over other indexes scales with dataset size, selectivity, and dimensionality (Chapter 3.5.5).

3.5.1 Implementation and Setup

We implement Tsunami in C++ and perform optimization in Python. We perform our query performance evaluation via single-threaded experiments on an Ubuntu Linux machine with Intel Core i9-9900K 3.6GHz CPU and 64GB RAM. Optimization and data sorting for index creation are performed in parallel for Tsunami and all baselines.

All experiments use 64-bit integer-valued attributes. Any string values are dictionary encoded prior to evaluation. Floating point values are typically limited to a fixed number of decimal points (e.g., 2 for price values). We scale all values by the smallest power of 10 that converts them to integers.

Evaluation is performed on data stored in a custom column store with one scan-time optimization: if the range of data being scanned is *exact*, i.e., we are guaranteed ahead of time that all elements within the range match the query filter, we skip checking

Table 3.3: Dataset and query characteristics.

	TPC-H.	**Taxi**	**Perfmon**	**Stocks**
records	300M	184M	236M	210M
query types	5	6	5	5
dimensions	8	9	7	7
size (GB)	19.2	13.2	13.2	11.8

each value against the query filter. For common aggregations, e.g. `COUNT`, this removes unnecessary accesses to the underlying data.

We compare Tsunami to other solutions implemented on the same column store, with the same optimizations, if applicable:

1. *Clustered Single-Dimensional Index*: Points are sorted by the most selective dimension in the query workload. If a query filter contains this dimension, we locate the endpoints using binary search. Otherwise, we perform a full scan.
2. The *Z-Order Index* is a multidimensional index that orders points by their *Z-value* [59]; contiguous chunks are grouped into pages. Given a query, the index finds the smallest and largest Z-value contained in the query rectangle and iterates through each page with Z-values in this range. Pages maintain min/max metadata per dimension to prune irrelevant pages.
3. The *Hyperoctree* [129] recursively subdivides space equally into hyperoctants (the d-dimensional analog to 2-dimensional quadrants), until the number of points in each leaf is below a predefined but tunable page size.
4. The *k-d tree* [17] recursively partitions space using the median value along each dimension, until the number of points in each leaf falls below the page size. The dimensions are selected in a round robin fashion, in order of selectivity.
5. *Flood*, introduced in Chapter 3.1.2. We use the implementation of [138] with two changes: we use Tsunami's cost model instead of Flood's original random-forest-based cost model, and we perform refinement using binary search instead of learned per-cell models (see [138] for details). We verified that these changes did not meaningfully impact performance. Furthermore, removing per-cell models dramatically reduces Flood's index size (on average by 20× [138]), and this allows us to more directly evaluate the impact of design differences between Flood and Tsunami without any confounding effects from implementation differences.

There are a number of other multi-dimensional indexing techniques, such as Grid Files [143], UB-tree [160], and R*-Tree [14]. We decided not to evaluate against these because Flood already showed consistent superiority over them [138]. We also do not evaluate against other learned multi-dimensional indexes because they are either optimized for disk [194, 108] or optimize only based on the data distribution, not the query workload [187, 42] (see Chapter 3.6).

3.5.2 Datasets and Workloads

We evaluate indexes on three real-world and one synthetic dataset, summarized in Tab. 3.3. Queries are synthesized for each dataset, and include a mix of range filters and equality filters. The queries for each dataset comes from a certain number of query types (Chapter 3.3.3), each of which answers a different analytics question, with 100 queries of each type. All queries perform a `COUNT` aggregation. Since all indexes must pay the same fixed cost of aggregation, performing different aggregations would not change the relative ordering of indexes in terms of query performance.

The **Taxi** dataset comes from records of yellow taxi trips in New York City in 2018 and 2019 [144]. It includes fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, and driver-reported passenger counts. Our queries answer questions such as "How common were single-passenger trips between two particular parts of Manhattan?" and "What month of the past year saw the most short-distance trips?". Queries display skew over time (more queries over recent data), passenger count (different query types about very low and very high passenger counts), and trip distance (more queries about very short trip distances). Query selectivity varies from 0.25% to 3.9%, with an average of 1.3%.

The performance monitoring dataset **Perfmon** contains logs of all machines managed by a major US university over the course of a year. It includes fields capturing log time, machine name, CPU usages, and system load averages. Our queries answer questions such as "When in the last month did a certain set of machines experience high load?". Queries display skew over time (more queries over recent data) and CPU usage (more queries over high usage). Query selectivity varies from 0.50% to 4.9%, with an average of 0.79%. The original dataset has 23.6M records, but we use a scaled dataset with 236M records.

The **Stocks** dataset consists of daily historical stock prices of over 6000 stocks from 1970 to 2018 [55]. It includes fields capturing daily prices (open, close, adjusted close, low, and high), trading volume, and the date. Our queries answer questions such as "Which stocks saw the lowest intra-day price change while trading at high volume?" and "What one-year span in the past decade saw the most stocks close in a certain price range?". Queries display skew over time (more queries over recent data) and volume (different query types about very low and very high volume). Query selectivity is tightly concentrated around $0.5\% \pm 0.04\%$. The original dataset has 21M records, but we use a scaled dataset with 210M records.

Our last dataset is **TPC-H** [180]. For our evaluation, we use only the fact table, `lineitem`, with 300M records (scale factor 50) and create queries by using filters commonly found in the TPC-H query workload. Our queries include filters over quantity, extended price, discount, tax, ship mode, ship date, commit date, and receipt date. They answer questions such as "How many high-priced orders in the past year used a significant discount?" and "How many shipments by air had below ten items?". Query selectivity varies from 0.40% to 0.64%, with an average of 0.54%.

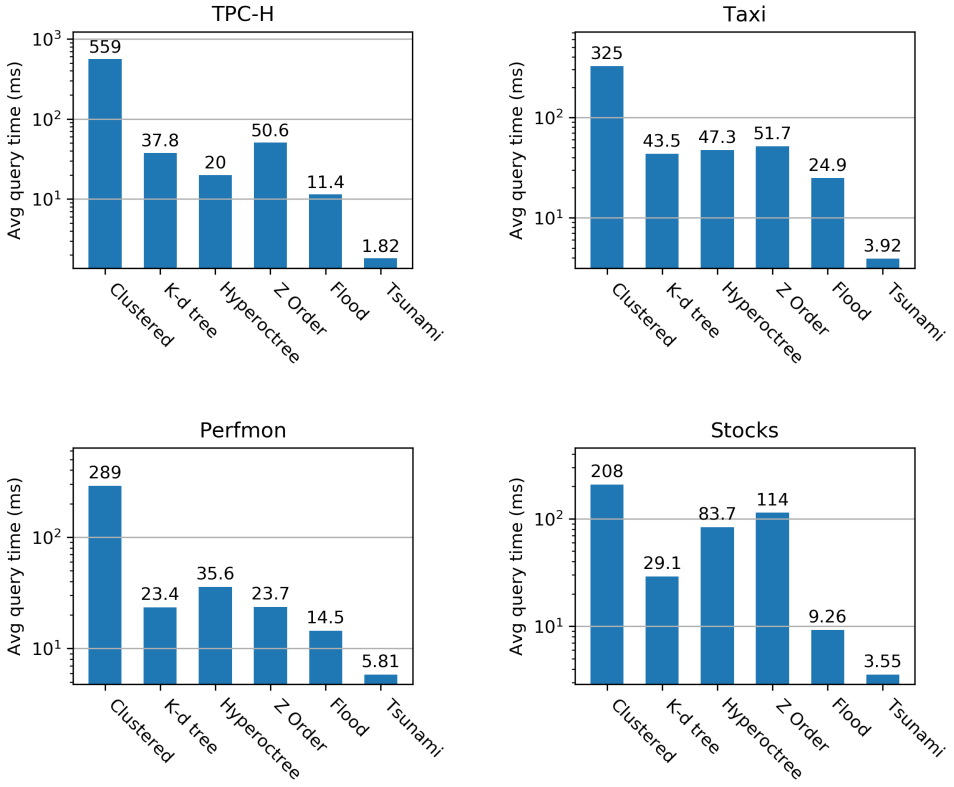

Figure 3-7: Tsunami achieves up to 6× faster queries than Flood and up to 11× faster queries than the fastest non-learned index.

3.5.3 Overall Results

Fig. 3-7 compares Tsunami to Flood and the non-learned baselines. Tsunami and Flood are automatically optimized for each dataset/workload. For the non-learned baselines, we tuned the page size to achieve best performance on each dataset/workload. Tsunami is consistently the fastest of all the indexes across datasets and workloads, and achieves up to 6× faster queries than Flood and up to 11× faster queries than the fastest non-learned index.

Tab. 3.4 shows statistics of the optimized Tsunami index structure. The Grid Tree depth and the number of leaf regions are relatively low, which confirms that the Grid Tree is lightweight, as desired. Because skew does not occur uniformly across data space, the number of points in each region can vary by over an order of magnitude.

The Grid Tree typically has a low number of nodes (Tab. 3.4), so the vast majority of Tsunami's index size comes from the cell lookup tables for the Augmented Grids in each region. Tsunami often has fewer total grid cells than Flood (Tab. 3.4) because partitioning space via the Grid Tree gives Tsunami fine-grained control over the number of cells to allocate in each region, whereas Flood must often over-provision partitions to deal with query skew (see Chapter 3.3.1). Fig. 3-8 shows that as a result of having fewer cells, Tsunami uses up to 8× less memory than Flood. Furthermore,

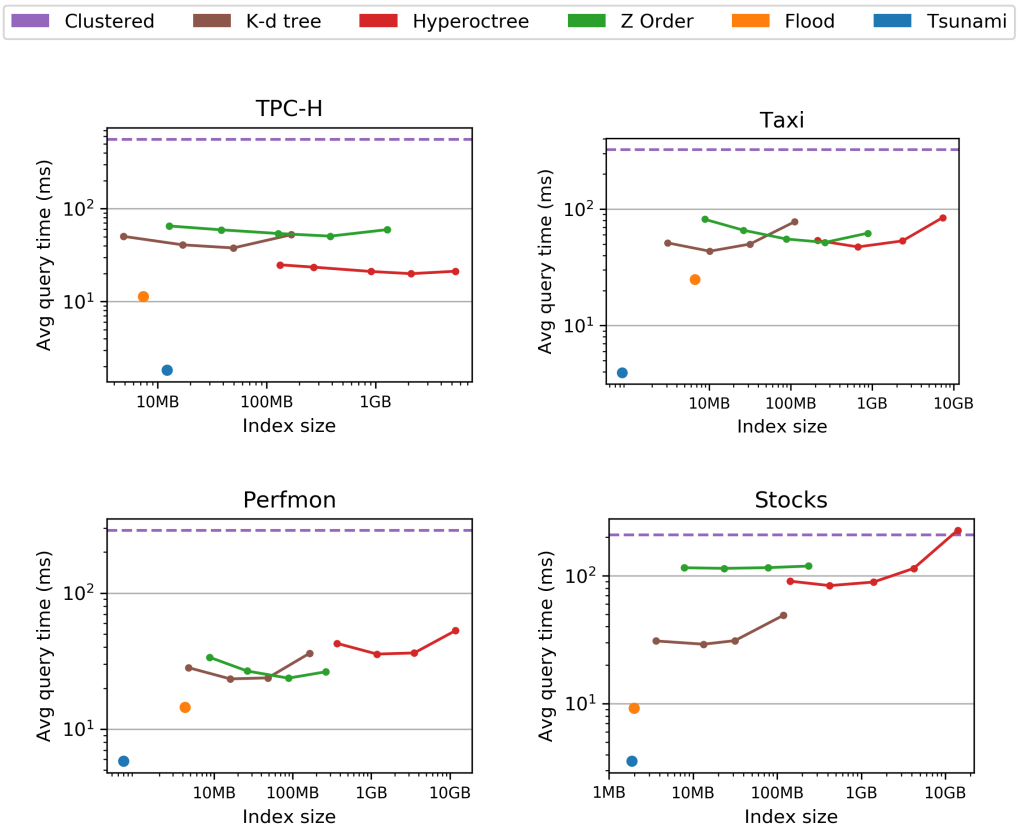

Figure 3-8: Tsunami uses up to 8× less memory than Flood and 7-170× less memory than the fastest tuned non-learned index.

Table 3.4: Index Statistics after Optimization.

	TPC-H	**Taxi**	**Perfmon**	**Stocks**
Tsunami				
Num Grid Tree nodes	39	35	42	54
Grid Tree depth	4	2	4	4
Num leaf regions	27	31	36	39
Min points per region	3.5M	1.9M	2.6M	2.4M
Median points per region	5.9M	3.3M	3.7M	3.2M
Max points per region	10M	6.7M	26M	41M
Avg FMs per region	0.67	0.55	0	1.1
Avg CCDFs per region	1.3	1.9	1.75	1.8
Total num grid cells	1.5M	99K	80K	220K
Flood				
Num grid cells	920K	840K	530K	250K

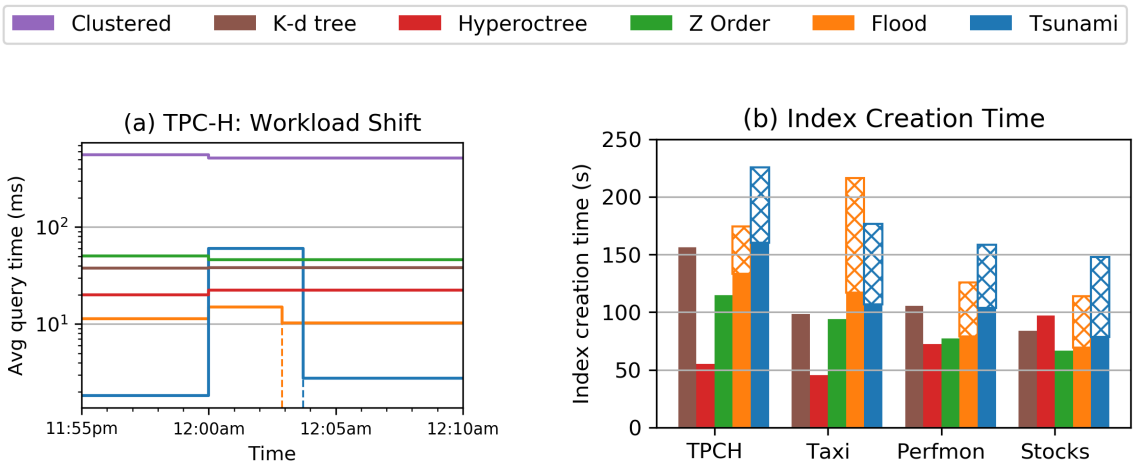

Figure 3-9: (a) After the query workload changes at midnight, Tsunami re-optimizes and re-organizes within 4 minutes to maintain high performance. (b) Comparison of index creation times (solid bars = data sorting time, hatched bars = optimization time).

Tsunami is between 7× to 170× smaller than the fastest optimally-tuned non-learned index across the four datasets.

3.5.4 Adaptibility

Tsunami is able to quickly adapt to changes in the query workload by re-optimizing its layout for the new query workload and re-organizing the data based on the new layout. In Fig. 3-9a, we simulate a scenario in which the query workload over the TPC-H dataset changes at midnight: the original query workload is replaced by a new workload with queries drawn from five new query types. This causes performance on the learned indexes to degrade. Tsunami (as well as Flood) automatically detects the workload shift (see Chapter 3.7) and triggers a re-optimization of the index layout for the new query workload. Tsunami's re-optimization and data re-organization over 300M rows finish within 4 minutes, and its high query performance is restored. This shows that Tsunami is highly adaptive for scenarios in which the data or workload changes infrequently (e.g., every day). The non-learned indexes are not re-tuned after the workload shift, because in practical settings, it is unlikely that a database administrator will be able to manually tune the index for every workload change.

Fig. 3-9b shows the index creation time in detail for Tsunami and the baselines. All indexes require time to sort the data based on the index layout, shown as solid bars. The learned approaches additionally require time to perform optimization based on the dataset and query workload, shown as the hatched bars. Even for the largest datasets, the entire index creation time for Tsunami remains below 4 minutes.

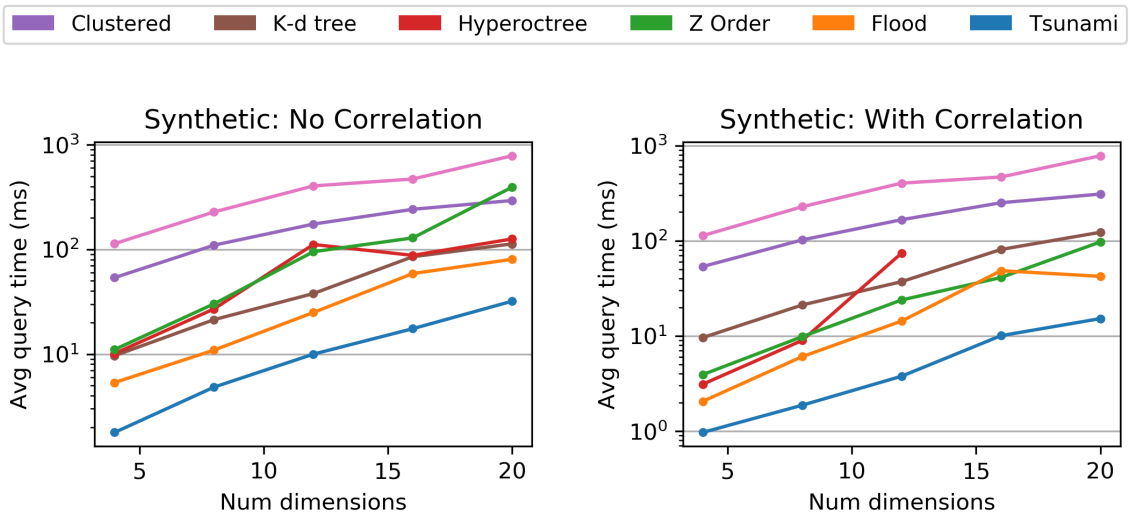

Figure 3-10: Tsunami continues to outperform other indexes at higher dimensions.

3.5.5 Scalability

Throughout this subsection, Tsunami and Flood are re-optimized for each dataset/-workload configuration, while the non-learned indexes use the same page size and dimension ordering as they were tuned for the full TPC-H dataset/workload in Chapter 3.5.3.

Number of Dimensions. To show how Tsunami scales with dimensions, and how correlation affects scalability, we create two groups of synthetic d-dimensional datasets with 100M records. Within each group, datasets vary by number of dimensions ($d \in \{4, 8, 12, 16, 20\}$). Datasets in the first group show no correlation and points are sampled from i.i.d. uniform distributions. For datasets in the second group, half of the dimensions have uniformly sampled values, and dimensions in the other half are linearly correlated to dimensions in the first half, either strongly ($\pm 1\%$ error) or loosely ($\pm 10\%$ error). For each dataset, we create a query workload with four query types. Earlier dimensions are filtered with exponentially higher selectivity than later dimensions, and queries are skewed over the first four dimensions.

Fig. 3-10 shows that in both cases, Tsunami continues to outperform the other indexes at higher dimensions. In particular, the Augmented Grid is able to take advantage of correlations to effectively reduce the dimensionality of the dataset. This helps Tsunami delay the curse of dimensionality: Tsunami has around the same performance on each d-dimensional correlated dataset as it does on the $(d-4)$-dimensional uncorrelated dataset.

Dataset Size. To show how Tsunami scales with dataset size, we sample records from the TPC-H dataset to create smaller datasets. We run the same query workload as on the full dataset. Fig. 3-11a shows that across dataset sizes, Tsunami maintains its performance advantage over Flood and non-learned indexes.

Query Selectivity. To show how Tsunami performs at different query selectivities, we

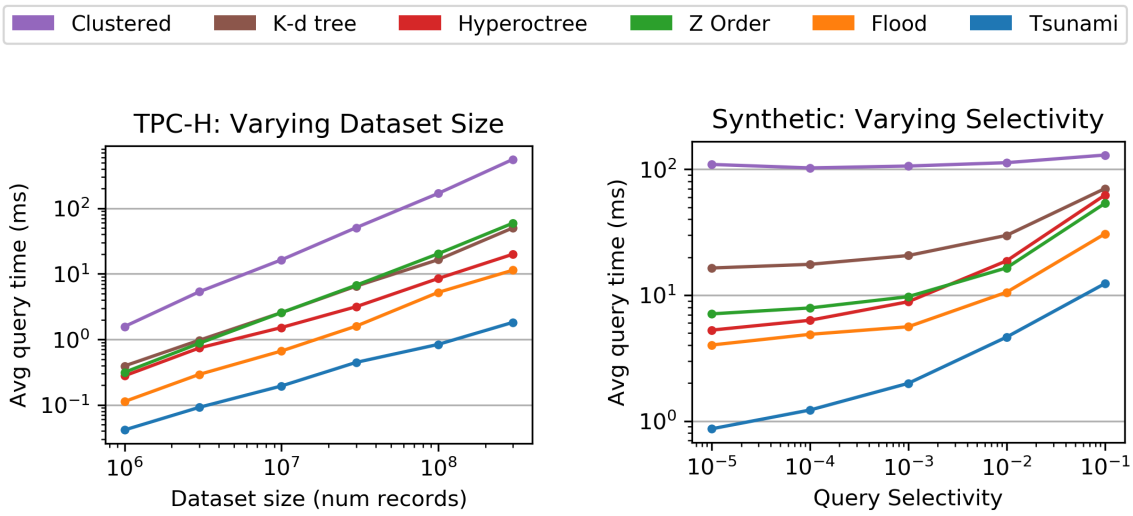

Figure 3-11: Tsunami maintains high performance across dataset sizes and query selectivities.

use the 8-dimensional synthetic dataset/workload with correlation (explained above) and scale filter ranges equally in each dimension in order to achieve between 0.001% and 10% selectivity. Fig. 3-11b shows that Tsunami performs well at all selectivities. The relative performance benefit of Tsunami is less apparent at 10% selectivity because aggregation time becomes a bottleneck.

3.5.6 Drill-down into Components

Fig. 3-12a shows the relative performance of only using the Augmented Grid (i.e., one Augmented Grid over the entire data space) and of only using Grid Tree (i.e., with an instantiation of Flood in each leaf region). Grid Tree contributes the most to Tsunami's performance, but Augmented Grid also boosts performance significantly over Flood. Grid Tree-only performs almost as well as Tsunami because partitioning data space via the Grid Tree often already has the unintentional but useful side effect of mitigating data correlations.

We now evaluate Augmented Grid's optimization procedure, which can be broken into two independent parts: the accuracy of the cost model (Chapter 3.4.3) and the ability of Adaptive Gradient Descent (Chapter 3.4.3) to minimize cost (i.e., average query time, predicted by the cost model). For each of our four datasets/workloads, we run Adaptive Gradient Descent (AGD) to find a low-cost Augmented Grid over the entire data space. We compare with three alternative optimization methods, all using the same cost model:

1. *Gradient Descent (GD)* uses the same initial (S_0, P_0) as AGD, then performs gradient descent over P, without ever changing the skeleton.
2. *Black Box* starts with the same initial (S_0, P_0) as AGD, then optimizes S and P according to the basin hopping algorithm, implemented in SciPy [168], for 50 iterations.

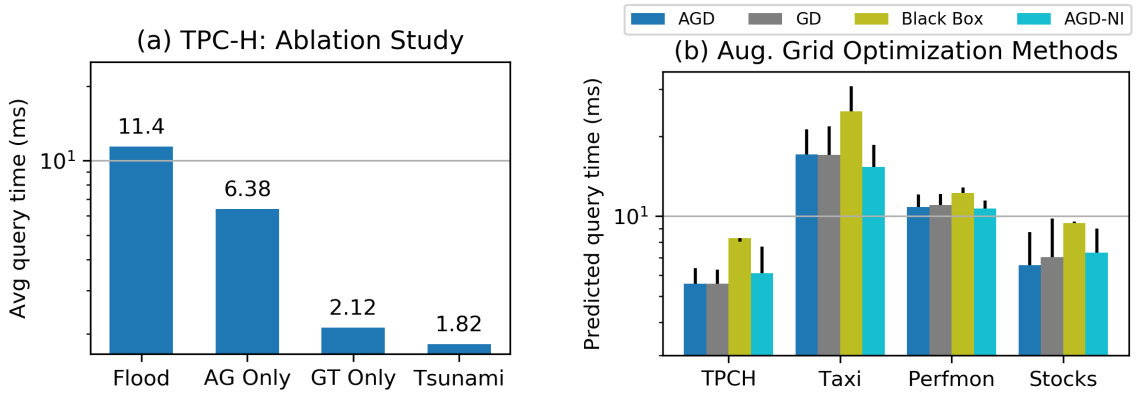

Figure 3-12: (a) Augmented Grid and Grid Tree both contribute to Tsunami's performance. (b) Comparison of optimization methods. Bars show the predicted query time according to our cost model. Error bars show the actual query time.

3. *AGD with naive initialization (AGD-NI)* sets the initial skeleton S_0 to use $CDF(X)$ for each dimension, then runs AGD.

Fig. 3-12b shows the lowest cost achieved by each optimization method. There are several insights. First, Black Box performs worse than the gradient descent variants, which implies that using domain knowledge and heuristics to guide the search process provides an advantage. Second, Adaptive Gradient Descent usually achieves only marginally better predicted query time than Gradient Descent, which implies that for our tested datasets, our heuristics created a good initial skeleton S_0. Third, Adaptive Gradient Descent is able to find a low-cost grid even when starting from a naive skeleton, which implies that the local search over skeletons is able to effectively switch to better skeletons. For the Taxi dataset, AGD-NI is even able to find a lower-cost configuration than AGD.

Fig. 3-12b additionally shows the error between the predicted query time using the cost model and the actual query time when running the queries of the workload. The average error of the model for all optimized configurations shown in Fig. 3-12b is only 15%.

3.6 Related Work

Traditional Multi-dimensional Indexes. There is a rich corpus of work dedicated to multi-dimensional indexes, and many commercial database systems have turned to multi-dimensional indexing schemes. For example, Amazon Redshift organizes points by Z-order [133], which maps multi-dimensional points onto a single dimension for sorting [9, 150, 198]. With spatial dimensions, SQL Server allows Z-ordering [131], and IBM Informix uses an R-Tree [76]. Other multi-dimensional indexes include K-d trees, octrees, R* trees, UB trees (which also make use of the Z-order), and Grid Files [143], among many others (see [146, 174] for a survey). There has also been

work on automatic index selection [116, 24, 183]. However, these approaches mainly focus on creating secondary indexes, whereas Tsunami co-optimizes the index and data storage.

Learned Indexes. Recent work by Kraska et al. [98] proposed the idea of replacing traditional database indexes with learned models that predict the location of a key in a dataset. Their learned index, called the Recursive Model Index (RMI), and various improvements on the RMI [46, 57, 91, 61, 178], only handle one-dimensional keys.

Since then, there has been a corpus of work on extending the ideas of the learned index to spatial and multi-dimensional data. The most relevant work is Flood [138], described in Chapter 3.1.2. Learning has also been applied to the challenge of reducing I/O cost for disk-based multi-dimensional indexes. Qd-tree [194] uses reinforcement learning to construct a partitioning strategy that minimizes the number of disk-based blocks accessed by a query. LISA [108] is a disk-based learned spatial index that achieves low storage consumption and I/O cost while supporting range queries, nearest neighbor queries, and insertions and deletions. Tsunami and these works share the idea that a multi-dimensional index can be instance-optimized for a particular use case by learning from the dataset and query workload.

Past work has also aimed to improve traditional indexing techniques by learning the data distribution. The ZM-index [187] combines the standard Z-order space-filling curve [133] with the RMI from [98] by mapping multi-dimensional values into a single-dimensional space, which is then learnable using models. The ML-index [42] combines the ideas of iDistance [83] and the RMI to support range and KNN queries. Unlike Tsunami, these works only learn from the data distribution, not from the query workload.

Data Correlations. There is a body of work on discovering and taking advantage of column correlations. BHUNT [21], CORDS [79], and Pyro [104] automatically discover algebraic constraints, soft functional dependencies, and approximate dependencies between columns, respectively. CORADD [89] recommends materialized views and indexes based on correlations. Correlation Map [88] aims to reduce the size of B+Tree secondary indexes by creating a mapping between correlated dimensions. Hermit [193] is a learned secondary index that achieves low space usage by capturing monotonic correlations and outliers between dimensions. Although the functional mappings in the Augmented Grid are conceptually similar to Correlation Map and Hermit, our work is more focused on how to incorporate correlation-aware techniques into a multi-dimensional index.

Query Skew. The existence of query skew has been extensively reported in settings where data is accessed via single-dimensional keys (i.e., "hot keys") [31, 12, 199]. In particular, key-value store workloads at Facebook display strong key-space locality: hot keys are closely located in the key space [199]. Instead of relying on caches to reduce query time for frequently accessed keys, Tsunami automatically partitions data space using the Grid Tree to account for query skew.

3.7 Future Work

Complex Correlations. Augmented Grid's functional mappings are not robust to outliers: one outlier can significantly increase the error bound of the mapping. We can address this by placing outliers in a separate buffer, similar to Hermit [193]. Furthermore, Augmented Grid might not efficiently capture more complex correlation patterns, such as temporal/periodic patterns and correlations due to functional dependencies over more than two dimensions. To handle these correlations, we intend to introduce new correlation-aware partitioning strategies to the Augmented Grid.

Data and Workload Shift. Tsunami can quickly adapt to workload changes but does not currently have a way to detect when the workload characteristics have changed sufficiently to merit re-optimization. To do this, Tsunami could detect when an existing query type (Chapter 3.3.3) disappears, a new query type appears, or when the relative frequencies of query types change. Tsunami could also detect when the query skew of a particular Grid Tree region has deviated from its skew after the initial optimization. Additionally, Tsunami is completely re-optimized for each new workload. However, Tsunami could be incrementally adjusted, e.g. by only re-optimizing the Augmented Grids whose regions saw the most significant workload shift.

Tsunami currently only supports read-only workloads. To support dynamic data, each leaf node in the Grid Tree could maintain a sibling node that acts as a delta index [171] in which inserts, updates, and deletes are buffered and periodically merged into the main node.

Persistence. Tsunami's techniques for reducing query skew and handling correlations are not restricted to in-memory scenarios and could be incorporated into an index for data resident on disk or SSD, perhaps by combining ideas from qd-tree [194] or LISA [108].

3.8 Conclusion

Recent work has introduced the idea of learned multi-dimensional indexes, which outperform traditional multi-dimensional indexes by co-optimizing the index layout and data storage for a particular dataset and query workload. We design Tsunami, a new in-memory learned multi-dimensional index that pushes the boundaries of performance by automatically adapting to data correlations and query skew. Tsunami introduces two modular data structures—Grid Tree and Augmented Grid—that allow it to outperform existing learned multi-dimensional indexes by up to 6× in query throughput and 8× in space. Our results take us one step closer towards a robust learned multi-dimensional index that can serve as a building block in larger in-memory database systems.

Chapter 4

MTO: An Instance-Optimized Storage Layout for Cloud Data

The instance-optimized data storage layout that we introduced in the previous chapter, Tsunami, is designed for single-table in-memory data. In this chapter, we introduce another instance-optimized storage layout technique, MTO (Multi-Table Optimizer), which is designed to simultaneously optimize the data storage layouts for *all* tables in datasets that are stored on disk or on the cloud.

To efficiently process increasingly larger volumes of data, modern cloud-based data analytics services persist data in remote cloud storage, such as Amazon S3, and access data by "compute nodes" during query processing. These systems group data records into large blocks, each with hundreds of thousands or millions of records in order to maximize compression ratios. To maximize throughput and minimize I/O operations per second, during query processing, a block (or a subset of columns from a block) is the smallest unit of I/O from cloud storage.

To avoid accessing blocks that are not relevant to a query, per-block metadata, which is often cached in memory, is used to skip blocks during query processing. The most common form of per-block metadata is zone maps [29, 130, 149, 39], which store the minimum and maximum value for each column in a data block. For example (Fig. 4-1), if a block's zone map shows that the records in the block span dates from March to April 2020, and the query filters for records with dates in January 2020, then this particular block does not have to be read from storage (i.e., can be skipped) during this query's execution.

Zone maps are cheap to maintain and potentially useful, but their effectiveness at block skipping is highly dependent on how records are assigned to blocks (i.e., the data layout). By default, most systems usually sort each table by a certain sort column (e.g., the date column), and will place contiguous chunks of records into the same block. Under this basic blocking scheme, queries that filter over the sort column will be able to skip blocks based on zone maps, but filters over other columns do not provide much skipping opportunity (Fig. 4-1). Z-order [134] is a multi-dimensional sorting technique, often deployed in practice for its simplicity. However, for Z-order to be effective for block skipping, the columns on which to define the Z-order and their relative order must be manually and carefully selected, and poor tuning can actually

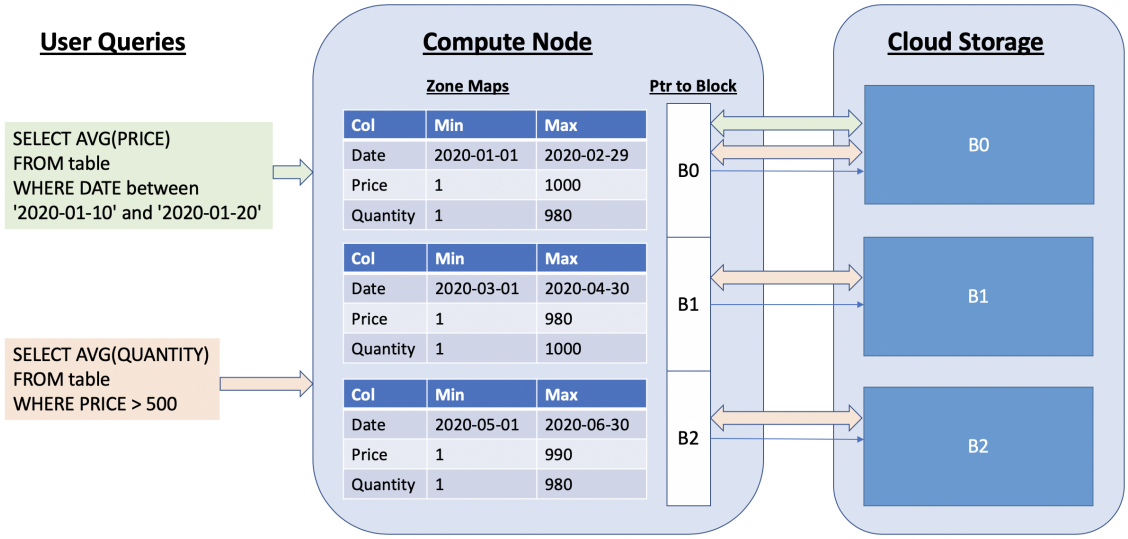

Figure 4-1: Zone maps over three data blocks. Using zone maps, the first query is able to skip blocks 1 and 2, whereas the second query cannot skip any blocks.

result in degraded performance.

To overcome the shortcomings of existing data layout techniques, instance-optimized data layouts for disk-based and cloud-based data "learn" a specialized blocking scheme (i.e., a sort order) that achieves high block skipping performance for a specific dataset and workload [195, 109]. Experiments on synthetic and real-world datasets and workloads show that instance-optimized data layouts can be orders of magnitude better in block skipping compared to simple sort-based data layouts as well as more advanced, fine-grained data skipping techniques [176, 177]. However, existing instance-optimized layouts can only optimize a single table's layout, for a query workload that only queries that table. In practice, analytics workloads typically contain many tables and the queries use diverse join patterns, such as in a star or snowflake schema.

One naïve approach to optimizing the layout for a multi-table dataset is to independently optimize each table's layout using an existing instance-optimized approach. However, as we show later, this approach does not perform significantly better, as it does not exploit knowledge about the joins. In this chapter, we propose MTO (Multi-Table Optimizer), the first instance-optimized data layout framework for optimizing whole datasets. Our key idea is to pass additional information about joins, which we refer to as *sideways information passing (SIP)*, through *join-induced predicates*, to jointly optimize the layout for all tables, simultaneously. This idea is inspired by prior work on SIP [85]; we discuss similarities and differences in Section 4.2.1. Furthermore, existing instance-optimized layout techniques [195, 138, 49] must re-optimize the entire layout in response to changes in the query workload. In contrast, MTO gracefully responds to workload changes through partial layout reorganization. We summarize our contributions as follows:

1. We propose MTO, the first instance-optimized data layout framework for multi-table datasets. MTO aims to minimize the overall number of blocks accessed in

an analytics workload with join queries, which are common in practice.

2. We introduce join-induced predicates, used in MTO to pass information through joins. We present algorithms that exploit join-induced predicates to "learn" better data layouts.

3. We introduce further practical techniques to ensure that MTO scales to larger datasets and query workloads and adapts to workload shift and data changes.

4. We evaluate MTO, both in simulations and by integrating with a commercial cloud-based data analytics service to measure end-to-end gains. We compare MTO against existing instance-optimized layouts and user-tuned blocking schemes, and show that MTO achieves up to 93% reduction in blocks accessed and 75% reduction in end-to-end query times compared to state-of-the-art blocking strategies.

In the rest of this chapter, we provide background (Section 4.1), introduce MTO's high-level design (Section 4.2), examine the details of MTO's algorithms (Sections 4.3 and 4.4), present experimental results (Section 4.5), review related work (Section 4.7), and conclude (Section 4.8).

4.1 Current Blocking Approaches

As shown in Fig. 4-1, zone maps are useful for skipping irrelevant blocks during query execution, but their effectiveness depends on the physical layout of the data among blocks (i.e., the sort order). We now describe existing approaches for data layout.

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

Z-ordering. One drawback to the sort key approach is that only queries that filter over the sort key column can benefit from block skipping. Z-ordering [134] "sorts" data over multiple columns simultaneously, and it is supported by several commercial systems [198, 40]. However, Z-ordering must be tuned carefully to achieve high performance. For example, a DBA must decide which columns to include in the Z-order, and whether to give more weight to certain columns over others. A poorly tuned Z-ordering can degrade block skipping performance compared to the sort key approach. Even when properly tuned, Z-ordering underperforms instance-optimized approaches.

Instance-optimized Layouts Instance-optimized layouts are specialized to perform well (e.g., achieve low overall query runtime) on a particular dataset and workload [195, 138, 109, 49]. By purposefully overfitting the layout for a specific dataset and workload, instance-optimized layouts are able to outperform existing approaches on that specific instance (dataset and workload).

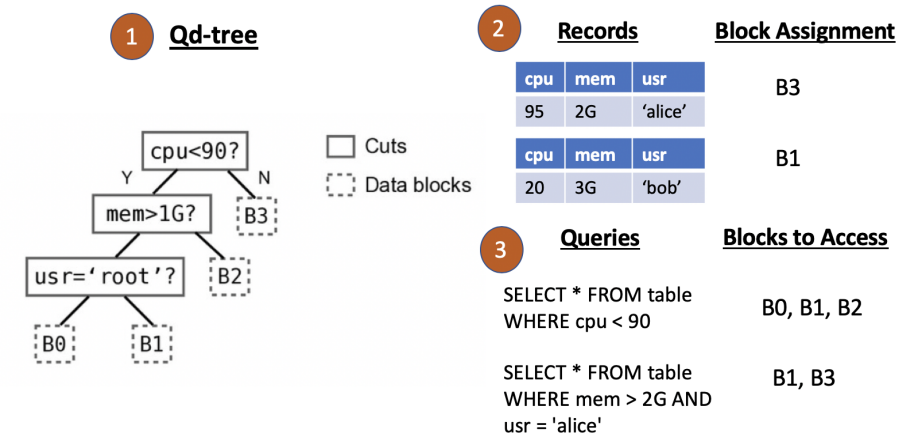

Figure 4-2: (1) Qd-tree defines blocks using cuts. (2) Qd-tree is used *offline* to route records to the blocks they are stored in and (3) is used *online* to determine which blocks need to be accessed during query execution.

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

4.1.1 Qd-tree

We now describe qd-tree [195], an existing instance-optimized data layout framework for single tables, which we use as a fundamental building block in our work. The intuition behind qd-tree is to tailor the block assignment strategy for a given query workload to reduce the number of blocks accessed when running that workload. For example, consider a workload consisting of a single query:

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

Let us divide the records of `table` into those that satisfy `X > 10` and those that do not. If we assign each set of records to a separate group of blocks (e.g., records that satisfy the predicate are assigned to blocks 1 and 3, while records that do not go in blocks 2 and 4), then during query processing, we only need to access the blocks corresponding to the "satisfying" set. We can apply similar logic to divide and block the records based on whether they satisfy the other predicate, `Y IN (1, 2, 3)`. However, if we sort/block the records based on their value in some unrelated column (e.g., `Z`), the query will likely need to access all blocks. This example illustrates that by blocking based on the specific filter predicates that appear in the query workload, we can reduce the number of blocks accessed during query processing.

We now provide a high-level overview of qd-tree, which we describe in more detail in the following subsections. The qd-tree workflow is as follows (Fig. 4-2):

1. The input to the workflow is a table and workload of queries that the user expects to run on the table. Using (a sample of) the table and query workload, construct a decision tree (which we call a qd-tree) that roughly evenly splits the records of the (sampled) table into data blocks (Section 4.1.1).

2. Offline, use the qd-tree to assign the table's records to blocks. This process is called *routing* a record (Section 4.1.1).

3. At query execution time, use the same qd-tree to determine which blocks the query needs to access (and therefore which data blocks can be skipped). This process is called *routing* a query (Section 4.1.1).

Qd-tree Structure

The qd-tree (Fig. 4-2) is a binary decision tree. Each node corresponds to some subset of records in the table. The root node corresponds to all records in the table. Each inner node contains a filter predicate, which we call a *cut*. The node's cut is used to divide its subset of records into two smaller subsets, one with records that satisfy the cut and the other with records that do not. The left child inherits the "yes" subset, and the right child inherits the "no" subset. The leaf nodes of the qd-tree correspond to data blocks. That is, the subset of records corresponding to a leaf node are assigned to the same data block.

Qd-tree Usage

We can use a qd-tree for both offline block assignment and online query processing. Given a qd-tree and a table, we route each record in the table through the qd-tree to assign it to the data block that it will be stored in. For example, consider the first record in Fig. 4-2. We route this record R through the qd-tree, from root to leaf. The root node's cut indicates that records that satisfy `cpu < 90` are inherited by the left child, while records that do not satisfy `cpu < 90` are inherited by the right child. Since R does not satisfy the root node's cut, we route R to the right child. The right child is a leaf node, and therefore we assign R to block 3. In the same manner, every record is assigned to a block.

At query execution time, we use the qd-tree to determine which blocks need to be accessed. For example, consider the first query in Fig. 4-2. We route this query through the qd-tree, from root to leaf. At the root node, the query only filters for records that could appear in the left child (i.e., any records that do not satisfy the cut `cpu < 90` are irrelevant), so we route the query to the left child. At this second node, the query could filter for records that appear in either child (i.e., records that satisfy and do not satisfy `mem > 1G` could both be relevant), so we route the query to both children. We continue to recurse in this manner, and at the end, we find that the query must access blocks 0, 1, and 2. Note that while records are always routed to exactly one block, queries can be routed to multiple blocks.

Qd-tree Construction Algorithm

We take the same greedy approach to construction as [195]: given a table and query workload, begin with all the records in a single block, i.e., the qd-tree has a single root node that contains all the records. In each iteration, we split a leaf node into two child nodes by applying a cut. Cuts are chosen from the set of *candidate cuts*, which is the set of filter predicates that appear in the query workload. For example, in the single-query workload described at the beginning of this section, there are two candidate cuts: `X > 10` and `Y IN (1, 2, 3)`. When choosing the cut for a node, we use the one amongst the candidate cuts that maximizes the number of records skipped by the resulting qd-tree over the given workload. We continue iterating until all leaf nodes have reached some desired size, measured by the number of records falling in the data block represented by that leaf node.

4.2 MTO Overview

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

We first introduce the key idea that differentiates MTO from existing instance-optimized approaches: sideways information passing using join-induced predicates. We then describe MTO's end-to-end workflow. We present more details in Sections 4.3 and 4.4.

4.2.1 Sideways Information Passing

Existing single-table layout approaches are sub-optimal in the multi-table case because they do not take advantage of sideways information passing between tables. To provide intuition, we use the following running example (Fig. 4-3): let the block size be 1M records. Let our dataset have two tables: Table A with 1M records, and Table B with 8M records. All of Table A's records will fall in the same block, so the sort order for Table A does not impact block skipping. We are only interested in the blocking strategy for Table B. Consider a workload consisting of queries similar to the following:

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

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

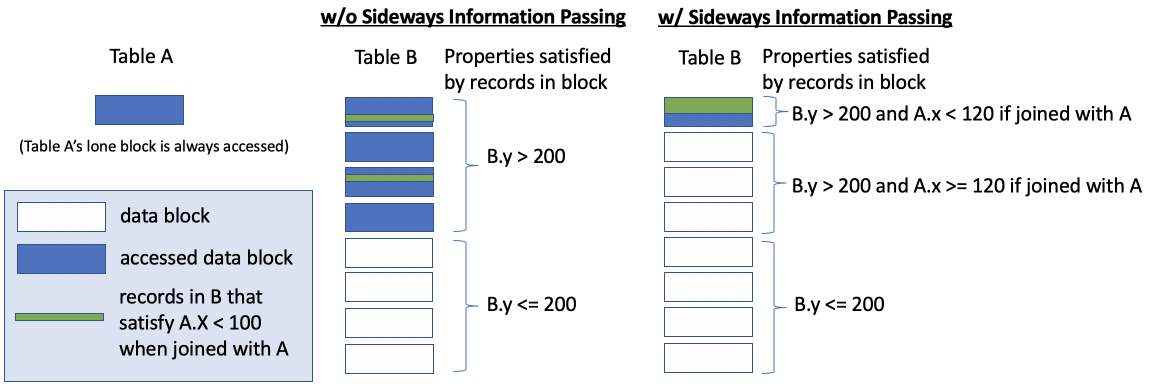

Figure 4-3: By taking advantage of sideways information passing to optimize the data layout, we can increase block skipping. Only blocks in the shaded regions are read.

By pushing down the two filter predicates, Table A's zone maps will be able to skip blocks based on the predicate `A.X < 100`, and Table B's zone maps will be able to skip blocks based on `B.Y > 200`. A DBA determining each table's layout independently would sort Table B's data by Y. Then the execution engine will use zone maps to skip over B's blocks where $Y \leq 200$, as shown in Fig. 4-3.

However, we can achieve even more block skipping using sideways information passing. We know that any records of the joined relation that have $A.X \geq 100$ are irrelevant. Therefore, the records in B that produce irrelevant records when joined with A are themselves irrelevant. By grouping together the records of B based on whether the join with A would produce relevant records, we can further increase block skipping. Essentially, Table B's zone maps are skipping blocks based on two predicates: `B.Y > 200`, as well as a new *join-induced predicate*: `B.KEY IN (SELECT A.KEY FROM A WHERE A.X < 100)`. Table 4.1 defines relevant terminology through an example. We explain join-induced predicates in more detail in Section 4.3.1.

Relation to Existing Approaches

The idea of join-induced predicates is similar to some existing techniques. Semi-join reduction [18, 62] uses sideways information passing at execution time to filter rows (e.g., construct a bitmap over the build input of a hash join and use it to filter rows on the probe input before they reach the join). In contrast, join-induced predicates in MTO are used in an offline optimization stage to determine the data layout and introduce negligible overhead during query execution.

Similar to semi-join reduction, data-induced predicates (diPs) [85] pass information about the blocks selected by a predicate (e.g., the zone maps over selected blocks), through joins, which induces a predicate on the joined table's join column that can be used to skip blocks on the joined table. diPs are applied during query optimization, which avoids the overhead of performing sideways information passing at execution time and gives the optimizer extra information with which to find better plans. However, diPs are only beneficial to block skipping when certain conditions about the data layout are met, most importantly that the join column values in the blocks that satisfy

Table 4.1: Join-induced predicate terminology example.

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

a predicate contain only a small portion of all possible join column values, otherwise the induced predicate will not be selective enough to skip blocks when applied to the joining table. In contrast, MTO explicitly constructs the block layout to maximize opportunities for skipping blocks during execution. We show in our evaluation that this difference allows MTO to outperform diPs.

4.2.2 MTO Workflow

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

Offline optimization

The MTO optimization algorithm takes a multi-table dataset and a query workload as input and creates one qd-tree per table, which will determine the data layout for that table's records. The algorithm has the following steps (Fig. 4-5):

1. Do the following for each query (Fig. 4-5 shows the workflow for one particular query):

   (a) Extract all simple predicates (Table 4.1) from the query, and group them based on which table they filter. In Fig. 4-5, the example query contains two simple predicates: `A.x < 100` and `B.y > 200`. Therefore, `A.x < 100` is extracted for Table A and `B.y > 200` is extracted for Table B.

   (b) Pass the simple predicates extracted in Step 1a through joins to create *join-induced predicates*. In the example, the simple predicate `B.y > 200` is

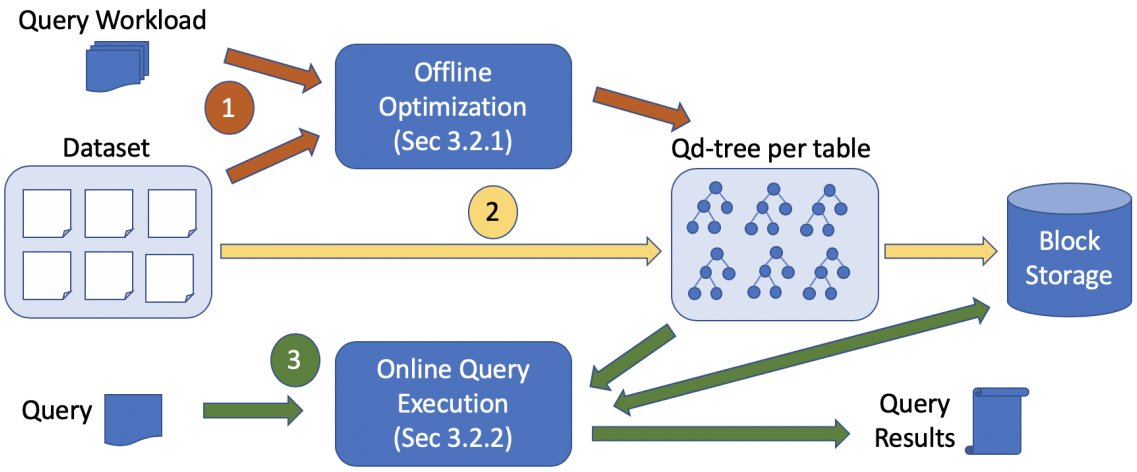

Figure 4-4: (1) In offline optimization, MTO produces a layout (a qd-tree per table) given a dataset and query workload. (2) MTO assigns records to blocks and stores them. (3) In online query execution, MTO skips blocks based on the layout.

passed though the join to Table A, which produces the join-induced predicate `A.key IN (SELECT B.key FROM B WHERE B.y > 200)`, which filters Table A. Similarly, the simple predicate `A.x < 100` passes through the join to Table B to create a join-induced predicate on Table B. For queries with more complex join graphs (e.g., Table A joins with Table B, which joins with Table C), a simple predicate can be passed through multiple joins (e.g., a simple predicate on Table A is passed through Table B to Table C and produces a join-induced predicate on Table C).

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

2. For each table independently: feed the table and overall query workload into the qd-tree construction algorithm (Section 4.1.1). The predicates over that table extracted in Step 1 (which could be either simple predicates or join-induced predicates) become the *candidate cuts* for the qd-tree. The constructed qd-tree determines that table's data layout.

Given the optimized layout, MTO assigns each table's records to data blocks using their respective qd-trees, as described in Section 4.1.1.

Online query execution

At query execution time, MTO uses the qd-tree for each table to determine which blocks need to be accessed. Following our running example, Fig. 4-6 shows the qd-tree that might be constructed for Table B by the MTO optimization algorithm. The qd-tree has three leaf nodes, so the records of Table B will be stored in three blocks

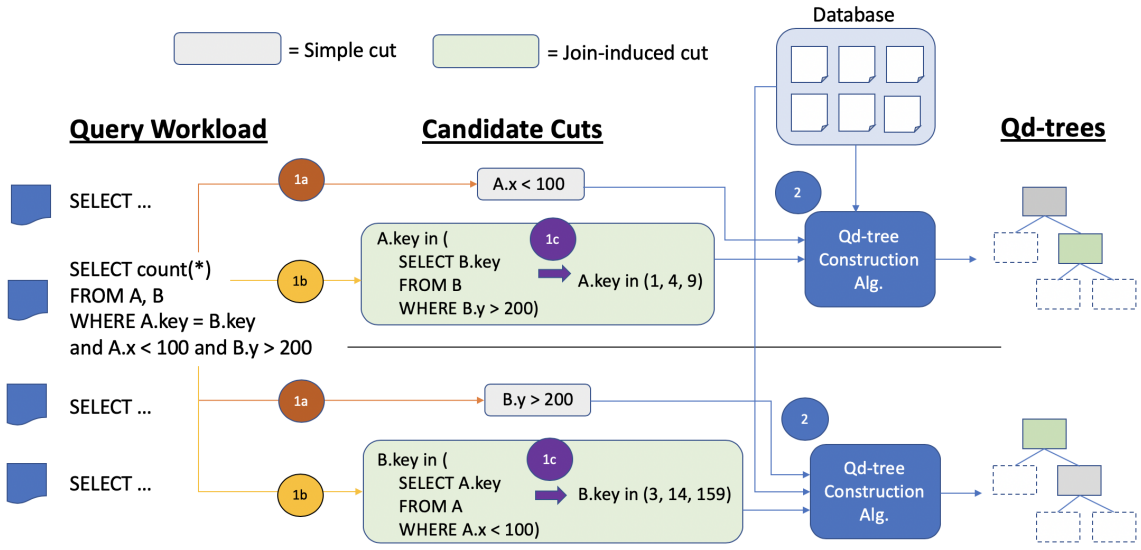

Figure 4-5: MTO optimization uses the query workload and dataset to create one qd-tree per table.

(B0, B1, B2). The qd-tree uses two cuts: a simple cut `B.y > 200`, and a join-induced cut shaded in green. To determine which blocks of Table B need to be accessed when processing the user query in Fig. 4-6, MTO will do the following (a similar process would occur independently to determine blocks to access on Table A):

1. Identify all predicates from the query on that table, including join-induced predicates, following the same procedure as Steps 1a and 1b (but not 1c) from Section 4.2.2.

2. Use the predicates to route through the table's qd-tree to identify which blocks need to be accessed, using the process described in Section 4.1.1. Section 4.3.1 provides details about routing through join-induced cuts (e.g., Step 2b in the example).

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

4.3 MTO Algorithms

In this section, we provide details about join-induced predicates and also describe how MTO maintains low optimization times even when scaling to larger datasets. We use the terminology shown in Table 4.1.

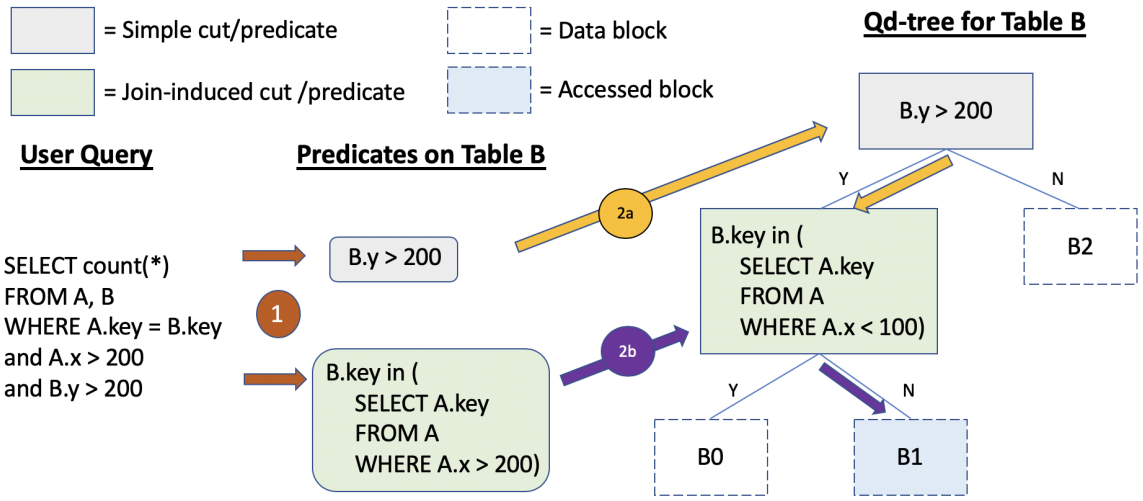

Figure 4-6: At query time, MTO uses the per-table qd-trees to determine which blocks to access from each table. This query only needs to read block 1 from Table B.

4.3.1 Join-induced Predicates

When Can We Induce?

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

In the simple example in Section 4.2.2, we induced a predicate from Table A to Table B, and vice versa. However, in some cases we cannot induce a predicate from one table to another while maintaining a semantically equivalent query. Consider the following query:

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

We can induce from A to B, producing the join-induced predicate `B.KEY IN (SELECT A.KEY FROM A WHERE A.X < 100)`. However, we cannot induce from B to A. To determine when predicates can be induced while maintaining a semantically equivalent query, we use the following set of rules, similar to those found in [85]:

- Predicates can be induced in both directions through inner joins; from the left to right side for a left outer join, and vice versa for right outer joins; in both directions through semi joins; and from the left to right side for a left anti-semi join, and vice versa for right anti-semi joins. Predicates cannot be induced through full outer joins.

- For self-joins, MTO logically creates two copies of the table, treats them as different tables, and applies the above rules.

- Predicates can be induced from an outer query into a correlated subquery [189] through any of the above rules. In the example query given above, a predicate from the outer query (`A.X < 100`) is induced into the correlated subquery (`SELECT COUNT(*) ...`) through an inner join (`A.KEY = B.KEY`).

Just because we *can* induce doesn't mean we *should*. In the optimization process, MTO only considers join-induced predicates whose induction paths are composed only of joins originating from columns with unique values (e.g., inducing from a dimension table into a fact table by joining on the dimension table's primary key, but not from a fact table's foreign key into a dimension table). This is not a fundamental limitation of join induction; instead, we enforce this restriction to make inserts and deletes more efficient (Section 4.4.2). We verified experimentally that this restriction has minimal impact on performance. Intuitively, this is because predicates induced from join columns with non-unique values tend to fall on smaller tables with fewer blocks (e.g., dimension tables), which limits the predicate's impact on the number of blocks skipped dataset-wide.

How Do We Use Them?

Like simple cuts in the qd-tree, join-induced cuts are used to route records and queries down the tree. To route records, we use the *literal* join-induced cut in the same way as a simple cut. To route queries, the qd-tree checks for subsumption between the query and the *logical* join-induced cut: if the query's join graph does not share the join-induced cut's induction path, route the query to both child nodes in the qd-tree. Otherwise, route to the left child if the query's filters on the source table intersect the source cut, and independently route to the right child if the query's filters on the source table intersect the *negation* of the source cut. For example, in Step 2b of Fig. 4-6, the query filter on the source table (`A.x > 200`) does *not* intersect the source cut (`A.x < 100`), but *does* intersect the negation of the source cut ($\texttt{A.x} \geq \texttt{100}$), so we only route to the right child.

Qd-tree nodes that use join-induced cuts must store both the logical and literal cuts. The logical cut (i.e., a query of nested semi-joins) is compact, but literal cuts can incur high memory costs, because the `IN` list can grow very large, especially over high-cardinality key columns. To reduce space usage, we compress `IN` lists as Roaring Bitmaps [22], which is the state-of-the-art bitmap compression technique [188].

4.3.2 Scalability through Sampling

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

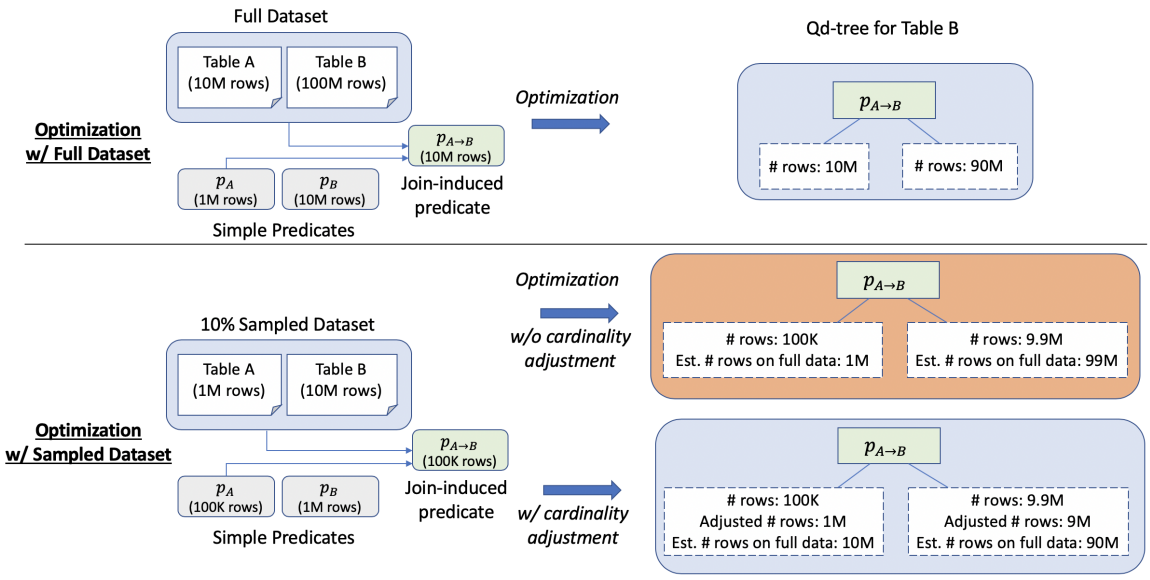

Figure 4-7: Cardinality adjustment allows MTO to achieve accurate block size estimates when optimizing based on a dataset sample, which improves the quality of the resulting layout.

size on the full dataset is b, MTO uses the adjusted block size $b \times s$ when optimizing based on the sampled dataset.[2]

It is well-known that the join of two uniform samples has quadratically fewer tuples than a sample of the original join [75]. MTO must account for this effect when evaluating the quality of join-induced cuts when optimizing on a sampled dataset. For example, Fig. 4-7 shows a dataset with 10M records in Table A and 100M records in Table B. The simple predicate p_B on Table B and the join-induced predicate $p_{A\to B}$ on Table B both select 10M records. However, on a sampled dataset with $s = 0.1$, p_B selects (10M)s=1M records, whereas $p_{A\to B}$ selects (10M)s^2=100K records. If MTO estimates block sizes on the full dataset as $1/s$ of the block sizes on the sample, then it produces inaccurate estimates of block size (e.g., the qd-tree shaded in orange in Fig. 4-7). Optimizing without taking this discrepancy into account may degrade the quality of the resulting layout.

To account for this effect, MTO attaches a value called the *cardinality adjustment (CA)* to every join-induced cut, defined as s^d, where d is the induction depth (e.g., the CA for $p_{X\to Y\to Z}$ is s^2). Therefore, in Fig. 4-7 the CA for $p_{A\to B}$ is s. The left block in the bottom qd-tree (which is constructed over a sample) has block size 100K records, but the cardinality-adjusted block size is $(100\text{K})/s = 1\text{M}$ records. This is then used to produce an accurate estimate of the block size on the full dataset.

Formally, let qd-tree node N cover r records of the sampled table, so that the estimated cardinality of N on the full dataset is r/s. Let N use join-induced cut p, so

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

that the left child N_L covers r_L records and the right child N_R covers $r_R = r - r_L$ records. If p has a CA of k, then the estimated cardinality of N_L on the full dataset is not r_L/s. Instead, it is r_L/sk. Accordingly, the estimated cardinality of N_R on the full dataset is not r_R/s, but instead $r/s - r_L/sk$. Simple cuts have a CA of 1.

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

4.4 Workload Shift and Data Changes

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

4.4.1 Dynamic Workloads

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

Minimizing Impact of Reorganization

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

Reward Function

Assume that workload shift has occurred and we have already observed some queries, denoted Q, from this new workload (e.g., a sample of recently-run queries); assume we expect to run q more queries from the same distribution as Q before the next workload shift. The reward of reorganizing a subtree T (i.e., replacing T with a new qd-tree T' over the records in T's blocks) is defined as $R(T, Q) = (q/w) \cdot B(T, Q) - C(T)$, where:

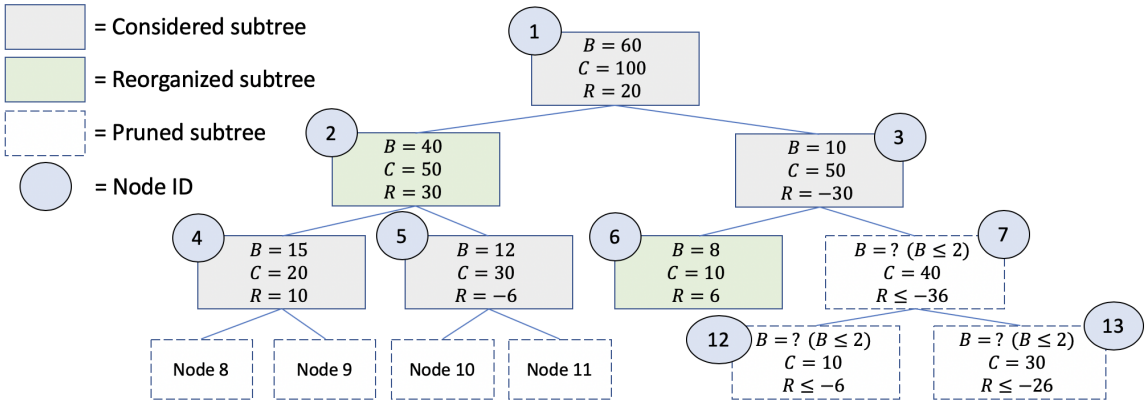

Figure 4-8: Using $q/w = 2$ in the reward, MTO chooses to reorganize the subtrees of nodes 2 and 6, achieving total reward 36.

- $B(T, Q)$ is the average number of block accesses that can be reduced for a query in Q if we fully reorganize the subtree T. It represents the *benefit* of reorganizing T in terms of expected impact on each of the next q queries. To compute $B(T, Q)$, we take all records in T's blocks and re-run offline optimization (Section 4.2.2) to construct a new qd-tree T', then take the difference in block accesses over Q between T and T'.

- $C(T)$ is the total number of blocks in T. It represents the *cost* of fully reorganizing T's blocks. Note that $C(T) \geq B(T, Q)$.

- w represents the *relative* overhead of reorganizing (i.e., re-compressing and re-writing blocks) vs. accessing blocks in the underlying storage system. For example, in our evaluation system (Section 4.5.1), compressing and writing a block is $\sim$100$\times$ slower on average than reading a block, so $w = 100$.

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

Finding the Optimal Reorganization Strategy

For each table in the dataset, we compute $R(T, Q)$ for each subtree T of the table's qd-tree. We want to find the set of non-overlapping subtrees that has the maximum combined reward; this *optimal set* can be empty, in which case overall reward is 0. We find the qd-tree's optimal set via dynamic programming: we visit all nodes, starting from the leaves and working towards the root. At each node, we determine the optimal

set over its subtree: the optimal set for a leaf L is $\{L\}$ if $R(L, Q) > 0$ and empty otherwise. The optimal set for a non-leaf T is either $\{T\}$ or the union of the optimal sets of its two children, whichever one has higher combined reward. The root node's optimal set is the qd-tree's optimal set. Fig. 4-8 shows a qd-tree in which the optimal set has two subtrees.

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

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

1. $B(T, Q)$ is upper bounded by the number of block accesses for the average query in Q using T's layout. This is because a new qd-tree T' cannot reduce the block accesses to less than zero.

2. $B(T, Q) \geq B(T_L, Q) + B(T_R, Q)$, where T_L and T_R are the left and right subtrees of T. This is because any reorganizations of T_L and T_R independently can also be achieved by reorganizing T.

3. If $R(T, Q) \geq B(T_L, Q) + B(T_R, Q)$, then no set of T's subtrees can have combined reward larger than $R(T, Q)$. This follows from property 2 and the fact that $C(T) \geq 0$.

To take advantage of these properties, we first use property 1 on every subtree to prune out any subtrees whose maximum possible reward is non-positive. For each non-pruned subtree T, we cache the upper bound for $B(T, Q)$. We then compute the reward for subtrees starting from the root node and continuing in breadth-first order (e.g., in node ID order in Fig. 4-8). When it comes time to compute the reward for T, we first check the cached upper bound for $B(T, Q)$. If the bound is low enough that $R(T, Q)$ cannot be positive, then we prune T.

Otherwise, we compute the true value of $B(T, Q)$ and update the cache to help prune later subtrees: (1) Benefits for T's subtrees are upper bounded by $B(T, Q)$. Let T's sibling and parent be S and P. Benefits for S's subtrees are upper bounded by $B(P, Q) - B(T, Q)$. This is possible through property 2. In Fig. 4-8, this helps us prune nodes 7, 12, and 13. (2) Once we compute the reward for T and its two children, we use property 3 to possibly prune out all further subtrees of T. In Fig. 4-8, this helps us prune nodes 8-11.

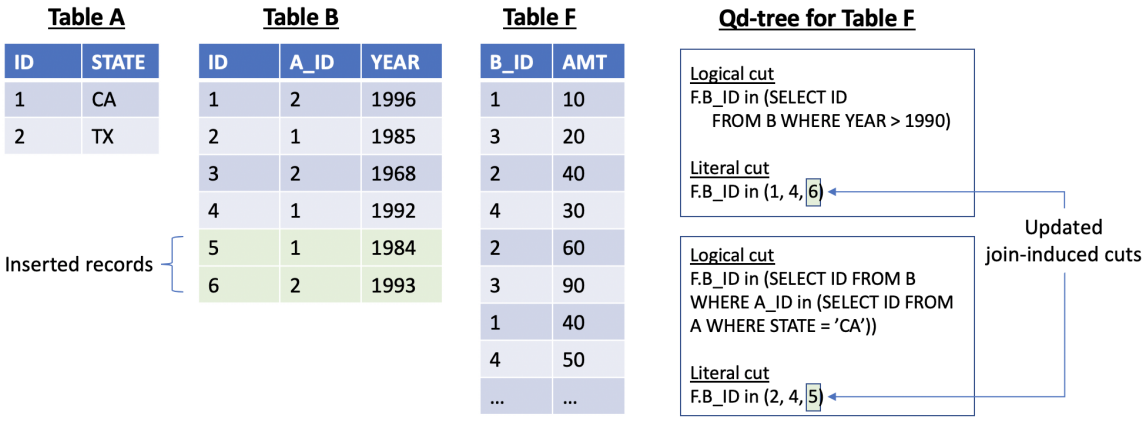

Figure 4-9: Inserting two records into table B causes updates to join-induced cuts in table F's qd-tree.

4.4.2 Dynamic Data

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

Data changes pose one unique challenge for MTO: join-induced cuts must be updated to reflect the new data. After an insert into a table, MTO must update all join-induced cuts in *other tables'* qd-trees that have the changed table on its induction path. In Fig. 4-9, inserting two records into table B results in updates to two join-induced cuts in table F's qd-tree. Any join-induced cuts in table B's qd-tree are unaffected. We perform the update by evaluating the relevant cut only on the inserted records, not all the records of table B. Similarly, a delete results in updates to join-induced cuts in other tables' qd-trees, performed by evaluating cuts only on the deleted records. A data update is handled as a delete followed by an insert.

A subtle but important ramification of updating join-induced cuts is that it shifts the "boundaries" between blocks. Will this force existing records to change blocks? Assuming referential integrity [190], and due to induced predicates only originating from join columns with unique values, like primary key columns (Section 4.3.1), inserts and deletes in MTO will never cause unchanged records to change blocks, because there cannot be records in the "boundary shift" region. For example, the inserted records in Fig. 4-9 do not join with any existing records in table F, so table F's join-induced cuts will select the same set of records before and after updating. However, data updates might cause updates to join-induced cuts that force existing records to change blocks.

4.5 Evaluation

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

- On a commercial cloud-based analytics service, MTO achieves up to 93% reduction in blocks accessed and up to 75% reduction in overall query time compared to alternative methods (Section 4.5.2). Queries with selective filters over joined tables benefit most from MTO (Section 4.5.3).
- MTO achieves low optimization times through sampling, resulting in faster end-to-end performance compared to alternatives (Section 4.5.4).
- MTO adapts its layout in response to workload shift and data changes (Section 4.5.5) and scales to larger query workload sizes and data sizes (Section 4.5.6).

4.5.1 Setup

Datasets and Workloads

We evaluate on three datasets: Star Schema Benchmark (SSB) [145], TPC-H [180], and TPC-DS [181], each by default with scale factor 100. This corresponds to around 60GB of data for SSB and 100GB of data for TPC-H and TPC-DS. For SSB, we use all 13 queries in the workload. For TPC-H, we support all 22 templates, and by default we use 8 randomly generated queries per template, resulting in a workload of 176 queries. For TPC-DS, we use 46 templates that vary in complexity[3], with one query per template.

Implementation and Systems

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

We performed a shallow integration of MTO into Cloud DW: each block across the multi-table layout is assigned a unique block ID (BID). For each table, we materialize a new column that contains the BID for each record. In storage, we sort each table by

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

its BID column. The per-block zone maps will now contain the min/max BIDs for records in the block. Before feeding each query into Cloud DW, MTO will rewrite the query by adding extra predicates which are used transparently by Cloud DW's zone maps to skip unnecessary blocks. For example, if routing a query through Table A's qd-tree tells us that processing the query only requires records from blocks 2 and 4 of Table A, we add the predicate `A.BID IN (2, 4)` to the query.

Comparisons

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

In simulation, we also use data-induced predicates [85] (diPs, described in Section 4.2.1) to enhance the performance of STO and Baseline, using range-sets of size 20. Since diPs are meant to be incorporated into the query optimizer, we were not able to show the performance of diPs in Cloud DW as part of our shallow integration.

Metrics

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

4.5.2 Overall Results

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

Simulated Block Skipping

Fig. 4-10a shows that across datasets, MTO achieves between 43%–96% reduction in simulated block accesses compared to Baseline and between 32%–94% reduction in simulated block accesses compared to the best alternative method.

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

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

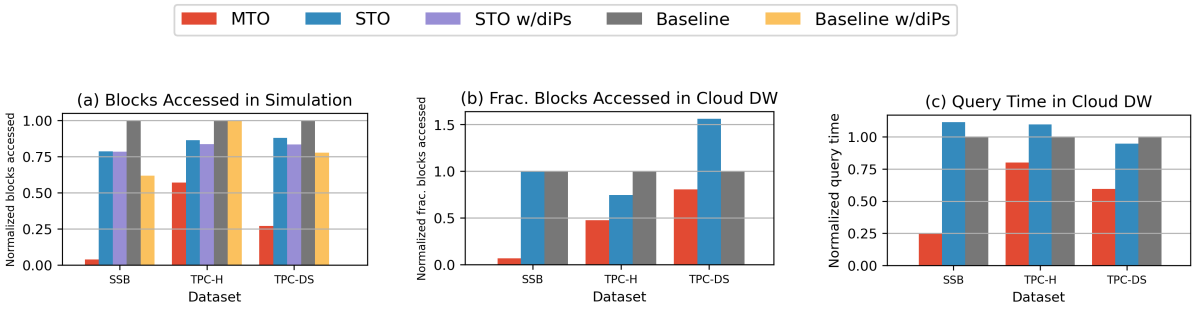

Figure 4-10: MTO achieves better overall workload performance than alternatives across datasets and metrics. Note that the y-axes are normalized to the metric achieved by Baseline.

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

Table 4.2: Statistics of MTO's qd-trees.

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

Table 4.2 shows that for MTO, the total number of cuts over all qd-trees varies across datasets. This trend is primarily due to the size of the query workload that MTO optimizes on: (1) workloads with more queries produce more candidate cuts, and (2) larger workloads require a finer-grained blocking strategy in order to maximize block skipping across all queries. For all datasets, MTO's qd-trees are composed mostly of join-induced cuts. On SSB, the induction depth of join-induced cuts is always 1, because all dimension tables are joined directly to the fact table. On TPC-H, the maximum induction depth is 4 (e.g., a join-induced cut with region as the source table and joining through nation, customer, and orders to reach the lineitem table).

Table 4.2 also shows that the memory overhead of MTO, which comes from its per-table qd-trees, is at most a few GB, which is small compared to the size of the data (∼100GB). MTO's size on TPC-H is much higher than on SSB and TPC-DS due to having more join-induced cuts with higher-cardinality literal cuts (e.g., many join-induced cuts on TPC-H originate from the orders table, which produces literal cuts with as many as 150M values).

Block Skipping on Cloud DW

Fig. 4-10b shows that MTO's simulated reduction in block accesses roughly translates to actual reduction of block accesses on Cloud DW. Across datasets, MTO achieves

between 19%–93% reduction in fraction of blocks accessed compared to the best alternative method.

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

End-to-end Runtimes on Cloud DW

Fig. 4-10c shows that across datasets, MTO achieves between 20%–75% reduction in end-to-end query runtimes compared to the best alternative method. The reduction in query time is generally not as dramatic as the reduction in block accesses because block access is only one part of the total time spent on query execution (e.g., time to compute joins is not reduced by block skipping). However, on TPC-DS, MTO and STO actually improve their normalized performance compared to Fig. 4-10b. This is because Baseline incurs heavier runtime costs of using semi-join reductions, as explained in Section 4.5.2.

4.5.3 Performance Breakdown by Query

Fig. 4-11 shows the fraction of queries that achieve a certain reduction in query time on Cloud DW compared to the alternative methods. Reduction in query times from MTO is achieved by all queries in SSB, around 50% of query templates in TPC-H, and around 75% of queries in TPC-DS. For a few queries, performance regresses when using MTO's layout. This is because MTO optimizes to achieve best overall block skipping across all queries in a workload. Therefore, MTO may allow performance to regress for certain queries in order for overall performance to improve.

When Does MTO Pay Off?

Fig. 4-10 shows that the performance benefits of MTO varies depending on the dataset. To better understand the conditions under which MTO offers performance benefit over STO and Baseline, we select five query templates with different filter/join characteristics from the TPC-H workload, all of which touch the fact table (lineitem): Q1 has no joins and scans most of the fact table, Q14 has a filter over the fact table on the sort column (`L_SHIPDATE`) but no filter over joined dimension tables, Q6 has

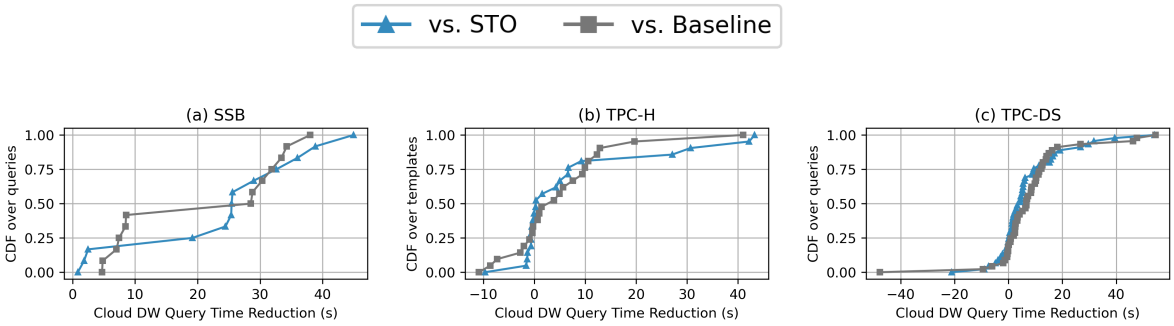

Figure 4-11: Reduction in query runtimes achieved by MTO, relative to STO and Baseline. Different queries achieve different performance gains.

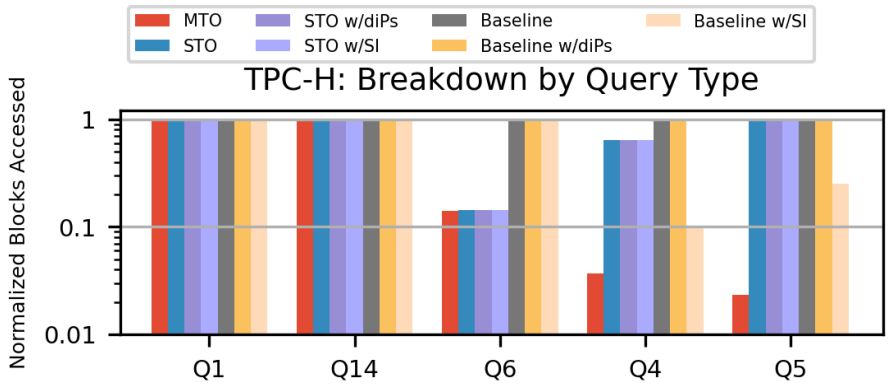

Figure 4-12: MTO has the most performance advantage over STO and Baseline on queries with selective filters over joined tables, like Q4 and Q5.

filters over the fact table on non-sort columns but no joins, Q4 has selective filters over joined dimension tables only on columns that are correlated to the fact table's sort column, and Q5 has selective filters over joined dimension tables on columns that are not correlated to the fact table's sort column. We create five different layouts using MTO and STO, specialized for each query template individually.

Fig. 4-12 shows simulated block skipping. For STO and Baseline, we also evaluate using data-induced predicates (diPs), as well as creating a secondary index (SI) on the fact table's join column (`L_ORDERKEY`) so that at runtime, we push a join-induced predicate from the dimension table to the fact table and use the secondary index to prune blocks.

Fig. 4-12 provides several insights: (1) On queries that have non-selective filters (like Q1) or selective filters only over the sort column (like Q14), MTO and STO have little or no advantage over Baseline because Baseline already prunes most irrelevant blocks. (2) On queries that have selective filters over non-sort columns and no joins or only non-selective filters over joined tables (like Q6), MTO and STO perform equally well, because MTO cannot take advantage of join-induced predicates, but they both outperform Baseline. (3) On queries with selective filters over joined tables that are

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

Table 4.3: Offline optimization times for Fig. 4-10.

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

Table 4.4: How many **X** until MTO runs more queries than **Y**?

correlated with the fact table's sort column (like Q4), MTO performs better than STO and Baseline, but using a secondary index allows Baseline to take advantage of correlations to filter out some blocks in the fact table at runtime. (4) On queries with selective filters over joined tables that are not correlated with the fact table's sort column (like Q5), MTO outperforms all alternatives by a large margin.

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

4.5.4 End-to-end Performance

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

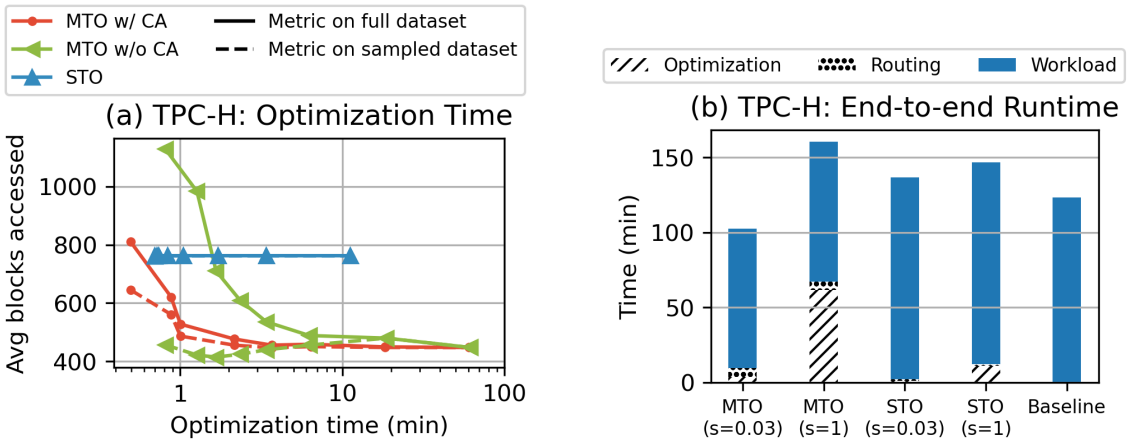

Figure 4-13: (a) MTO and STO can decrease optimization time by using sampling. Cardinality adjustment (CA) helps MTO mitigate performance degradation. (b) MTO achieves the lowest end-to-end runtime when optimized with a 3% data sample.

Optimization Time

Table 4.3 shows the time that MTO and STO take to find the optimal layout for each dataset, when optimized using the data sample rates shown in the table. These sample rates were chosen so that the simulated performance of the layout optimized on the sample has less than 1% difference with the layout optimized on the full data. Smaller query workloads and simpler join patterns generate fewer candidate cuts, which leads to lower optimization times. Therefore, MTO optimization finishes quickly for SSB, which has 13 queries and a maximum induction depth of 1, while optimization takes longer on TPC-H, whose workload has 176 queries and a maximum induction depth of 4 (Table 4.2). Similarly, data routing (i.e., assigning each record to a block) is slowest on TPC-H because the qd-trees optimized on TPC-H are larger and the paths from root to leaf are deeper (Table 4.2). Optimization and routing times are lower for STO than MTO because STO does not need to consider join-induced cuts during optimization and does not include join-induced cuts in its qd-trees, which makes both steps computationally simpler than for MTO.

Fig. 4-13a shows the impact of varying the sample rate between 1 (i.e., no sampling) and 0.0003 on optimization time for TPC-H. The solid lines show blocks accessed in simulation when evaluated on the full dataset, whereas the dotted lines show blocks accessed when evaluated on the sample. With cardinality adjustment (CA) (Section 4.3.2), the metric computed on the sample is close to the true metric on the full dataset, whereas without CA, the sampled metric is inaccurate. By using CA, MTO can reduce its optimization time from nearly an hour without sampling to under 4 minutes with a 3% sample, while achieving nearly the same layout quality. STO's layout quality is negligibly impacted by sampling because it does not consider join-induced cuts, which are most affected by sampling.

End-to-end Time

Fig. 4-13b shows the end-to-end time for the TPC-H workload with 176 queries, including the offline optimization and routing times. By optimizing on a 3% sample of the data, optimization time is a small fraction of overall runtime for both MTO and STO, and therefore the faster query times achieved by MTO allow it to complete the end-to-end workload quickest. Query routing latency (i.e., determining which blocks a query must read) is on the order of milliseconds per query, which is negligible compared to total query time, which is on the order of seconds per query (Fig. 4-11).

Since MTO pays the upfront time cost to perform offline optimization and data routing, how long does it take for MTO to catch up to STO and Baseline? Table 4.4 shows the total number of queries MTO runs before surpassing STO and Baseline, as well as the time it takes for MTO to complete that many queries (including time for offline steps). For example, on SSB, MTO runs more queries than STO after $\sim$2 minutes. In all cases, MTO reaches this crossover point before the workload completes.

4.5.5 Dynamic Workloads and Data

Dynamic Workloads

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

Fig. 4-14a shows that immediately after the workload shift, queries on MTO have higher execution times than queries on Baseline, because MTO's layout is not optimized for the observed workload (i.e., templates 12-22). Re-optimizing and physically reorganizing the entire layout based on the observed workload (MTO Full Reorg) takes more than two hours; during reorganization, queries are still executed on the old layout (Section 4.4.1). However, MTO is able to use partial re-optimization (Section 4.4.1, using $q = 200$ and $w = 100$) to physically reorganize only a subset of existing blocks. Partial re-optimization and reorganization completes in under an hour (MTO Partial Reorg), while achieving nearly the same resulting performance benefit as a full reorganization, because it only reorganizes the blocks that have the most impact on performance. The bottom half of Fig. 4-14a shows the impact of reorganization on the total number of queries executed over time. Even though MTO initially executes fewer queries than Baseline after the workload shift, it is able to quickly recoup the lost time after reorganization.

Table 4.5 shows that as we increase q in the reward function while fixing $w = 100$ (Section 4.4.1), MTO will choose to reorganize a larger fraction of the data. Therefore, a user can adjust q to trade off between decreased execution time of future queries and computation costs of reorganization. The time for physically performing reorganization (i.e., writing/compressing blocks) is roughly proportional to the fraction of data reorganized, and reorganizing all TPC-H data takes around 2 hours in our setup. Furthermore, the time to perform re-optimization is kept relatively low (compared to

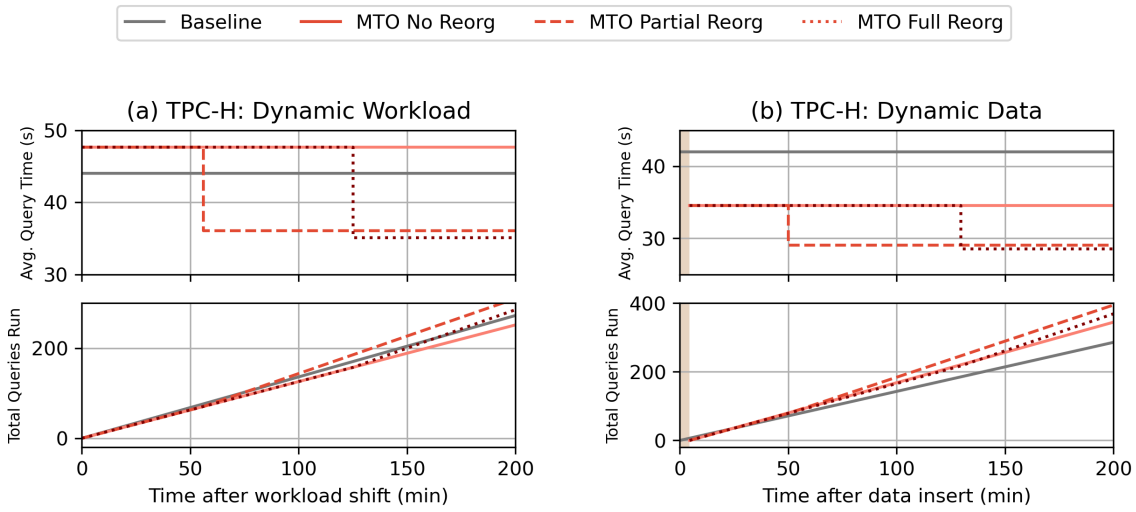

Figure 4-14: (a) MTO initially performs worse than Baseline after workload shift but performs better in the long run after reorganization. (b) MTO maintains its advantage over Baseline after data insertion.

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

Table 4.5: MTO behavior after workload shift.

time for performing reorganization) because we use the properties from Section 4.4.1 to avoid unnecessary computation, and therefore only consider a small fraction of all subtrees during the re-optimization process (Table 4.5).

Dynamic Data

To show how MTO adapts to dynamic data, we use the TPC-H dataset. We first remove all records from the orders table with orderdate after Jan 1, 1996, and all records in the lineitem table that join with the removed order records. This leaves around 61% of the records in both the orders and lineitem tables. We perform offline optimization using this partial dataset, then insert the records we had removed. This represents the common use case in which new records are inserted into the fact tables (in this case, new records from 3 years of orders).

Fig. 4-14b shows that MTO takes around 4 minutes to update join-induced cuts

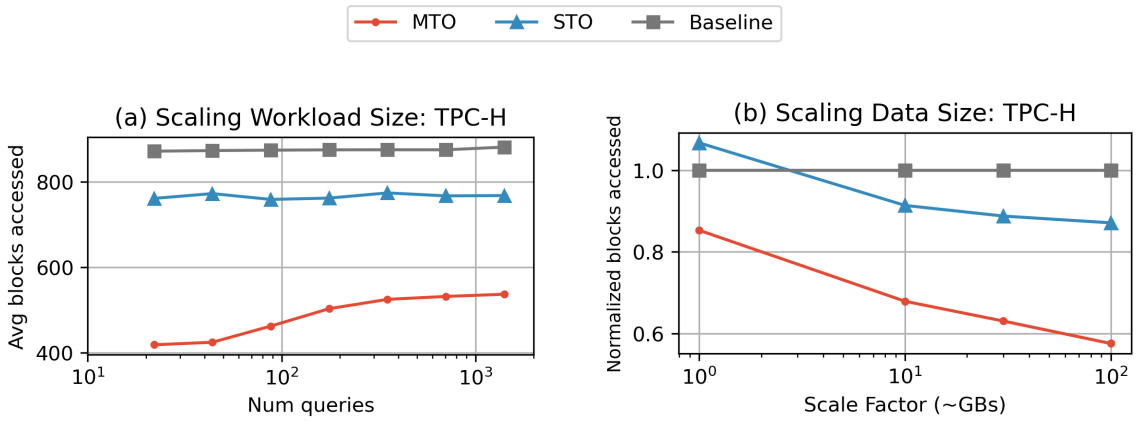

Figure 4-15: MTO scales to larger query workload sizes and improves its relative performance at larger data sizes.

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

4.5.6 Scalability

Workload Size

On TPC-H, we vary the number of queries per template in the workload from 1 to 64. Since there are 22 templates, this results in workloads ranging from 22 to 1408 queries. Fig. 4-15a shows that as workload size increases, average blocks accessed by MTO increases slightly, because the fixed number of data blocks does not provide MTO enough degrees of freedom to optimize for all queries in a larger workload with more unique predicates. Nevertheless, MTO maintains a relative performance advantage over alternative methods, with 30% and 39% fewer blocks accessed than STO and Baseline, respectively, for a workload with 1408 queries.

Data Size

On TPC-H, we vary the scale factor from 1 to 100, while maintaining the workload of 176 queries and a block size of 500K records. Fig. 4-15b shows that MTO (and STO) achieves greater reduction in block accesses over Baseline as data size increases;

larger data size leads to more total data blocks, which allows MTO (and STO) to take advantage of finer-grained blocking strategies.

4.6 Discussion & Future Work

Optimization Objective. MTO optimizes the layout for a particular workload with the objective of minimizing the overall number of blocks accessed across all queries in the workload. As shown in Section 4.5.3, this can lead to some queries in the workload requiring more block accesses than would have been required using the heuristic baseline layout. Users have the freedom to choose the optimization objective. For example, if users want to ensure that no queries will degrade in terms of blocks accessed compared to a baseline, MTO can set the qd-tree's objective function to have a high penalty when the number of blocks accessed is above the baseline.

Furthermore, MTO's optimization objective is currently defined solely by scan cost (i.e., number of blocks accessed), but could be augmented to also take join cost into account (e.g., how would a particular blocking assignment affect the choice of physical join operator for query execution).

Scalability. MTO uses data sampling to maintain low optimization times. However, there are other complementary techniques for achieving scalability. Instead of uniform sampling per table, we could use stratified sampling, in which we sample at lower rates from regions of the data space that are accessed by fewer queries. For example, MTO should not sample any records that are not accessed by any query, since the blocks that contain those records are never accessed, so their layout is irrelevant. This may allow MTO to produce a layout with the same performance while using a smaller overall sample, therefore decreasing optimization time.

MTO can also take advantage of query sampling. In the most basic form, MTO optimizes based on a uniform sample of the query workload. A more complex approach is to first cluster queries based on common tables/columns accessed (e.g., each template in TPC-H is a cluster), and sample queries stratified by cluster. Another approach is to cluster the candidate cuts for qd-tree based on the similarity of records filtered, and sample stratified by cluster. For example, the cuts `PRICE < 7.99` and `PRICE < 8.00` likely achieve similar performance, so we need to only consider one of them in the qd-tree.

Workload and Data Shift. MTO assumes a static workload. In response to workload shift, MTO's performance may degrade. The simplest approach to maintaining performance under workload shift is to re-optimize the layout. However, this approach is costly because it requires re-assigning each record to a potentially different block. If the workload shift only impacts one region of the data (e.g., queries over Europe have shifted), a less costly alternative approach is to only re-optimize the relevant subtree of the qd-tree (e.g., if the root node has cut `REGION = 'EUROPE'`, only re-optimize the left subtree).

One common form of workload shift is time-varying predicates (e.g., queries that always select the last day of data will have different date literals for every invocation). By predicting the change in time-varying predicates, MTO can optimize its layout based on not only on the past query workload, but also the expected future queries.

MTO also assumes a static dataset. In response to data updates, MTO may need to re-optimize its layout in order to maintain high performance, using similar approaches as those described above.

Instance-optimization. Aside from better block partitioning, there are other possible avenues for reducing data access through instance-optimized strategies. For example, one might create materialized views specialized for certain query patterns or denormalize tables that are frequently joined. These complementary techniques come with their own challenges, such as space overhead and maintainability. In our work, we focus on how to create a layout with only one copy of all the data, with all tables in their original schemas.

4.7 Related Work

Physical Data Layouts & Partitioning. Cloud-based analytics services typically distribute data across multiple nodes or partitions, in order to scale out computation and load balance among computational resources. Data is often distributed either based on ingestion time, or using range, hash, or round-robin distribution schemes [106]. Automatic design advisors use what-if analyses and data mining to auto-tune the physical design and partitioning scheme [4, 139, 163]. Certain automated approaches are specialized for transactional workloads [32, 156, 158] or analytic workloads [52, 114]. MTO may be applied within each node or partition created by these schemes.

Qd-tree [195] (Section 4.1.1) and Sun et al. [176, 177] propose physical data layouts that maximize block skipping. Amoeba [172] adapts its partitioning to ad-hoc workloads. These approaches optimize the layout for a single table, whereas MTO jointly optimizes the layout for multiple tables.

Instance-Optimized Databases. There has been a recent research trend towards instance-optimized database systems and components. Whereas design decisions in traditional systems are often made though manual tuning or heuristics, the goal of instance-optimized systems is to automatically specialize database components and algorithms to a particular use case, sometimes using machine learning. MTO and qd-tree [195] are frameworks for instance-optimized data layouts. [11] introduces layouts for hybrid read-write workloads tailored to the data and query workload. Recent works have proposed instance-optimized, or learned, approaches for partition advising [73], tuning [184], data structures and indexes [97, 60, 192, 138, 77, 47, 49, 109], query optimization [125, 102, 152, 127], cardinality estimation [51, 90, 197, 196], job scheduling [122], workload forecasting [117], and complete database systems [95].

Sideways Information Passing. Similar to MTO's join-induced predicates, data-induced predicates [85] (Section 4.2.1), column equivalence [53], and magic-set rewrit-

ing [170] can also be used to push predicate information through joins. The performance benefit of these techniques depends on the data layout (e.g., pushed predicates cannot help skip blocks if every block contains records satisfying the predicate). MTO uses join-induced predicates to explicitly construct a layout that maximizes opportunities for block skipping during execution.

During query execution, sideways information passing between two joined tables or subexpressions, often in the form of semi-join reduction [18], can be used to skip blocks and speed up joins [82, 19, 153]. In contrast, MTO performs sideways information during offline optimization in order to produce a better joint layout for multiple tables.

Some auxiliary data structures cache useful information about joining tables. These include materialized views, join indexes [148], and join zone maps [149]. These data structures use extra storage space and incur maintenance overhead. In contrast, MTO does not duplicate any of the base data.

4.8 Conclusion

One of the dominant costs for query processing in cloud-based data analytics services is the I/O for accessing large data blocks from cloud storage. Per-block zone maps are a commonly-employed technique for reducing I/O by skipping blocks, but their effectiveness is dependent on how the records are assigned to blocks, i.e., the data layout. Existing approaches for optimizing data layouts only target a single table, and their performance suffers in the presence of join-based queries. In this chapter, we propose MTO, a data layout framework that automatically and jointly optimizes the blocking strategy for all tables in a multi-table dataset for a given query workload. We show that by taking advantage of sideways information passing through joins during the optimization process, MTO produces layouts that achieve up to 93% reduction in blocks accessed and 75% reduction in end-to-end query times on a commercial cloud-based data analytics service.

Chapter 5

SageDB: An Instance-Optimized Data Analytics System

In the previous chapters, we have introduced individual instance-optimized physical design components, focusing on indexes (Chapter 2) and data storage layouts (Chapters 3 and 4). In this chapter, we explore what it takes to synthesize multiple instance-optimized physical design components into an end-to-end database system under a global optimization policy.

Most modern data management systems fall on a spectrum between general-purpose and application-specific. For example, PostgreSQL [1] is extremely general purpose, and powers a diverse range of analytical and transactional workloads. Apache Spark is slightly specialized towards analytic tasks, but can still handle a wide variety of use cases (e.g., batch reporting, ad-hoc interactive queries, data science, and ML) and low-level workloads (e.g., I/O-bound, CPU-bound, in-memory, on-disk, in the cloud). On the other hand, systems like Google's Mesa [69] and Napa [3] were custom-built to power Google Ads, and are not suitable for any other application. While these systems improve efficiency, these bespoke systems require years of intense engineering effort and are only achievable by large corporations with significant resources.

Ideally, users should be able to have the efficiency of specialized systems along with the flexibility of general-purpose systems. Tuning configuration options ("knobs") is easier than building an entirely new system, and can bridge some of the performance gap. However, experienced engineers and database administrators still go through the time-consuming and error-prone tuning process for each application. Recent research proposes techniques for automatic knob tuning [27]; however, the performance impact of tuning such knobs is still limited. For example, users can only adjust the size of a data block, not how data is laid out on disk. Fundamentally, general-purpose systems are designed to be task agnostic, so for most tasks a tuned general-purpose system will perform worse than a custom-tailored system.

In previous chapters, we have shown that existing system components can be replaced with *instance-optimized* or *learned* components, which are able to automatically adjust to a specific use case and workload. For example, learned index structures (Chapter 2) offer the same read functionality as traditional index structures (e.g. B+ trees) while providing better performance in both latency and space consumption.

Instance-optimized data storage layouts (Chapters 3 and 4) are able to improve scan performance by skipping data with greater effectiveness than traditional sorting-based partitioning techniques.

However, these instance-optimized components have largely been designed and evaluated in isolation, and there have only been a few efforts to integrate them into an end-to-end system. Bourbon [36] replaces block indexes in an LSM-tree with learned indexes and demonstrates latency improvements. Google integrated learned indexes into BigTable [2] with similar findings, mainly due to a smaller index footprint and fewer cache misses when traversing the index. While these are useful initial studies, it is still unclear how multiple instance-optimized components would work together in concert. In fact, it is easy to imagine a number of learned components destructively interfering with each other. Is it possible to build a system that autonomously custom-tailors its major components to the user's requirements, approaching the performance of a bespoke system but with similar ease of use as a general-purpose system?

To the best of our knowledge, there is no end-to-end data system built with instance-optimization as a foundational design principle. We previously presented our vision and blueprint for such a system, called SageDB [96]. In this chapter, we present our first prototype of SageDB, and show how two carefully selected components can work together in practice. These instance-optimized components are (1) (multi-dimensional) data layouts and data replication and (2) *partial* materialized views. These techniques minimize I/O when scanning data from disk and maximize computation reuse through intelligent pre-materialization of partial results. While the ultimate goal is to automatically trigger self-optimization whenever necessary, for the current prototype we decided to expose a single easy-to-use command to the user — `OPTIMIZE` — with a user-given space budget. Doing so gives the user control over when SageDB should start to instance-optimize the internal components to improve performance for the user's workload while respecting the space constraint.

Building a usable database takes years and several attempts (e.g., Oracle took until version 7 to become stable), so this chapter should largely be regarded as a progress report on how to integrate learned components and the potential benefits they can provide when combined. As such, this chapter aims to inform the research and industry communities about the potentials, limitations, and future research challenges of learned instance-optimization.

In summary, we make the following contributions:

1. We introduce two new instance-optimized techniques: partial materialized views (PMVs), which is a generalization of traditional materialized views with more degrees of freedom, and replicated data layouts, which combines the idea of instance-optimized data layouts with partial table replication.
2. We introduce a global optimization algorithm that jointly and automatically configures partial materialized views and replicated data layouts given the user's data and workload. As a result, a user only needs to decide when to issue the `OPTIMIZE` command, and SageDB will automatically decide how to simultaneously configure all instance-optimized components.
3. We present an evaluation of our prototype implementation of SageDB against other systems, including a commercial cloud-based data warehouse product, which

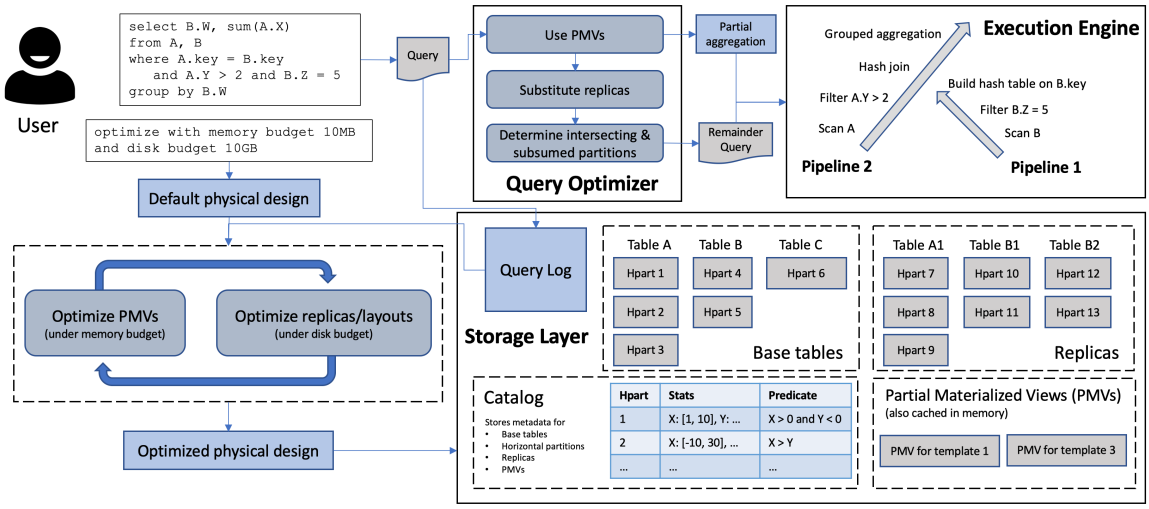

Figure 5-1: A user query passes through the rule-based optimizer, which determines if and how to use SageDB's instance-optimized components, then runs on SageDB's vectorized execution engine. When users issue an `OPTIMIZE` command, SageDB automatically configures its instance-optimized components to maximize performance based on the user's query history.

SageDB outperforms by up to 3× on end-to-end query workloads and up to 250× on individual queries.

In the remainder of this chapter, we give background (Section 5.1), present an overview of SageDB (Section 5.2), introduce its instance-optimized components (Section 5.3) and how that complexity is surfaced to the user (Section 5.4), present experimental results (Section 5.5), discuss remaining design challenges and future work (Section 5.6), review related work (Section 5.7), and conclude (Section 5.8).

5.1 SageDB

In this section, we provide a brief overview of the state of research on instance-optimized systems. Then we describe our motivations and design principles for building SageDB.

Background. Instance-optimization (a term inspired by the definition of instance-optimal algorithms [165]) refers to specializing a system based on the dataset and workload to achieve performance close to specialized solutions [96]. While there exists many possible ways to create instance-optimized components, a common approach is to tightly couple a model of the user's workload with a novel data structure designed to take advantage of that model. Sometimes, this approach is also referred to as learned systems or algorithms with predictions/oracles [80]. For example, learned indexes [99] model the user's data to accelerate searches on that dataset. Instance-optimized data layout techniques [137, 195] create workload-specific physical designs that minimize I/O during query execution. Past work tended to improve performance for a single instance-optimized component in isolation, but not for the entire database.

For example, learned indexes were evaluated on single-key lookup workloads instead of complete transactional workloads, and data layouts were evaluated on selective scan-heavy queries. Note that instance-optimized systems are fundamentally different from automatic knob-tuning approaches. Knob-tuning optimizes the hyperparameters of a system and is agnostic to the underlying data distribution. Instance-optimization designs systems that take advantage of knowledge about the specific data and/or workload distribution.

Motivation and Design Principles. We had two motivations for building SageDB. First, we aim to show that instance-optimization can provide benefits for end-to-end workloads with diverse query patterns instead of just database components evaluated in isolation. Second, we hoped that building and evaluating SageDB on real data and workloads would identify the most important pain points and roadblocks and guide us towards the most impactful directions for future work in instance-optimized systems. Like many existing learned components [124, 99, 195], we focus on analytic workloads as well. We leave investigation of instance-optimization for transactional workloads to future work.

We used several general principles to guide our design:

1. **Avoid regression.** One of the biggest deterrents to the adoption of instance-optimized techniques in practice is the fear that they might result in catastrophic failures or performance regressions under changing or even adversarial workloads. This fear of regression often outweighs the promise of potential performance improvements. In SageDB, we err on the side of caution: we must consider a component's downsides just as carefully as its upsides, and it must be simple to disable the component if necessary. The worst case should be no impact—not negative impact.
2. **Minimize the burden on the user.** Configuring the components should require as little as possible from the user, both in terms of interaction and understanding. The complexity of incorporating new instance-optimized components into SageDB should be completely hidden from the user—they should not need to read more documentation or issue new commands in order to make use of those new components. Accordingly, SageDB is designed such that the user only needs to issue a single `OPTIMIZE` command to trigger all optimizations.
3. **Avoid negative interference.** When combining a number of learned components, it is natural to worry that optimizing each component individually might not lead to an optimal global configuration. In the worst case, different learned components might "step on each other," degrading system performance. We must carefully consider how each component affects the others.

5.2 Design Overview

In this section, we provide a high-level overview of SageDB as a system and its instance-optimized components. Section 5.3 describe the instance-optimized components in more detail, and Section 5.4 covers the global optimization procedure. Fig. 5-1 provides an overview.

5.2.1 System

Storage Layer

SageDB stores data and performs query execution on a single node. SageDB by default stores data in columnar format, although row-store format is also available. The records of a table are divided into *horizontal partitions*. Each partition is stored as a separate file; each column of each partition can be accessed individually, without reading other columns. String columns are dictionary encoded, and integer columns are compressed using bit-packing.

For each horizontal partition, we store statistics used for execution-time data skipping, including the minimum value, maximum value, and number of distinct values for each column. In addition, we optionally store a predicate for each partition, with the property that all records in the partition are guaranteed to satisfy the predicate (see Section 5.3.2 for details). When a query scans from a table, SageDB compares the query's filter with the per-column statistics and the optional predicates to determine the set of horizontal partitions that can be skipped, i.e., the partitions for which the statistics and predicate guarantee that no row can match the filter. SageDB uses memory-mapped file I/O for data files stored on local SSD or disk. For long-term persistence, data files are stored on AWS S3 or other cloud object stores.

Query Optimizer and Execution Engine

SageDB has a vectorized execution engine that processes a chunk of data at a time. SageDB uses non-compiled pipelines with push-based execution (see Fig. 5-1 for an example). The first pipeline for each table involves scanning data from disk, for which the granularity of a chunk is a horizontal partition. Each pipeline may involve a projection over the columns or a filter over the rows. SageDB supports lazy materialization by maintaining a bitmap of relevant rows and passing the bitmap through the pipeline. SageDB uses multi-threaded parallel execution of pipelines.

SageDB has a rule-based query optimizer that determines the minimal set of columns that need to be read from each table, determines which horizontal partitions to scan from each table by using per-partition statistics and predicates to skip irrelevant partitions, orders tables for hash joins so that the largest table is the probe side, and constructs the execution pipelines.

Usage and SQL Support

We assume that queries issued by the user contain meaningful patterns and are not completely ad-hoc. More formally, we assume that user queries can be categorized into *templates* (also referred to as prepared statements), which are queries whose filters contain changeable parameters. For example, the template in Fig. 5-2 has parameters which are represented in the SQL text by `?`. SageDB gives users the ability to explicitly create these templates and issue queries by specifying the template ID and the parameter values, as shown in Fig. 5-2.

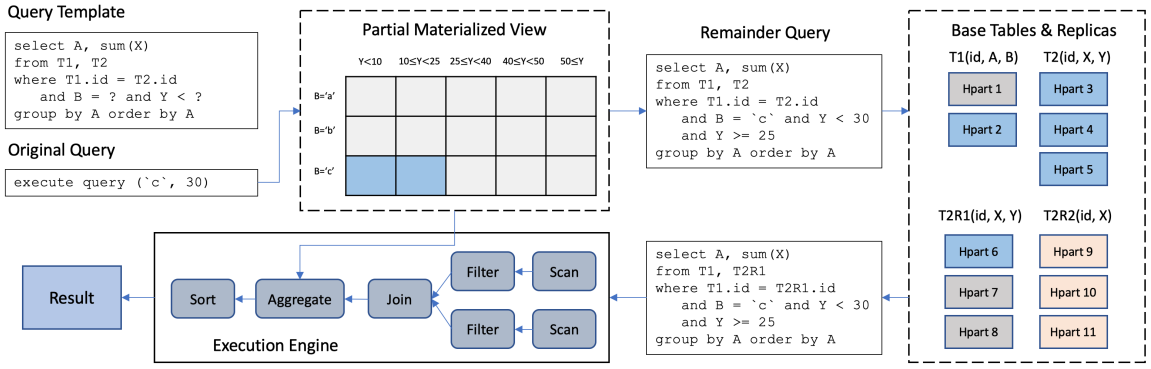

Figure 5-2: The example query takes advantage of the partial materialized view (PMV) to produce a remainder query with a more selective filter. It then reads from a replica instead of the base table in order to reduce scan cost.

SageDB supports a command-line SQL interface as well as a Python connector library. Users can load data into tables from either CSV files or Parquet files. SageDB returns query results to the user in JSON format. SageDB currently supports select-project-join-aggregate queries that can contain `GROUP BY`, `ORDER BY`, `HAVING`, `LIMIT`, `DISTINCT`, and analytic functions. Supported aggregation functions include `COUNT`, `COUNT DISTINCT`, `COUNT APPROX DISTINCT` (using HyperLogLog [58]), `SUM`, `AVG`, `MIN`, and `MAX`, including multi-attribute aggregations. SageDB supports nested queries through through unnesting [142] and treats CTEs as temporary tables. SageDB only supports inner equijoins, implemented as hash joins. SageDB also supports `INSERT`, but it is not a focus of the current design.

5.2.2 Instance-Optimization

What distinguishes SageDB from traditional systems is the degree to which it is able to customize its design for a specific use case. Many of the techniques that traditional analytic systems use to optimize for a given dataset and workload fall in two categories. First, users are allowed to create materialized views, which are used at query time to substitute a subquery or the entire query itself. This can result in serious performance improvements—some systems' performance relies almost entirely on aggressive use of materialized views [3]—and significant commercial effort has been put on automating materialized view selection [8, 147], maintenance [128], and matching [64].

Second, users are allowed to specify how the records of a table should be sorted. Classically, each table can be sorted by a specified column (i.e., the sort key), and some systems aim to automate sort key selection [7], but newer systems now also support multi-column sort orders such as the Z-order [198, 38]. Contiguous chunks of the sorted records are grouped into blocks, and systems traditionally store per-block metadata, such as the minimum and maximum value for each column [149, 29, 39, 130], which are used to skip irrelevant blocks during query processing.

Materialized views and data layouts can have a significant impact on performance. However, in traditional systems they are used independently of each other, and

furthermore, they are limited in complexity, which can limit their effectiveness. SageDB takes both components, expands their scope to go far beyond the capabilities of traditional systems, and combines them under a single global optimization objective. In particular, SageDB introduces the concept of *partial* materialized views, and SageDB uses instance-optimized block-based data layouts in combination with data replication. We explain these components in depth in Section 5.3, but we first briefly provide high-level intuition by presenting an example of their usage (Fig. 5-2).

An Illustrative Example

Assume there are two tables: a small table `T1` with columns `(id, A, B)`; and a large table `T2` with columns `(id, X, Y)`. Assume that the user creates a query template:

```
select A, sum(X) from T1, T2
where T1.id = T2.id and B = ? and Y < ?
group by A order by A
```

Assume that `A` and `B` are a low-cardinality categorical columns, while `Y` is a high-cardinality column whose values are unique floating-point numbers. A *traditional* materialized view for answering queries following this template would look like:

```
select A, B, Y, sum(X) as sumX from T1, T2
where T1.id = T2.id
group by A, B, Y
```

When the user issues a query using this template by specifying values for the parameters, the engine would answer the query directly from this materialized view instead of scanning the base tables, `T1` and `T2`, with a query such as:

```
select A, sum(sumX) from MaterializedView
where B = ? and Y < ?
```

However, since column `Y` has unique values, the materialized view has as many rows as the base table `T2`. This makes the materialized view expensive to store and also greatly reduces its performance benefits. In fact, executing using the materialized view might be slower than scanning the base tables. In this example, the engine would need to scan four columns from the materialized view and apply filters to two of those columns, whereas the original query would only need to read three columns from `T2` and apply filters to one column (the cost of reading and filtering the smaller `T1` are negligible) and perform a potentially inexpensive join.

To avoid the limitations of traditional materialized views, SageDB introduces the concept of *partial* materialized views (PMVs). A PMV is associated with a specific query template. Each cell in the grid (Fig. 5-2) represents a filtered subset of the joint data distribution of the base tables. For example, the top-left cell represents the data of `T1` and `T2` (joined by `id`) that satisfies the predicate `B='a' and Y<10`. Note that a PMV's grid is specific to a certain join pattern, namely, the join pattern observed in the template.

Each cell stores the result of the template's aggregation over only the data that it represents. For example, the top-left cell would store a relation that is equivalent to the result of executing

```
select A, sum(X) from T1, T2
where T1.id = T2.id and B='a' and Y<30
group by A
```

When executing a query following this template (Fig. 5-2), the SageDB query optimizer will find the cells that are *subsumed* by (i.e., entirely contained within) the query's filter, which for the example query in the figure are the two cells highlighted. We then produce the *remainder query*, which is the query whose filter has removed the parts that are already subsumed by the PMV (details in Section 5.3.1) and is therefore much more selective.

Furthermore, unlike traditional systems which allow users to specify a sort order for each table, SageDB has the ability to create multiple partial replicas for each base table (i.e., a replica containing a subset of the columns but all of the records of the base table), each with their own instance-optimized data layout. Before executing the remainder query, SageDB's optimizer considers whether to scan from the base table or a replica. Fig. 5-2 shows that there are two replicas of `T2`, namely `T2R1` with columns `(id, X, Y)` and `T2R2` with columns `(id, X)`. Imagine that the data layout for `T2R1` has specifically been optimized for the template (Section 5.3.2 presents more details), so that we would only need to read one horizontal partition from `T2R1` (highlighted in blue), whereas we would need to read all horizontal partitions from `T2`. Therefore the SageDB optimizer would substitute `T2R1` into the query. Note that we cannot use `T2R2` because it does not contain all the necessary columns.

Finally, the modified remainder query is fed to the execution engine, and the partial aggregations from the two subsumed PMV grid cells are merged during the aggregation step.

This example shows how SageDB's instance-optimized components work to reduce query execution cost: first, the PMV eliminates part of the query filter, which reduces the cost of joins (because the join inputs are smaller) and aggregation (because we aggregate fewer records). Second, substitution of base tables with replicas reduces scan cost by reducing the number of horizontal partitions scanned. The former technique is not easily supported in traditional systems; the latter is supported in traditional systems but is limited to simple data layouts (e.g., sort keys) and requires the user to manually specify replicas and layouts, whereas SageDB uses automatically-configured instance-optimized data layouts.

An important part of our contribution is not only *supporting* these techniques in SageDB, but also *automatically optimizing* their configuration. In particular, the performance of each component is dependent on the other. In Section 5.4, we describe our algorithm for co-optimizing these components given the user's data and workload.

5.3 Instance-Optimized Components

In this section, we more formally introduce SageDB's instance-optimized components, which we gave intuition for in Section 5.2.2. First, partial materialized views (PMVs) are a novel technique for generalizing traditional materialized views with more degrees of freedom. Second, although the idea of combining instance-optimized data layouts

with data replication has been proposed in [175], SageDB's replicated data layouts applies them to the novel context of disk-resident datasets composed of multiple tables, and we introduce a novel optimization algorithm (Section 5.4.4).

5.3.1 Partial Materialized Views

A partial materialized view (PMV) is associated with a specific query template. For a given query template (see Section 5.2.1), a *templated column* is a column that is directly involved in a filter predicate that includes a parameter. For the template in Fig. 5-2, the two templated columns are `B` and `Y`. A partial materialized view (PMV) for a given query template is logically defined as a grid over the templated columns. If a templated column is used twice in the same template (e.g., the filter includes `Y > ? AND Y < ?`), the column is only used once in the grid. For each grid cell, the PMV stores the result of executing the query template over only the data represented by the grid cell.

In concept, several templates can share the same PMV. For example, two templates that have the same filter and group-by clauses but have different aggregations (e.g., template 1 computes `SUM(A)` but template 2 computes `MIN(B)`) can share the same grid. However, this reduces our flexibility to adjust the amount of resources (i.e., memory budget, see Section 5.4.3) allocated to each template. For example, it is inefficient for a infrequently-queried low-cost template and a frequently-queried high-cost template to share the same grid; instead, the former should have a coarse-grained grid with fewer cells that uses low memory and the latter should have a fine-grained grid with more cells that uses more memory. It is therefore unlikely that two templates have the same optimal PMV grid. Therefore, we decide in SageDB to limit each PMV to a single template.

Construction

Given a PMV grid definition, we construct PMV in a single pass over the data. In fact, the construction can be posed as a SQL query. For example, the PMV in Fig. 5-2 is constructed as:

```
select A, [CASE WHEN B='a' AND Y<10 THEN 1 ELSE WHEN...], sum(X)
from T1, T2 where T1.id = T2.id
group by A, [CASE WHEN B='a' AND Y<10 THEN 1 ELSE WHEN...]
```

where the `CASE` expression will output a cell number based on the record's value in the templated columns[1]. Note that in the construction query, we remove the parameterized filter predicates, but leave remaining filter predicates as-is (e.g., if there were an additional predicate `AND X>0` in the template).

[1]Instead of having a case for every cell, an optimization is have a `CASE` expression for each grid dimension individually, and then combine them to form a unique cell number.

Usage

To use PMVs at query time, we first logically determine which cells are subsumed by the query filter. We then exclude those regions of the data space from the filter. To determine which cells are subsumed, we break down the filter into its atomic components by splitting apart `AND`s and `OR`s. The example query in Fig. 5-2 has one `AND`, and therefore two atomic components. Any atomic component that only references a single templated column can be checked against the corresponding grid dimension. For the query in Fig. 5-2, the atomic component `B='c'` is checked against the partitions of grid dimension `B`, and we see that only one partition is subsumed, and the atomic component `Y<30` subsumes the two partitions that, when combined, represent `Y<25`. An expression describing the subsumed cells can then be constructed by re-combining the atomic components, e.g., `B='c' AND Y < 25`.

To modify the query filter, we add a `NOT` of an expression describing the subsumed regions. For the example query in Fig. 5-2, the remainder query is

```
select A, sum(X) from T1, T2
where T1.id = T2.id and B='a' and Y<30 and not (B='a' and Y<25)
group by A
```

Note that the expression in parentheses describes the subsumed cells. This may result in an overly complicated filter, but the SageDB optimizer uses an SMT solver [43] to simplify filters into conjunctive normal form (CNF) before passing it to the execution engine.

SageDB caches partial materialized views in memory but they are also persisted to disk and cloud storage.

Strengths and Limitations

The scope of PMVs is quite broad. PMVs can be used for nearly any query template with parameterized filters, since the usage technique is very generic. This idea extends to multiple templated columns, and also to queries with joins (such as the one in Fig. 5-2), for arbitrary filter predicates (containing both `AND`s and `OR`s).

However, there are some scenarios in which PMVs are unlikely to help (see Section 5.5.3 for experiments). For templates that produce large aggregations (i.e., group by high-cardinality columns), the PMV becomes expensive to store and has limited benefits[2], similar to the limitation of traditional materialized views presented in Section 5.2.2. Also, if there are many templated columns, the high-dimensional PMV grid is less effective at isolating subsumed cells due to the curse of dimensionality (e.g., Gaming Q4, Section 5.5.3).

[2]Typically, these types of queries include a `LIMIT` clause (e.g., TPC-H Q10). Unfortunately, we cannot simply take take a `LIMIT` within each cell of the PMV grid, because a global top-K is not equivalent to merging the top-K of each cell.

5.3.2 Replicated Data Layouts

Prior work on instance-optimized data layouts [137, 48, 195, 45] has already shown that more complex data layouts perform better than traditional single-column or multi-column sort orders. However, these prior instance-optimized techniques assume that they modify the original copy of the data. In SageDB, we want to avoid this because it violates our design principle of avoiding regressions, because an unexpected future query may execute slower on the "optimized" layout than the original layout.

In SageDB, we do not modify the layout of the original copy, which we refer to as *base tables*. Instead, we use a user-provided additional disk space budget to create partial replicas of tables. A partial replica contains a subset of the columns from the base table (which may be the full set). The data layout for each replica is independent. For each query, the query optimizer chooses to read from the replica (or the base table) that minimizes scan cost. The challenge is to determine which subset of the workload to optimize each replica for in order to achieve the best performance, since it does not make sense to optimize multiple replicas for the same query/template if the execution engine only uses one replica at execution time—we examine this optimization problem in Section 5.4.4.

Construction

For a given replica (i.e., a subset of columns from a base table) and a set of queries to optimize the replica's data layout for, we use the same algorithm as in [195] to create a set of horizontal partitions. SageDB defines a target number of records per horizontal partition, which by default is set to 2M rows based on the latencies we observed for Amazon S3[3]. Each block is associated with a predicate, with the property that all records in the block satisfy the predicate, and also all records that satisfy the predicate are in the block, i.e., blocks do not "overlap." For brevity, we omit the details of the algorithm, which can be found in [195] and is summarized in Section 2.1 of [45].

Usage

For each table referenced in the query, the SageDB optimizer iterates over the replicas, first checks whether the replica contains all the necessary columns, and checks the per-horizontal partition metadata to determine the number of files and rows that need to be read from each, and picks the replica with the lowest cost (see Section 5.4.1). This procedure is done for each table *independently*, because substituting replicas purely improves scan cost. Downstream operators that introduce dependencies between tables, just as joins, are not affected.

Strengths and Limitations

Replicated layouts have the greatest impact on reducing cost for scan-heavy queries with selective filters. However, replicated data layouts only help reduce scan cost

[3]Future versions of SageDB will automatically tune this parameter based on the observed performance.

(by skipping irrelevant data blocks) but cannot reduce the cost of other parts of query execution, such as joins, and are therefore less effective for queries where joins dominate execution time (see Section 5.5.3 for examples). Furthermore, if the query filter is extremely complex (e.g., composed of many conjunctions and disjunctions over many columns), then even instance-optimized data layouts may not be able to meaningfully outperform a full table scan, due to the curse of dimensionality.

5.4 The `OPTIMIZE` Command

The user can issue the `OPTIMIZE` command to trigger automatic configuration of SageDB's instance-optimized components. The command has two arguments, a budget for the amount of memory space that SageDB can use to store PMVs, and a budget for the amount of disk space that SageDB can use to store replicated data layouts. The user is allowed to set either budget to zero, though this would of course limit the effectiveness of the optimization.

The user's only responsibility is to decide when to issue the `OPTIMIZE` command. We envision that the user runs the command during a time of low system load, so that the optimization process does not affect performance of concurrently running queries; this is the same advice that data warehouse providers typically give to users when suggesting knob tuning recommendations. Ideally, the user should have already issued a representative set of queries on SageDB, because the optimization will require examining and modeling the user's query history. For example, if the user uses SageDB to run a daily batch reporting job, then they may want to run the first day's batch, then issue the `OPTIMIZE` command overnight, so that the next day's batch can take advantage of performance improvements.

When the user triggers the `OPTIMIZE` command, SageDB needs to automatically configure its instance-optimized components simultaneously. Why not simply optimize PMVs and replicated layouts independently, each on the full query workload? The choice of PMVs affects the optimal replicated layouts, because PMVs produce remainder queries and in some cases answer the entire query, so the layout should only be optimized for the remainder queries. The choice of replicated layouts also affects the optimal PMVs; depending on how effective the layouts are at processing a template's remainder queries, we may want to allocate more or less memory budget for that template's PMV (e.g., a PMV is useless if the remainder query would anyway require scanning all of the data because of a poor data layout).

SageDB uses an iterative algorithm that optimizes PMVs and layouts, dependent on the other, in a loop until convergence. We now describe the cost model which forms the optimization objective, then the global optimization procedure.

5.4.1 Cost Model

SageDB uses an analytic cost model. The cost of a query is the sum of scan cost, join cost, and aggregation cost:

$$\begin{aligned}\text{ScanCost} &= w_0(\#\text{ horizontal partitions scanned}) \\ &\quad + w_1(\#\text{ scanned records})(\#\text{ columns read}) \\ \text{JoinCost} &= w_2(\#\text{ build side records}) + w_3(\#\text{ probe side records}) \\ &\quad + w_4(\#\text{ output records})(\#\text{ output columns}) \\ \text{AggCost} &= w_5(\#\text{ aggregated records})\end{aligned}$$

Scan cost and (hash) join cost are evaluated for each table/join, while aggregation cost is computed for the post-join relation. The weights w_i are tuned based on the hardware. To estimate the features, we use a simple cardinality estimator which assumes independence between columns and uniform data distributions of the values in each column. We could use a more complex cost model, or even a learned cost model, but that is orthogonal to the core optimization technique.

5.4.2 Global Optimization

The optimization objective is to minimize total workload cost, i.e., the sum of costs, according to the cost model, for all queries in the workload. The algorithm is as follows:

1. The catalog stores a log of all past user queries. We examine that history and cluster queries into *templates*. A template is a query for which constant literals in the query filter are replaced by placeholders. Within each template, if a certain placeholder always has the same constant value, we remove the placeholder and simply use the value. We expect that many real workloads (e.g., daily batch reporting jobs, dashboard queries) have repeated query patterns and are naturally composed of templates.
2. Starting from the default physical configuration, which only contains the base tables in their original layout and has no PMVs and no replicas, perform the following steps in a loop, until the relative cost decrease from the previous iteration of the loop is less than a certain threshold, by default 1%:
   (a) Optimize the PMVs, using an objective function that takes the current replicated layout configuration into account (see Section 5.4.3).
   (b) Feed all queries through the optimized PMVs to construct a workload consisting only of remainder queries.
   (c) For each remainder query with joins, push down all single-table predicates to their respective tables and create a single-table query for each table.
   (d) Optimize the replicas and data layouts on the single-table remainder queries (see Section 5.4.4). Each table is optimized only for the queries that filter on that table.

The intuition behind the loop is to incrementally optimize each component given the currently-optimized configuration of the other component. For example, the first time

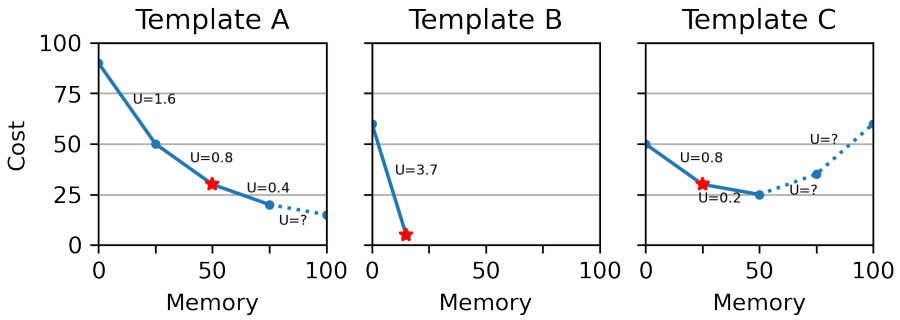

Figure 5-3: Optimizing PMVs for three templates with a total memory budget of 100 and step size of 25. Utility is visualized as the slope of the lines. Red stars represent the selected configurations. Dotted lines would not actually be considered in the optimization.

that PMVs are created, some PMVs may be rejected because the remainder queries would anyways be very expensive on the default layout. However, after the layouts are optimized once, it is likely that those PMVs are selected in the next iteration, because the layout is now optimized. This way, the dependencies between PMVs and replicated layouts are captured.

There are no regressions from one iteration to the next (i.e., cost can only decrease) because the algorithm can always select to choose the same PMVs or replicated layouts as the previous iteration.

5.4.3 Optimizing Partial Materialized Views

Given a memory budget and a set of templates T, we need an algorithm to decide how much memory to allocate to building a PMV for each template, and also what the PMV grid should be given that memory allocation. We first describe the latter, since it is used as a subroutine in the former.

Optimizing the PMV grid.

If a query template t has n templated columns, then a PMV for t is a grid with n dimensions, one representing each templated column (see Fig. 5-2 for an example). The domain of each dimension's values are logically divided into equally-sized buckets, much like an equi-depth histogram: for each dimension $i \in [0, n)$, we create b_i buckets by setting the boundary values between buckets in such a way that $1/b_i$ of all records fall in each bucket. For templated columns involved in equality filters (i.e., `=`, `!=`, `IN`), we ensure that each bucket only contains one unique value of that column, since a bucket is only useful if the filter is able to subsume it; if there are more unique values than buckets, we only use the b_i most frequent values.

Given a query template t and a space budget s, we want to configure a PMV grid, i.e., set b_i for $i \in [0, n)$, that minimizes the total cost, according to our cost model, of executing all queries in the workload that come from t. (Note that scan cost of the

remainder query *depends* on the current replicated layout configuration, in the context of the global optimization algorithm in Section 5.4.2.) We use Bayesian optimization to determine the b_i for each dimension that minimizes cost, under the space budget constraint s. Throughout this process, we make use of SageDB's ability to simulate a PMV without physically creating it, by only storing the PMV metadata (i.e., the grid definition), which is used to generate remainder queries as if the PMV actually existed and to estimate the memory usage of the PMV.

Allocating memory to templates.

Given a total memory budget B and a set of templates $T = \{t_1, \ldots, t_n\}$, our algorithm aims to minimize $\sum_i C_i(s_i)$, under the constraint that $\sum_i s_i \leq B$, where $C_i(s)$ is the total cost of executing the queries from template t_i if we could create a PMV for t_i with space s, as described above. While solving this optimization problem, we would like to minimize the number of times we compute $C_i(s)$, since PMV simulation, while cheaper than physically creating the PMV, is still expensive.

We make the following observation: allocating more space to a template's PMV generally has diminishing marginal returns. That is, for a template $t_i \in T$, the cost function $C_i(s)$ is convex. For intuition, consider a simple template, `SELECT SUM(A) FROM T WHERE B < ?`, where column B contains numeric values. A PMV for this template would essentially divide the domain of column B into n equally-sized cells. By using the PMV, a query from this template would only ever need to scan/aggregate the data corresponding to one cell, because all cells to the "left" would be subsumed. Each cell contains around $1/n$ of the data, so the cost as a function of the number of cell is approximately $C(n) = 1/n$, which is convex. This intuition also roughly extends to higher-dimensional grids.

Therefore, the intuition behind the optimization algorithm is that instead of allocating the total memory budget across the different templates in one shot, we take an iterative approach where we incrementally allocate more space to the template with the highest impact on cost. Essentially, we do not know the cost functions $C_i(s)$ upfront, so we incrementally explore these cost functions, starting from $s = 0$. We now formalize this algorithm.

Our algorithm works by incrementally allocating b memory budget at a time, where $b < B$. We refer to b as the "step size." If a template t_i currently has a PMV that uses memory space s, we define the (marginal) utility $U_i(s, b)$ of allocating another b space as as $(C_i(s) - C_i(s + b))/b$. Throughout the optimization algorithm, we maintain a memo that stores, for each template t_i, three pieces of data: (1) the amount of space currently allocated to that template s_i, (2) the cost $C_i(s_i)$, and (3) the utility $U_i(s_i, b)$ of allocating b more space to the template. The algorithm proceeds as follows (Fig. 5-3 shows an example):

1. The memo is initially empty, i.e., zero space is allocated to each template. We begin by computing the utility $U_i(0, b)$ for each template. This requires computing $C_i(b)$ for each template t_i.
2. Pick the template with the highest utility: $\arg\max_i U_i(s_i, b)$. Change that template's entry in the memo. That is, if the entry was previously $(s_i, C_i(s_i), U_i(s_i, b))$,

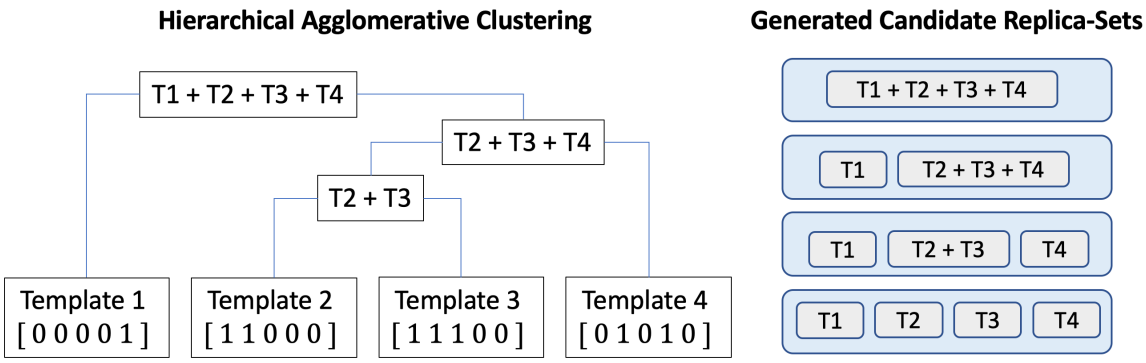

Figure 5-4: Hierarchical clustering of four query templates on a table with five columns, which produces four candidate replica-sets (blue) over seven distinct replicas (gray).

we now replace it with $(s_i + b, C_i(s_i + b), U_i(s_i + b, b))$. This requires computing $C_i(s_i + b)$. In case the PMV has reached its maximum size (i.e., we have done the equivalent of a traditional materialized view that groups by the templated columns), we do not consider this template further.

3. Repeat step 2 until the space budget is filled.

In general, a smaller step size b means a closer-to-optimal solution. By default, we set $b = B/n$, where n is the number of templates. This is small enough to guarantee that every template can get a PMV, if that is indeed optimal. Furthermore, this means that the complexity of the algorithm (i.e., the number of invocations of a cost function) is $O(n)$: we perform $O(n)$ cost function calls to initialize the memo, and we perform $O(B/b) = O(n)$ additional cost function calls before the budget is filled.

5.4.4 Optimizing Replicated Layouts

Assume the dataset is composed of m tables, $T_1, \ldots, T_m$. Given a total disk budget B and a set of single-table remainder queries Q_i for each table T_i, our algorithm aims to find the set of replicas, along with the data layout for each replica, that minimizes the total scan cost of all remainder queries. Note that we do not need to consider join cost or aggregation cost, since the amount of data scanned does not affect the inputs to downstream operators like joins and aggregations. Our algorithm has two steps: finding a collection of candidate *replica-sets* for each table, then selecting the optimal set of replica-sets.

Generating candidate replica-sets.

This step is repeated for each table T_i. Given $|Q_i|$ remainder queries, there are $2^{|Q_i|}$ possible replicas we could create, i.e., we could create a replica whose data layout is optimized for any subset of the queries. For simplicity, we consider each query template as one atomic unit, but nonetheless, given n templates, there are an exponential number of possible replicas, so a brute force search is infeasible. Instead, we generate a collection of promising *replica-sets*, i.e., a set of replicas along with their optimized data layouts.

Our insight is that we should only consider replicas whose layouts are optimized for a set of similar query templates. More concretely, we generate a embedding for each query template, in two different ways that represent two different notions of similarity:

1. A binary embedding (i.e., composed only of 0's and 1's) of columns that appear in the query filter (in both parameterized and constant predicates). Templates with similar embeddings will benefit from similar layouts. As an extreme example, three templates that all only filter on `colA` will both benefit from a replica that sorts records by `colA`, whereas a replica optimized for three templates that filter on three different columns would not do a great job for any template.
2. A binary embedding of columns that appear anywhere in the query template, i.e., the columns that the execution engine needs to read when processing this query. By placing templates with similar embeddings in the same cluster, we minimize the number of columns we would need to include in a replica for that cluster (recall that a replica does need to include all columns from the base table, only the columns necessary for execution).

There is a fundamental trade-off between space and cost: having a different replica for every different template will achieve the lowest scan cost, but will take the most disk storage space, while optimizing a single replica for all the templates takes the least space but will not reduce scan cost as much. The candidate replica-sets that we generate should form a Pareto frontier that spans this space-cost tradeoff.

Specifically, use hierarchical agglomerative clustering [173] over the embedded space to separate the n templates into anywhere from 1 cluster to n clusters (see Fig. 5-4 for an example). For each cluster, we generate a replica whose data layout is optimized (using the algorithm from [195]) for only the templates in its cluster, containing only the columns from the base table needed to execute the templates in its cluster. This results in n replica-sets for each type of embedding, and since we use two types of embeddings, we have up to $2n - 2$ unique replica sets (since the replica-sets corresponding to 1 cluster and n clusters will be the same for both embeddings). Due to the nature of hierarchical clustering, these replica-sets are built from up to $3n - 3$ unique replicas. Therefore, the time complexity of this step is $O(n)$. For each replica-set, we compute the scan cost of executing the queries Q_i, according to the cost model.

Selecting replica-sets.

After generating a collection of candidate replica-sets for each table, we need to select a global configuration of replicas, i.e., select zero or one replica-set for each table, that minimizes total scan cost under the space budget B. This optimization problem is almost identical to the 0-1 knapsack problem, so we use the standard dynamic programming solution to find the optimal collection of replica-sets. The only difference is that if we select multiple replica-sets corresponding to the same table, we only use the one that reduces scan cost the most.

If the query workload has n templates and m tables, we generate up to nm candidate replica-sets, so the standard dynamic programming algorithm for the 0-1 knapsack

Table 5.1: Dataset and workload characteristics.

	Gaming	Stack Overflow	TPC-H
num tables	5	1	8
num rows in largest table	3.06B	507M	600M
uncompressed size (GB)	426	52	100
num templates	13	13	15

problem takes $O((nm)^2)$ time. In practice, this is very fast, for several reasons: first, the time-intensive optimization steps (i.e., simulating PMVs and replicated layouts and feeding estimated statistics through the cost model) have already been done. Second, n is typically small (e.g., TPC-H has 22 templates). Third, even if there are many tables in the dataset, most of these tables are small; we do not even need to consider creating replicas for tables that only have enough records for one horizontal partition.

5.5 Evaluation

In this section, we present the results of an experimental study that compares SageDB with other data analytics systems on both real and synthetic datasets and workloads. Overall, this evaluation shows:

1. SageDB outperforms a commercial cloud-based analytics system by up to 3× on end-to-end query workloads and up to almost 250× on individual query templates (Section 5.5.2).
2. SageDB's instance-optimized components benefit different types of queries to different degrees, but almost all queries benefit from at least one instance-optimized component (Section 5.5.3).
3. SageDB's optimizations rarely result in regressions for individual queries, and the `OPTIMIZE` command can easily be completed as a nightly job (Section 5.5.4).

5.5.1 Setup

We run SageDB on a EC2 machine with 4 vCPUs and 32GB RAM (`i3en.xlarge`), with data on an attached EBS volume with 4000 IOPS. We compare against a popular cloud data warehousing product, which we call System X, running on a single node with the same number of cores and memory. We also compare against Umbra [141], a high-performance on-disk analytics research prototype by TUM, which incorporates many state-of-the-art techniques such as just-in-time code compilation, though it currently doesn't support indexes and cannot be tuned for a particular workload.

We evaluate using three datasets and workloads (Table 5.1). We include full dataset schema and workload specifications in Appendix B.

1. **Gaming** is a real-world dataset from the gaming division of a major technology company, donated to us under the condition of anonymity. There are two fact

tables, with roughly 2B and 3B rows respectively, and three smaller dimension tables. We use a real workload provided by the company.
2. **Stack Overflow** is a single-table dataset with 500M records, each of which represents a post on Stack Overflow.
3. **TPC-H** is a standard analytics benchmark. We use scale factor 100 to generate the data.

All experiments that involve running a query workload will first deterministically shuffle the order of queries (i.e., we want to avoid caching effects of running all queries of the same template sequentially). We then run the workload three times and report the median time for each query.

5.5.2 Overall Results

We first compare SageDB directly against System X and Umbra on the three datasets and workloads. We show two different configurations for SageDB: (1) unoptimized, the out-of-the-box version of SageDB before the user has issued the `OPTIMIZE` command, (2) optimized, the state of SageDB after the user has issued the `OPTIMIZE` command with a memory budget of 1GB (which is a small fraction of overall memory) and a disk budget equal to the size of the original dataset (which we believe to be reasonable since datasets are often fully replicated for fault tolerance anyway, especially on the cloud).

We show two different versions of System X: (1) the out-of-the-box configuration, after the data has been loaded. (2) A tuned version, in which we enable System X's ability to automatically select a sort key for each table, as well as automatically select materialized views. We believe that these capabilities represent the state-of-the-art in automated physical design in a large-scale commercial analytic system. To ensure we maximize System X's performance, we performed additional hand-tuning: we included hand-picked materialized views for each dataset, and for Stack Overflow, the tuned version also sorts the table using an interleaved sort key (i.e., Z-order) over the `post_date` and `score` columns, which improves performance because `score` is correlated with many of the commonly filtered columns. In summary, the tuned System X reflects the combination of automatic tuning and hand tuning. The disk storage cost of our manually-tuned materialized views are 40%, 2%, and 100% of the size of the original dataset for the Gaming, Stack Overflow, and TPC-H datasets respectively, which is higher than SageDB's 1GB budget for PMVs but smaller than its budget for replicated data layouts; System X does not allow users to access automatically-created materialized views, so the total storage cost of all materialized views is likely higher. Umbra does not use any extra storage space because it does not support indexes.

Fig. 5-5 shows that across the three workloads, SageDB outperforms the other systems on average query runtime by up to 3×. As evidence of the effectiveness of SageDB's instance-optimized components, SageDB optimized outperforms the unoptimized version of itself by between 3–6×, whereas System X tuned, which uses a combination of manual tuning and state-of-the-art automatic tuning, achieves between 25% and 3× performance gain over the default version of itself. Umbra performs best when the working set fits in memory; otherwise it is bottlenecked by disk since it

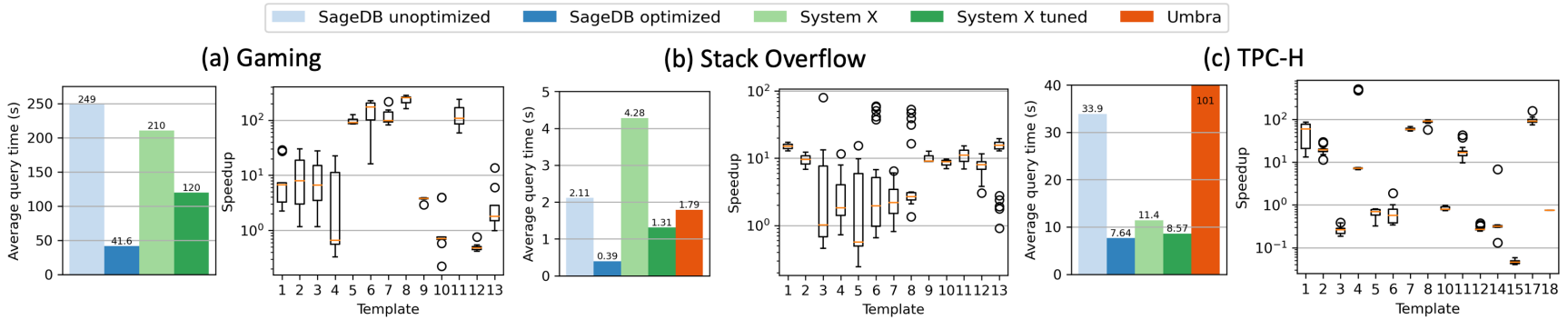

Figure 5-5: For each dataset, we show average query time on each system for the end-to-end workload, as well as a per-template breakdown of speedups achieved by SageDB optimized compared to System X tuned. SageDB outperforms other systems by up to 3× on end-to-end query workloads and achieves up to almost 250× speedup for individual templates.

doesn't use any indexes or layouts, which is why it performs poorly on TPC-H. Umbra was unable to complete all queries in the workload for Gaming, which is why we do not include it in the plot.

For each workload, Fig. 5-5 also shows a per-template breakdown of speedups achieved by the optimized version of SageDB compared to the tuned version of System X. For individual query templates, median speedups are as high as 250× (Gaming Q8). In general, templates for which SageDB performs worse than System X are ones for which SageDB's instance-optimized components do not make an impact (e.g., TPC-H Q18, see Section 5.5.3), ones for which the tuned System X has sort keys and materialized views that achieve the same purpose as SageDB's instance-optimized components (e.g., Gaming Q12), or ones for which System X's raw execution engine is simply more efficient than SageDB's (e.g., some TPC-H templates). The variability in speedups is simply due to the fact that the effectiveness of SageDB's instance-optimization depends not only on the query template, but also the specific parameter values of the template; for example, a parameter value that results in a non-selective filter may not benefit as much from replicated layouts as a selective filter.

While the performance numbers of SageDB are promising compared to System X and Umbra, it has to be pointed out that SageDB is still a prototype and is not yet feature-complete like System X (e.g., we do not support outer joins). Rather, there are two takeaways: first, SageDB as an out-of-the-box system, ignoring instance-optimized components, has roughly comparable performance to System X and Umbra when evaluated on the same hardware in a single-node setting. Second, and arguably more importantly, optimization allows SageDB to outperform the out-of-the-box version of itself by up to 6×. Next, we dive deeper in which how each instance-optimized component contributes to that performance gain.

5.5.3 Ablation Study

How much do each of SageDB's individual instance-optimized components contribute to the overall performance? In Table 5.2, we break down the effect of each instance-optimized components on each template of each workload. Overall, there are several takeaways.

First, different components help more for different types of queries. For example, replicated data layouts are especially helpful for queries that either filter on a single table (e.g., Stack Overflow queries, TPC-H Q1) or have inexpensive joins (e.g., TPC-H Q14). PMVs are helpful whenever they are applicable, and especially if it fully answers the query so that the remainder query is empty (e.g., Gaming Q8 and Stack Overflow Q1).

Second, SageDB's performance when all components are combined is sometimes better than any individual component on its own. For example, Stack Overflow Q4 and Q11 benefit from some synergy between PMVs, which answer most of the query, and then using the replicated data layouts to speed up the remainder query.

Third, there are some types of queries for which PMVs or replicated data layouts make no impact, as we alluded to in Sections 5.3.1 and 5.3.2. For example, TPC-H Q18 produces extremely large aggregations (since it groups by the primary key of a

Table 5.2: Ratio of average query time on the unoptimized SageDB vs. when the specified components is enabled. Higher is better. Highlighted is the component that makes the most impact on each template.

Gaming	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13		
PMVs only	1.7	1.68	1.72	1.0	51.9	99.7	93.8	1.28e+03	2.43	1.12	77.5	1.01	1.36		
Replicated layouts only	1.25	1.32	1.33	1.65	1.14	1.05	1.05	1.06	1.0	1.1	1.12	1.0	1.21		
All	3.1	3.76	3.11	1.65	51.9	1.17e+02	1.05e+02	1.33e+03	2.96	1.17	85.5	1.02	1.81		
Stack Overflow	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13		
PMVs only	10.5	1.36	1.55	2.16	1.9	1.39	7.75	0.648	1.39	6.3	1.2	2.78	2.04		
Replicated layouts only	7.0	1.53	1.95	3.19	3.38	4.33	9.95	2.43	3.61	13.6	1.09	2.21	15.3		
All	17.1	1.6	2.24	4.78	3.66	4.33	10.9	2.44	3.82	14.2	1.71	2.78	15.6		
TPC-H	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q10	Q11	Q12	Q14	Q15	Q17	Q18
PMVs only	3.17	25.9	1.0	1.18	2.04	1.0	2.47e+02	63.3	2.46	1.78e+02	1.08	1.09	1.02	27.3	1.0
Replicated layouts only	56.7	1.03	1.02	1.05	1.03	5.18	1.07	1.04	1.05	1.07	6.42	1.69	3.12	1.04	1.05
All	80.1	27.3	1.03	1.26	2.13	6.02	2.54e+02	67.2	3.32	2.02e+02	7.97	2.12	3.15	29.1	1.05

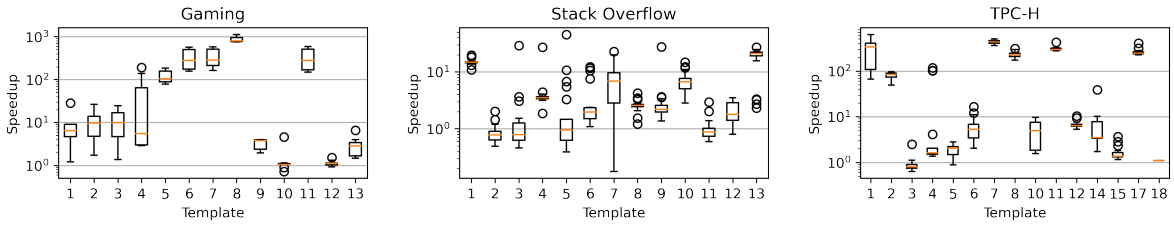

Figure 5-6: Per-query speedups for SageDB compared to its unoptimized configuration. Regressions are rare. The orange line represents the median; the box represents first and third quartiles; whiskers extend from the box by 1.5× the inter-quartile range; dots are those past the end of the whiskers.

table with 150M rows before applying a `LIMIT`), so a PMV is unhelpful and would exceed the memory budget anyway.

Fourth, occasionally using instance-optimized component is worse than not using it. For example, on Stack Overflow Q8, using PMVs decreases performance compared to the default. This is because SageDB always uses PMVs if they exist, but in this particular case, the query itself ran relatively quickly, and the extra optimizer overhead from computing subsumed cells in the PMV ate into the performance gains. This points to a direction for future work, which is to automatically determine, for each query, whether a certain instance-optimized component should be disabled.

5.5.4 Microbenchmarks

Regressions

SageDB improves overall performance, but we also want to ensure that individual queries do not regress. Table 5.2 showed that on a query template level, performance does not regress. Fig. 5-6 takes this a step further by breaking down individual query performance for each template, comparing the speedup in query runtime between the optimized and unoptimized configurations of SageDB. In general, regressions are rare, and when regressions do occur, they are minor compared to performance gains. Often, regressions are due to extra query optimization overheads for very short-running queries.

Space Budget

Fig. 5-5 showed SageDB's performance when optimized with 1GB memory budget and disk budget equal to the size of the original dataset. To show how performance would change if the budgets were set differently, we hold one budget constant while varying the other budget, on the Gaming dataset. Fig. 5-7 shows the overall workload cost as each budget varies, compared to the cost of having zero budget (i.e., if the corresponding component were disabled). As more space is given, cost decreases and performance improves. Note that the cost curve is convex for PMV optimization (note the log scale for the x-axis), confirming our intuition from Section 5.4.3. For replicated

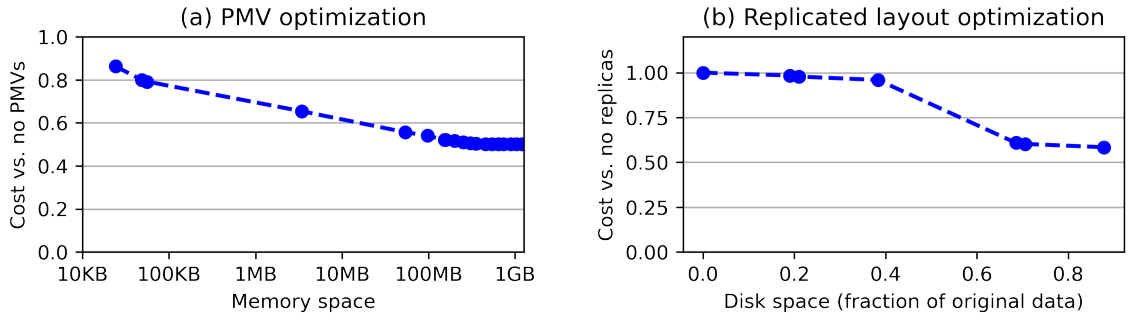

Figure 5-7: Gaming dataset: cost decreases as more space is provided to the `OPTIMIZE` command.

Table 5.3: Optimization Time (in seconds).

	Gaming	**Stack Overflow**	**TPC-H**
Optimization algorithm	107	117	143
PMV construction	4340	265	4150
Replicated layout construction	16100	1280	2400
Total	20500	1660	6690

layout optimization, there is a significant decrease in cost at 70% disk space because that is the boundaries past which an especially important replica fits within the space budget.

Optimization Time

We expect that users should trigger the `OPTIMIZE` command during a time of low system load, similar to what popular data warehouse products advise their customers to do when following recommended optimizations. Therefore, optimization should not interfere with normal workload execution.

Table 5.3 breaks down the time that SageDB spends on each step of optimization for each dataset. Overall, optimization finishes in less than 6 hours for the largest dataset, which reasonably fits into periods of low system load (e.g., overnight). Even if the optimization is performed while queries are running, this quickly pays off in terms of saved query time. For example, on the Gaming workload, since the benefit from optimization is around 200 seconds per query on average (Fig. 5-5), it only takes just over 100 queries executed to recoup the time "lost" to optimization.

5.6 Lessons Learned and Future Work

In this section, we take a step back and consider how the current SageDB design compares to our original design principles (Section 5.1). We also highlight important directions for future work.

Avoid regression. Due to SageDB's design, for any particular query we can always fall back to the default out-of-the-box configuration without instance-optimization. For example, the optimizer can always choose to read from the base table instead of from the replicas. This is in contrast to, for example, past work on instance-optimized layouts that directly modify the original data, which makes it impossible to fall back to the default. Indeed, we show in Section 5.5.4 that we avoid regressions on all templates.

The implication is that the burden of avoiding regressions (e.g., deciding to not use the PMV for a certain query) falls on the query optimizer, which may make mistakes due to inaccurate cost models or cardinality estimates. Besides integrating a more sophisticated query optimizer, one way to guard more aggressively against regression is to allow the optimizer to use the instance-optimized components only when cost reduction is greater than a certain threshold, i.e., be more conservative in using instance-optimized components.

Minimize the burden on the user. SageDB places minimal technical burden on the user: their only responsibility is to issue an `OPTIMIZE` command, with a space budget, during times of low system load. However, our longer-term vision is to remove all responsibility altogether by automatically deciding when to perform optimization and which components to re-optimize. This will require detecting when either the data or the workload have shifted significantly enough to merit a re-optimization, and can incorporate ideas from [45]. It will also require considering the cost of re-optimization itself, as well as forecasting the future workload—even if the workload has shifted, we may not want to optimize if the workload will shift again soon anyway.

Inserts. Allowing SageDB and other instance-optimized systems to adapt to inserts is a key area of future work. The main challenge behind inserts is that they may invalidate optimizations constructed based on a static snapshot of the data. Replicas with instance-optimized data layouts can avoid invalidation through delta buffering. For example, new data is inserted into a special horizontal partition. The existing horizontal partitions remain unchanged, and when scanning, SageDB can still take advantage of data skipping over existing partitions, but may need to always read the new partition. At a later time, when the user again triggers the `OPTIMIZE` command, the buffered data in the new partitions are incorporated into the new data layout.

Likewise, PMVs are not necessarily invalidated when data is inserted into a new horizontal partition, especially if data is only changing in one base table (e.g., users append data to a fact table but the dimension tables are stable); we essentially execute two separate queries, one over the data over which the PMV was constructed and another over the new/buffered data, and merge the results. At a later time, we can perform incremental maintenance on the PMV, by essentially building a new PMV over the new/buffered data, with the same grid definition as the existing PMV, and then merging each cell with its counterpart in the existing PMV. Determining when to trigger these incremental maintenance operations without increased complexity for the user is a key direction of future work.

Expanding Components. Since SageDB so far has focused primarily on physical design, good candidates for components to add next are ones that improve the logical

side, e.g., a learned query optimizer or a learned cost model. We believe the main challenge will be to keep these components "in sync." For example, the physical design optimization depends on the query optimizer (especially its ability to simulate PMVs and layouts) and cost model. If the optimizer or cost model changes due to retraining, then the physical design may be out-of-date, e.g., some pieces of the physical design will no longer be selected by the optimizer.

With more components, there is also further opportunity for model sharing. For example, when we optimize the replicated data layouts we might simultaneously build an updated learned model of the data distribution for the learned query optimizer; otherwise, an RL-based query optimizer might need to be retrained from scratch.

5.7 Related Work

Automatic database tuning. Modern data system have an increasing number of knobs and configuration options to be tuned by database administrators or by (semi-)automatic tools. There have been efforts to automatically tune a DBMSs' configuration since the early 2000s. Much of the previous work on automatic database tuning has focused on optimizing the physical design of the database [25], such as selecting indexes [5, 70], partitioning schemes [6, 32, 156], or materialized views [5]. Based on the method used to find the ideal configuration, the previous work can be divided into two categories: rule-based methods [35, 105] and ML-based methods [200, 50, 185, 119, 118, 155]. Cosine [23] focuses on self-designing key-value stores. Both approach performance optimization differently, with SageDB using learned components while Cosine essentially creates more knobs to tune. NoisePage [154] focuses on designing a self-optimizing database like SageDB by defining an objective function and action space. A centralized service learns to optimize the objective through the actions. NoisePage learns how to take standard actions in the database, such as adding/dropping indices, configuring knobs, and scaling hardware resources. Compared to instance-optimized components or systems, automatic database tuning has fewer degrees of freedom and is typically performed in a black-box manner.

Instance-optimized components. Further research has expanded the breadth and depth of instance-optimized components. More sophisticated learned indexes use multivariate data distributions to create multidimensional indexes [137, 48, 195]. There are now instance-optimized versions of bloom filters [132, 37, 182] and hash tables [167]. New use cases, from caching [115, 92] to query optimization [124, 126, 103] to scheduling [121], have leveraged learning to improve performance. SageDB aims to take this to the next step: where prior work designed components to adapt to the data and workload, SageDB intends to design an entire system with that capability.

Computation Reuse. PMVs can be considered a form of computation reuse, in which we pre-materialize certain results that will be used multiple times in the future. Other forms of computation reuse are multi-query optimization [169, 166] (which aims to find a globally optimal execution plan for a batch of queries), materialized views [94], data cubes [68], and sub-expression materialization [84]. There have also been various

works on *opportunistically* caching intermediate results that are already materialized during query execution, and reusing these intermediate results during future query execution [136, 81, 157, 110]. These techniques all assume that (sub)queries must be fully processed using pre-computed or cached results, whereas PMVs have the flexibility of *partially* answering the query, while the cheaper remainder query scans the base data. PMVs are orthogonal to the idea of partially maintaining the state of a traditional materialized view, such as in Noria [63].

Replication and Data Layouts While there have many works on instance-optimized data layouts [137, 48, 195, 45], the only other work to consider the combination of partial replication with instance-optimized data layouts is CopyRight [175]. However, their optimization algorithm makes assumptions that are specialized for grid-based data layouts for in-memory data over a single table, while SageDB handles multi-table disk-based datasets.

5.8 Conclusion

In this chapter, we presented a progress report on SageDB, a first instance-optimized data system, focused on analytics. SageDB incorporates two instance-optimized components into one system that exposes a simple interface to the user. While our prototype system already achieves impressive results, our aspirations for SageDB are far from complete. Our roadmap for future work includes implementing techniques to eliminate performance regressions, gracefully handling data changes, and incorporating further instance-optimized components. We hope that this report leads us a step closer towards making the vision for instance-optimized systems a reality.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this thesis, we formally defined instance-optimization as the process of co-designing (1) a mechanism with enough degrees of freedom to allow for fine-grained customization to any particular use case and (2) a policy, or optimization algorithm, for automatically specializing the mechanism to a specific use case, i.e., a well-defined dataset and workload.

We then presented new techniques for the instance-optimization of core components of a database's physical design: first, indexes are a fundamental component of any database that supports operational workloads with a mix of point lookups, short range queries, inserts, updates, and deletes. Learned Indexes [100] took the first step towards introducing instance-optimization to traditional database indexes, but it was critically limited by its lack of support for data modification operations. Therefore, we introduced ALEX (Chapter 2), an updatable learned index that dynamically adjusts both its models and its structure in the presence of data modifications, while maintaining its speed and space advantages over both non-learned indexes and the original Learned Index itself.

Second, storage layouts are a fundamental way in which analytic databases improve performance, by minimizing the amount of data that is read during query processing. We introduced two new instance-optimized storage layouts that address critical limitations that prevent existing instance-optimized data layouts from being effective for real-world use cases. First, Tsunami (Chapter 3) is designed for in-memory data and handles correlated data and skewed query workloads, which are the common case in practice. Second, MTO (Chapter 4) is designed for data on disk or on cloud storage and is the first instance-optimized data layout that handles workloads in which queries join multiple tables.

Finally, we examined through SageDB (Chapter 5) how an end-to-end instance-optimized database system can be built through the careful co-design of multiple instance-optimized components, unified through a global optimization algorithm. Overall, we hope that this thesis serves as a building block for the further exploration and development of impactful and practical instance-optimized database systems.

6.2 Future Work

While the work presented in this thesis demonstrates the opportunities and promise of instance-optimized database systems, certain limitations of existing instance-optimization techniques still need to be addressed in order for instance-optimization to be practical and usable in production systems. In particular, users want robustness guarantees to ensure that performance will not regress when using instance-optimized databases, especially robustness against dynamic changes in the dataset and workload. Overall, we believe that the instance-optimization of database systems will continue to be a rich and active field of research. Here, we highlight several important directions for future work.

Theoretical Results and Robustness. While instance-optimization has shown impressive empirical results on real-world datasets and workloads, widespread adoption of instance-optimization will also require theoretical guarantees and robustness. Some work has already been done [56], but there is still much more that can be done. In particular, robustness guarantees will require more accurate definitions of what defines a use case, in a similar vein as prior work on robust physical design auto-tuning [135] that required a rigorous framework for measuring uncertainty and workload similarity.

Dynamic Workloads and Datasets. Instance-optimization is defined in the context of a certain use case, which means that it is essential to detect when the use case has changed enough to merit a re-optimization. In fact, the process of re-optimization is itself an instance-optimization problem: there are many mechanisms for transitioning from one database configuration to another (e.g., shuffling data from one set of files to another set can be done with a Map-Reduce with a configurable number of stages), and we need policies to choose the best transition strategy. The overall policies for the instance-optimized database must not only consider the performance benefits of a certain configuration, but it must also take into account the (probabilistic) cost of a potential re-optimization. In the context of instance-optimized data layouts, MTO made a first foray into exploring how to more efficiently respond to workload shifts through partial layout reorganization instead of full layout reorganization, and self-organizing data containers [120] aim to fully automate the process of detecting and handling workload shifts.

Techniques for handling dynamic workloads and datasets can be either reactive or proactive. Reactive techniques aim to quickly detect changes after they have already started to occur, and to quickly adapt to those detected changes; this is the predominant technique used by existing instance-optimized physical design components. However, proactive techniques have the potential to be much more effective: such techniques predict how the data and workload will change in the future, and directly optimize for that future. For proactive techniques to be effective, we require more accurate forecasting of the future data and workload. We also need to explore multi-step planning techniques, e.g., instead of optimizing a single data layout for a single static dataset and workload, we plan a sequence of data layouts and times at which we transition from each layout to the next that are optimized for the anticipated future sequence of data updates and workload changes.

Generalized Global Physical Design Optimization. SageDB showed how to compose multiple instance-optimized physical design components under a global optimization algorithm. However, there are various other instance-optimized physical design components in the literature, such as learned membership filters [132, 182], learned frequency sketches [74], and instance-optimized data compression [78]. If implemented in the same system, all of these components would compete for the same resources (e.g., storage space and compute for creation and maintenance), and would furthermore have an affect on each other's performance.

It would be inefficient to completely re-design the global optimization algorithm each time we incorporate an additional instance-optimized component, and it is also unsatisfactory to incrementally modify the global optimization algorithm for each additional component as an afterthought. Ideally, we should have a generalized global physical design optimization algorithm that interacts with each specific component through a well-defined API. Under such a design, every new component can be incorporated into the global optimization by implementing this API. Then, the global optimization algorithm can treat each component as a black box and allocate resources accordingly, under some global objectives and constraints. The challenge is that, as we observed in SageDB, instance-optimized components are not independent. Therefore, the API must be carefully designed to expose some ability for components to share information with each other, e.g., some shared state which might be captured by a model.

A More Complete Instance-Optimized Database. SageDB is currently an instance-optimized database system that includes multiple instance-optimized components from the data storage tier. A remaining challenge is to integrate an instance-optimized data storage tier with an instance-optimized query processing tier. For example, we can incorporate the extensive work on learned query optimizers [124, 126], learned cardinality estimation [90, 196], and learned cost models [72]. As we alluded to in Section 5.6, we believe the main challenge will be to keep components from the data storage tier and the query processing tier "in sync." For example, physical design optimization depends on the query optimizer and cost model. If the optimizer or cost model changes due to re-optimization, then the physical design may be out-of-date, e.g., some pieces of the physical design will no longer be selected by the optimizer.

Online vs. Offline Optimization. Note that some of the instance-optimized techniques in this thesis used offline optimization algorithms that needed to be triggered at some point in time by a user (Tsunami, MTO, and SageDB), while one used an online algorithm that continuously modified the mechanisms in the background (ALEX). Online and offline algorithms each have their pros and cons. An online algorithm completely removes any need for user intervention, and can appear more seamless to the user. At the same time, users may want visibility and interpretability into how instance-optimization affects the performance and costs of their system, which is easier to understand when optimization is triggered at a specific point in time, as in the case with offline optimization, than if optimization is constantly performed in the background, as is the case with online optimization. In general, it is important to further explore and understand the tradeoffs between online and offline

instance-optimization techniques.

Beyond Relational Databases. Finally, this thesis has focused on relational database systems, but the ideas of instance-optimization are applicable to other types of database systems, such as NoSQL databases, graph databases, time-series databases, or even data lakes composed of data in diverse formats. This requires a new analysis of the most impactful components to instance-optimize. For example, the way in which a graph is physically stored can be optimized for the access patterns. An auto-tuner might decide between using a sparse or dense representation, but an instance-optimized graph database might be able to materialize any design on the sparsity spectrum.

Appendix A

Supplementary Material for ALEX

A.1 Extended Bulk Loading Evaluation

In this section, we provide an extended version of Section 2.5.2, which evaluates the speed of ALEX's bulk loading mechanism against other indexes. ALEX uses two optimizations for bulk loading—approximate model computation (AMC) and approximate cost computation (ACC)—which we explain in more detail below.

For each index, we bulk load 100 million keys from each of the four datasets from Section 2.5. This includes the time used to sort the 100 million keys. Fig. A-1, which is a more detailed version of Fig. 2-11a, shows that on average, ALEX with no optimizations takes 3.6× more time to bulk load than B+Tree, which is the fastest index to bulk load. However, with the AMC optimization, ALEX takes 2.6× more time on average than B+Tree. With both optimizations, ALEX only takes 50% more than time on average than B+Tree, and in the worst case is only 2× slower than B+Tree. The results for ALEX in Section 2.5.2 use both optimizations. On the YCSB dataset, ALEX's structure is very simple (Table 2.2), and therefore is very efficient to bulk load even when unoptimized; in fact, ALEX's large node sizes allow ALEX to bulk load faster than other indexes due to the benefits of locality.

Fig. A-2 shows the impact of the two optimizations on the throughput of running a read-heavy or write-heavy workload on ALEX after bulk loading. The AMC optimization has negligible impact on throughput for all datasets and for both workloads. Adding the ACC optimization has negligible impact on the read-heavy workload, but decreases throughput by up to 9.6% on the write-heavy workload; we provide explanation for this behavior below.

Based on these results, we conclude that the AMC optimization should always be used to improve bulk loading performance, whereas the ACC optimization might cause a slight decrease in throughput performance and therefore should be used only if faster bulk loading is required. We now explain the two optimizations in more detail.

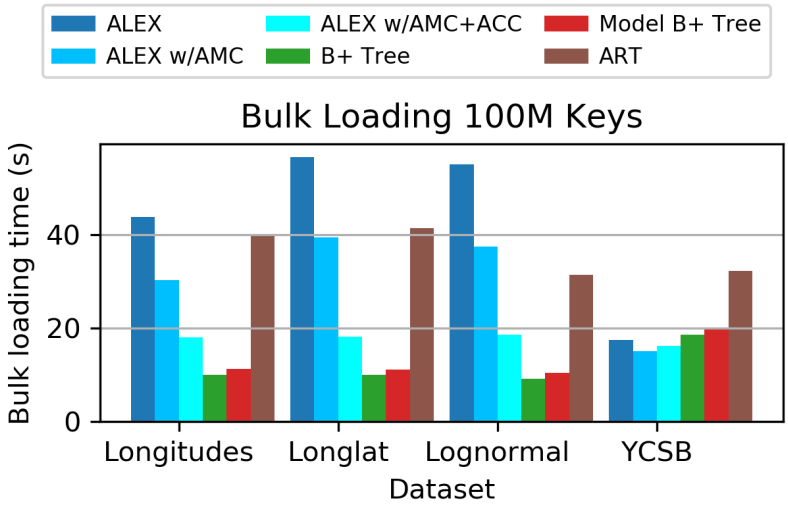

Figure A-1: With both optimizations (AMC and ACC), ALEX only takes 50% more than time than B+Tree to bulk load when averaged across four datasets.

A.2 Approximate Model Computation

We perform approximate model computation (AMC) efficiently while achieving accuracy through *progressive systematic sampling*. Given a data node of sorted keys, we perform *systematic sampling* (i.e., sampling every nth key) to obtain a small sample of keys, and compute a linear regression model using that sample. We then repeatedly double the sample size and recompute the linear model using the larger sample. When the relative change in the model parameters (i.e., the slope and intercept) both change by less than 1% from one sample to the next, we terminate the process. In our experience, this 1% threshold strikes a balance between achieving accuracy and using small sample sizes, and did not need to be tuned.

Note that by using systematic sampling, all keys in the sample used to compute the current model will also appear in all subsequent samples. Therefore, each linear model can be computed *progressively* starting from the existing model computed from the previous sample, instead of from scratch. No redundant work is done, and even in the worst case, AMC will take no more time than computing one linear model from all keys (if we ignore minor overheads and the effects of locality).

A.3 Approximate Cost Computation

We also perform approximate intra-node cost computation (ACC) for a data node of sorted keys through progressive systematic sampling. However, ACC differs from AMC in two ways. First, the cost for a data node must be computed from scratch for each sample; the cost depends on where keys are placed within a Gapped Array, which itself depends on which keys are present in the sample. Second, and more importantly, the cost of a data node naturally increases with the number of keys in the node. This

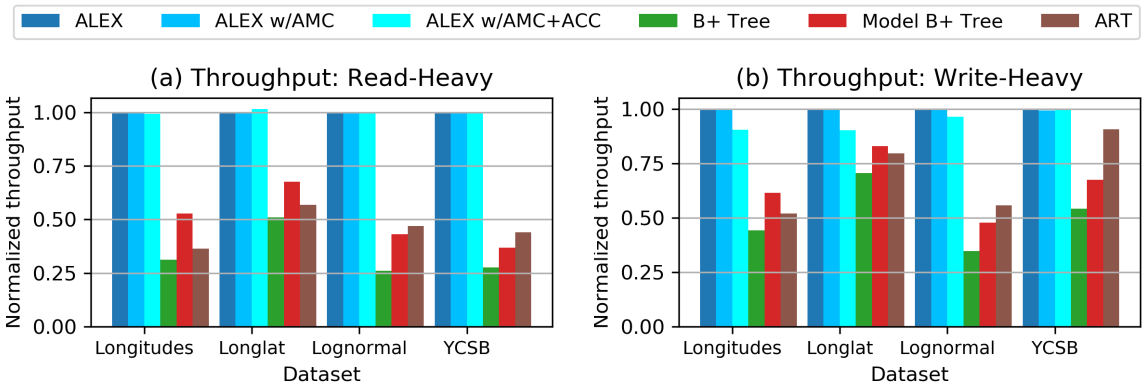

Figure A-2: Using the AMC optimization when bulk loading does not cause any noticeable change in ALEX performance, but ACC can cause a slight decrease in throughput for write-heavy workloads.

makes ACC an extrapolation problem (i.e., use the cost of a small sample to predict the cost of the entire data node), whereas AMC is an estimation problem (i.e., use a small sample to directly estimate the model parameters).

ACC repeatedly doubles its sample size. Let the latest three samples be s_1, which is half the size of s_2, which is half the size of s_3. Let the costs computed from these samples be c_1, c_2, and c_3, respectively. We use the c_1 and c_2 to perform a linear extrapolation to predict c_3. If this prediction is accurate (i.e., if relative error with the true c_3 is within 20%), then we use c_2 and c_3 to perform a linear extrapolation to predict the cost of the entire data node, and we terminate the process.[1] Otherwise, we continue doubling the sample size. The intuition behind this process is that we want to verify the accuracy of extrapolation using small samples before extrapolating to the entire data node. We allow a higher relative error than for AMC because the extrapolation process is inherently imprecise, since it is impossible to accurately predict the cost using a sample without a priori knowledge of the data distribution.

We can now explain why Fig. A-2 shows that adding the ACC optimization decreases throughput on the write-heavy workload by up to 9.6%. It is because the average number of shifts per insert, which is one component of the intra-node cost, is difficult to estimate accurately. Therefore, if ACC underestimates the component of cost related to shifts, the bulk loaded ALEX structure may be inefficient for inserts (e.g., an insert that requires more shifts than expected can be very slow). The intra-node cost is more difficult to approximate accurately for the longitudes and longlat datasets, which is why the decrease in throughput is most noticeable for those two datasets. However, note that over time, the dynamic nature of ALEX will eventually correct for incorrectly estimated costs, so throughput performance in the long run will be independent of the bulk loading mechanism.

[1] In reality, we do not predict the cost directly, but rather each component of the cost (search iterations per lookup and shifts per insert) independently. This is because the expected extrapolation behavior differs: iterations per lookup grows logarithmically with sample size, whereas shifts per insert grows linearly with sample size.

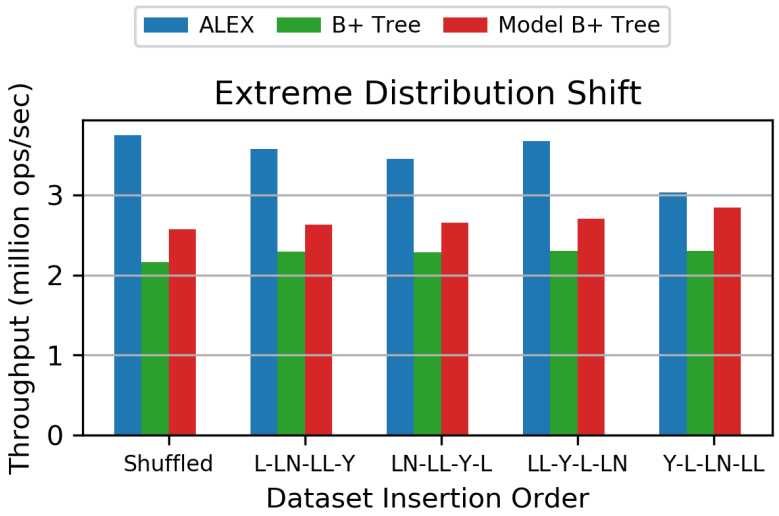

Figure A-3: ALEX maintains high performance under radically changing key distribution, although performance does differ slightly depending on the distribution used for bulk loading.

A.4 Extreme Distribution Shift Evaluation

In order to evaluate the performance of ALEX under a radically changing key distribution, we combine the four datasets from Section 2.5 into one dataset by randomly selecting 50 million keys from each of the four datasets in order to create one combined dataset with 200 million keys. We scaled keys from each dataset to fit in the same domain. Note that we would not typically expect a single table to contain keys from four independent distributions. Therefore, this complex combined dataset is an extreme stress test for the adaptibility of ALEX.

We run a write-heavy workload (50% point lookups and 50% inserts) over the combined dataset, but we vary the order in which keys are bulk loaded and inserted. For all variants, we bulk load using 50 million keys and run the write-heavy workload until the remaining 150 million keys are all inserted. We create four variants that represent distribution shift; each variant bulk loads using the 50 million keys selected from one of the four original datasets, then gradually inserts keys from the other three original datasets, in order. For example, the "L-LN-LL-Y" variant bulk loads using the 50 million keys selected from the longitudes (L) dataset, then runs the write-heavy workload by inserting the 50 million keys from the lognormal (LN) dataset, then the longlat (LL) dataset, and finally the YCSB (Y) dataset. For reference, we also include a variant in which all 200 million keys are shuffled, so that no key distribution shift is observed.

Fig. A-3 provides three insights. First, on the workload that represents no distribution shift ("Shuffled"), ALEX continues to outperform other indexes. It is interesting to note that the throughput of ALEX on the combined dataset is between the throughputs achieved on each dataset individually (Fig. 2-9c): higher than for longlat, and

lower than for the other three datasets. Second, ALEX achieves lower throughput in the four variants that represent distribution shift than without distribution shift, but still outperforms other indexes. This result aligns with the intuition that ALEX must spend extra time restructuring itself to adapt to the changing key distribution. Third, the throughput differs based on which dataset's keys are used to bulk load ALEX. When bulk loading using keys from a complex key distribution, such as longlat, ALEX achieves throughput similar to the variant with no distribution shift; on the other hand, when bulk loading using keys from a simple key distribution, such as YCSB, ALEX throughput suffers. This is because when bulk loading with a simple key distribution, the bulk loaded structure of ALEX will be shallow, with few nodes (Table 2.2). When the subsequently inserted keys come from a much more complex key distribution, ALEX must quickly adapt its structure to be deeper and have more nodes, which can incur significant overhead. On the other hand, when bulk loading with a complex key distribution, the bulk loaded structure is already deep, with many nodes, and so can more readily adapt to changes in the key distribution without too much overhead.

To allow ALEX to more quickly adapt the RMI structure to radically changing key distributions: (1) we check data nodes periodically for cost deviation instead of only when the data node is full, and (2) if the number of shifts per insert in a data node is extremely high, we force the data node to split (as opposed to expanding and retraining the model). When no distribution shift occurs, these two checks have negligible impact on performance, because checking for cost deviation has minimal overhead and cost deviation occurs infrequently (Table 2.3). By default, we check for cost deviation for every 64 inserts into that data node, and over 100 shifts per insert is considered extremely high.

A.5 Extended Range Query Evaluation

Varying Range Query Scan Length

We extend the experiment from Fig. 2-10 to all four datasets. Fig. A-4 shows that across all datasets, ALEX maintains its advantage over fixed-page-size B+Tree, and re-tuning the B+Tree page size can lead to better range query performance but will decrease performance on point lookups and inserts. For both ALEX and B+Tree, performance on YCSB is slower than for the other three datasets because YCSB has a larger payload size, which worsens scan locality.

A.5.1 Mixed Workload Evaluation

We evaluate a mixed workload with 5% inserts, 85% point lookups, and 10% range queries with a maximum scan length of 100. The remainder of the experimental setup is the same as in Section 2.5.1. Fig. A-5 shows that ALEX maintains its performance advantage over other indexes. The ART implementation from [34] does not support range queries.

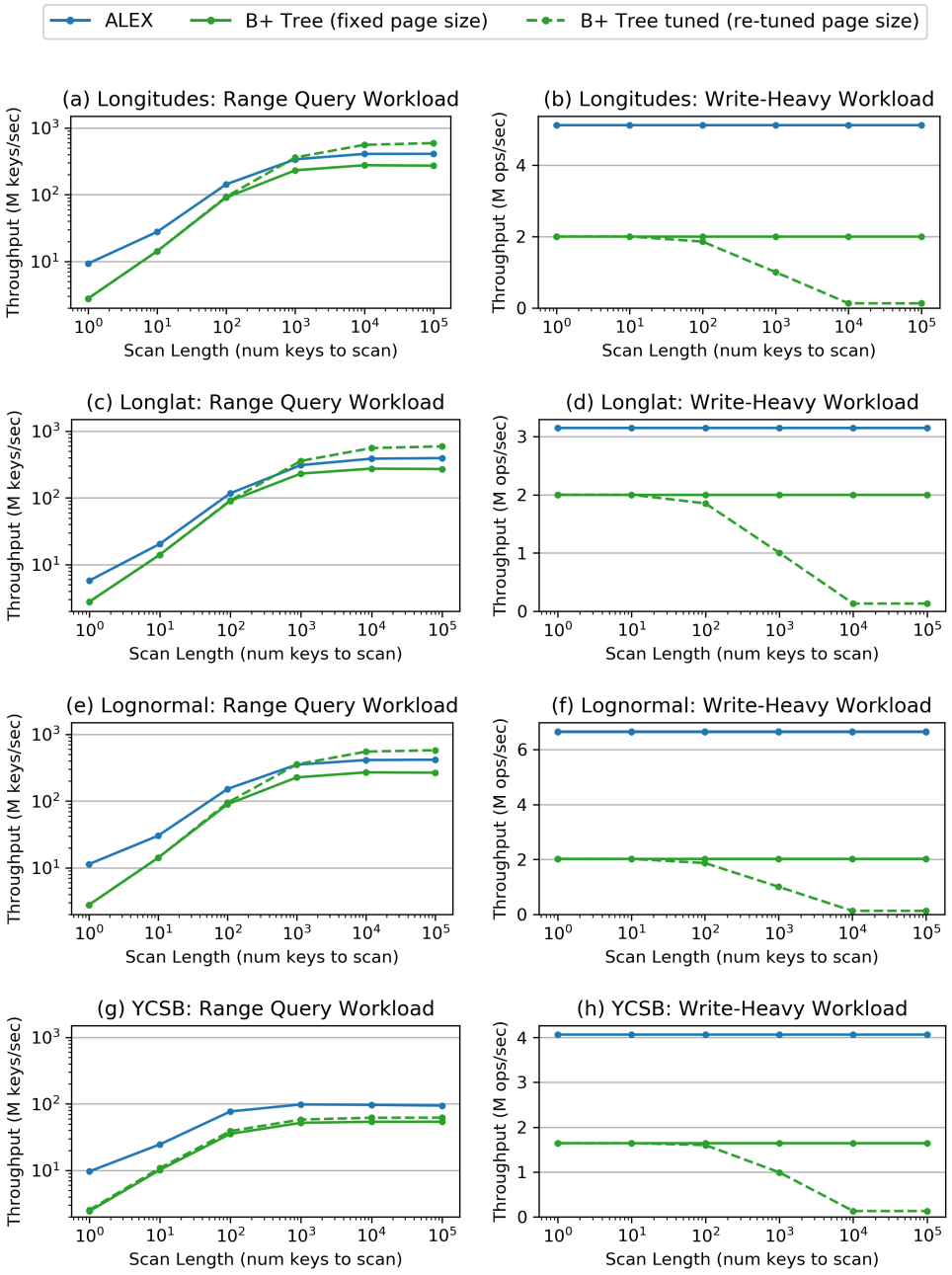

Figure A-4: Across all datasets, ALEX maintains an advantage over fixed-page-size B+Tree even for longer range scans.

A.6 Drilldown into Cost Computation

In this section, we first provide more details about the cost model introduced in Section 2.3.3. We then evaluate the performance of computing costs using cost models.

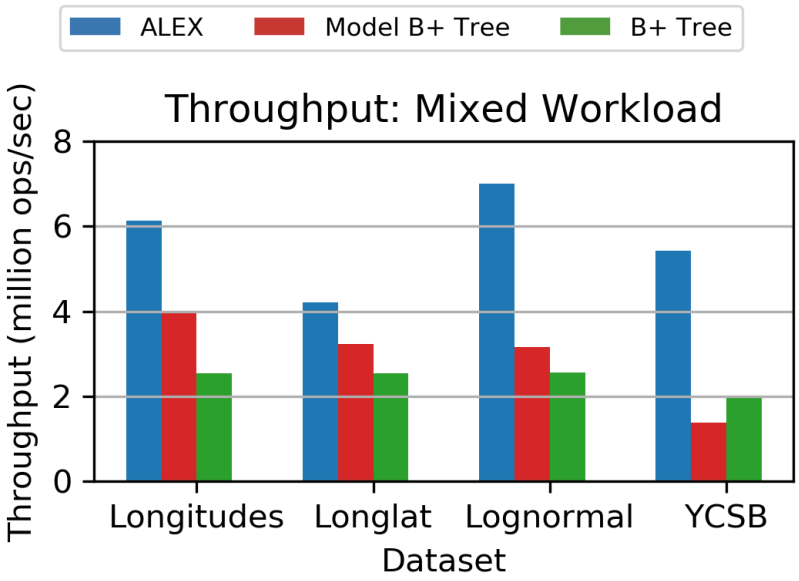

Figure A-5: ALEX maintains high performance under a mixed workload with 5% inserts, 85% point lookups, and 10% short range queries.

A.6.1 Cost Model Details

We formally define the cost model using the terms in Table A.1. At a high level, the intra-node cost of a data node represents the average time to perform an operation (i.e., a point lookup or insert) on that data node, and the TraverseToLeaf cost of a data node represents the time for traversing from the root node to the data node.

For a given data node $N \in \mathcal{D}$, the intra-node cost $C_I(N)$ is defined as

$$C_I(N) = w_s S(N) + w_i I(N) F(N) \tag{A.1}$$

Both lookups and inserts must perform an exponential search, whereas only inserts must perform shifts. This is why $I(N)$ is weighted by $F(N)$.

For a given data node $N \in \mathcal{D}$, the TraverseToLeaf cost $C_T(N)$ of traversing from the root node to N is defined as

$$C_T(N) = w_d D(N) + w_b B(\mathcal{A}) \tag{A.2}$$

The depth of N is the number of pointer chases needed to reach the data node. In our cost model, every traverse to leaf has a fixed cost that is caused by the total size of the ALEX RMI, because larger RMI causes worse cache locality.

For an instantiation of ALEX $\mathcal{A}$, the cost of $\mathcal{A}$ represents the average time to perform a query (i.e., a point lookup or insert) starting from the root node, and is defined as

$$C(\mathcal{A}) = \frac{\sum_{N \in \mathcal{D}} (C_I(N) + C_T(N)) K(N)}{\sum_{N \in \mathcal{D}} K(N)} \tag{A.3}$$

In other words, the cost of $\mathcal{A}$ is the sum of the intra-node cost and TraverseToLeaf cost

Table A.1: Terms used to describe the cost model

Term	Description
$\mathcal{A}$	An instantiation of ALEX
$\mathcal{N}$	Set of all nodes in $\mathcal{A}$
$\mathcal{D}$	Set of data nodes in $\mathcal{A}$. This means that $\mathcal{D} \subseteq \mathcal{N}$
$S(N)$	Average number of exponential search iterations for a lookup in $N \in \mathcal{D}$
$I(N)$	Average number of shifts for an insert into $N \in \mathcal{D}$
$K(N)$	Number of keys in $N \in \mathcal{D}$
$F(N)$	Fraction of operations that are inserts (as opposed to lookups) in $N \in \mathcal{D}$
$C_I(N)$	Intra-node cost of $N \in \mathcal{D}$
$C_T(N)$	TraverseToLeaf cost of $N \in \mathcal{D}$
$D(N)$	Depth of $N \in \mathcal{N}$ (root node has depth 0)
$B(\mathcal{A})$	Total size in bytes of all nodes in $\mathcal{A}$
w_s, w_i, w_d, w_b	Fixed pre-defined weight parameters

of each data node, normalized by how many keys are contained in the data node. We normalize because each data node does not contribute equally to average query time. For example, a data node that has high intra-node cost but is rarely queried might not have as much impact on average query time as a data node with lower intra-node cost that is frequently queried. We use the number of keys in each data node as a proxy for its impact on the average query time. An alternative is to normalize using the true query access frequency of each data node.

The weight parameters w_s, w_i, w_d, w_b do not need to be tuned for each dataset or workload, because they represent fixed quantities. For our evaluation, we set $w_s = 10$,$w_i = 1$,$w_d = 10$, and $w_b = 10^{-6}$. In terms of impact on throughput performance, these weights intuitively mean that each exponential search iteration takes 10 ns, each shift takes 1 ns, each pointer chase to traverse down one level of the RMI takes 10 ns, and each MB of total size contributes a slowdown of 1 ns due to worse cache locality. As a side effect, w_b acts as a regularizer to prevent the RMI from growing unnecessarily large. We found that our simple cost model performed well throughout our evaluation. However, it may still be beneficial to formulate a more complex cost model that more accurately reflects true runtime; this is left as future work.

A.6.2 Cost Computation Performance

The cost of the entire RMI, $C(\mathcal{A})$, is never explicitly computed. Instead, all decisions based on cost are made locally. This is possible due to the linearity of the cost model. For example, when deciding between expanding a data node and splitting the data node in two, we compare the *incremental* impact on $C(\mathcal{A})$ between the two options. This only involves computing the intra-node cost of the expanded data node and each of the two split data nodes; the intra-node costs of all other data nodes in the RMI

Table A.2: Fraction of time spent on cost computation

	longitudes	**longlat**	**lognormal**	**YCSB**
Read-Only	0	0	0	0
Read-Heavy	0.000271	0.000214	0.000617	0
Write-Heavy	0.00142	0.00901	0.00452	0.116
Write-Only	0.0270	0.0732	0.0237	0.149

remain the same.

Cost computation occurs at two points during ALEX operation (Section 2.3.3): (1) when a data node becomes full, the expected intra-node cost is compared to the empirical intra-node cost to check for cost deviation. This comparison has very low performance overhead because the empirical values of $S(N)$ and $I(N)$ are maintained by the data node, so computing the empirical intra-node cost merely involves three multiplications and an addition. (2) If cost deviation is detected, ALEX must make a cost-based decision about how to adjust the RMI structure. This involves computing the expected intra-node cost of candidate data nodes which may be created as a result of adjusting the RMI structure. Since the candidate data nodes do not yet exist, we must compute the expected $S(N)$ and $I(N)$, which involves implicitly building the candidate data node. The majority of time spent on cost-based decision making is spent on computing expected $S(N)$ and $S(I)$.

Table A.2 shows the fraction of overall workload time spent on computing costs and making cost-based decisions. On the read-only workload, no time is spent on cost-based decision because nodes never become full. As the fraction of writes increases, an increasing fraction of time is spent on cost computation because nodes become full more frequently. However, even on the write-only workload, cost computation takes up a small fraction of overall time spent on the workload. YCSB sees the highest fraction of time spent on cost computation, due to two factors: data nodes are larger, so computing $S(N)$ and $I(N)$ for larger candidate nodes takes more time, and lookups and inserts on YCSB are efficient, so data nodes become full more quickly. Longlat sees the next highest fraction of time spent on cost computation, which is due to the high frequency with which data nodes become full (Table 2.3).

A.7 Comparison of Gapped Array and PMA

The Gapped Array structure introduced in Section 2.2.2 has some similarities to an existing data structure known as the Packed Memory Array (PMA) [16]. In this section, we first describe the PMA, and then we describe why we choose to not use the PMA within ALEX.

Like the Gapped Array, PMA is an array with gaps. Unlike the Gapped Array, PMA is designed to uniformly space its gaps between elements and to maintain this property as new elements are inserted. The PMA achieves this goal by rebalancing local portions of the array when the gaps are no longer uniformly spaced. Under random inserts from a static distribution, the PMA can insert elements in $O(\log n)$

time, which is the same as the Gapped Array. However, when inserts do not come from a static distribution, the PMA can guarantee worst-case insertion in $O(\log^2 n)$ time, which is better than the worst case of the Gappd Array, which is $O(n)$ time.

We now describe the PMA more concretely; more details can be found in [16]. The PMA is an array whose size is always a power of 2. The PMA divides itself into equally spaced segments, and the number of segments is also a power of 2. The PMA builds an implicit binary tree on top of the array, where each segment is a leaf node, each inner node represents the region of the array covered by its two children, and the root node represents the entire array. The PMA places density bounds on each node of this implicit binary tree, where the density bound determines the maximum ratio of elements to positions in the region of the array represented by the node. The nodes nearer the leaves will have higher density bounds, and the nodes nearer the root will have lower density bounds. The density bounds guarantee that no region of the array will become too packed. If an insertion into a segment will violate the segment's density bounds, then we can find some local region of the array and uniformly redistribute all elements within this region, such that after the redistribution, none of the density bounds are violated. As the array becomes more full, ultimately no local redistribution can avoid violating density bounds. At this point, the PMA expands by doubling in size and inserting all elements uniformly spaced in the expanded array.

We do not use the PMA as the underlying storage structure for ALEX data nodes because the PMA negates the benefits of model-based inserts, which is critical for search performance. For example, when rebalancing a local portion of the array, the PMA spreads the keys in the local region over more space, which worsens search performance because the keys are moved further away from their predicted location. Furthermore, the main benefit of PMA—efficient inserts for non-static or complex key distributions—is already achieved by ALEX through the adaptive RMI structure. In our evaluation, we found that ALEX using data nodes built on Gapped Arrays consistently outperformed data nodes built on PMA.

A.8 Analysis of Model-based Search

Model-based inserts try to place keys in Gapped Array in their predicted positions. We analyze the trade-off between Gapped Array space usage and search performance in terms of c, the ratio of Gapped Array slots to number of actual keys. Assume the keys in the data node are $x_1 < x_2 < \cdots < x_n$, and the linear model before rounding is $y = ax + b$ when $c = 1$, i.e., when no extra space is allocated. Define $\delta_i = x_{i+1} - x_i, \Delta_i = x_{i+2} - x_i$. We first present a condition under which all the keys in that data node are placed in the predicted location, i.e., search for all keys are direct hits.

Theorem 2. *When $c \geq \frac{1}{a \min_{i=1}^{n-1} \delta_i}$, every key in the data node is placed in the predicted location exactly.*

Proof. Consider two keys in the leaf node x_i and $x_j, i \neq j$. The predicted locations before rounding are y_i and y_j, respectively. When $|y_i - y_j| \geq 1$, we know that the

rounded locations $\lfloor y_i \rfloor$ and $\lfloor y_j \rfloor$ cannot be equal. Under the linear model $y = c(ax+b)$, we can write the condition as:

$$|y_i - y_j| = |ca(x_i - x_j)| \geq 1 \tag{A.4}$$

If this condition is true for all the pairs $(i, j), i \neq j$, then all the keys will have a unique predicted location. For the condition Eq. (A.4) to be true for all $i \neq j$, it suffices to have:

$$\min_{i=1}^{n-1} ca(x_{i+1} - x_i) \geq 1 \tag{A.5}$$

which is equivalent to $c \geq \frac{1}{a \min_{i=1}^{n-1} \delta_i}$. □

We now understand that $c = 1$ corresponds to the optimal space, and $c \geq \frac{1}{a \min_{i=1}^{n-1} \delta_i} = c_{max}$ corresponds to the optimal search time (ignoring the effect of cache misses). We now bound the number of keys with direct hits when $c < c_{max}$.

Theorem 3. *The number of keys placed in the predicted location is no larger than* $2 + \left|\{1 \leq i \leq n-2 | \Delta_i > \frac{1}{ca}\}\right|$, *where* $\left|\{1 \leq i \leq n-2 | \Delta_i > \frac{1}{ca}\}\right|$ *is the number of* Δ_i*'s larger than* $\frac{1}{ca}$.

Proof. We define a mapping $f : [n-2] \to [n]$, where $f(i)$ is defined recursively according to the following cases:

Case (1): $y_{i+2} - y_i > 1$. Let $f(i) = 1$. Case (2): $y_{i+2} - y_i \leq 1, \lfloor y_{i+1} \rfloor = \lfloor y_i \rfloor, f(i-1) \leq i$ or $i = 1$. Let $f(i) = i + 1$. Case (3): Neither case (1) or (2) is true. Let $f(i) = i + 2$.

We prove that $\forall 1 \leq i < j \leq n-2$, if $f(i) > 1, f(j) > 1$, then $i + 1 \leq f(i) \leq i + 2, j + 1 \leq f(j) \leq j + 2$, and $f(i) < f(j)$.

First, when $f(i) > 1, f(j) > 1$, we know that case (1) is false for both i and j. So $f(i)$ is either $i + 1$ or $i + 2$, and $f(j)$ is either $j + 1$ or $j + 2$.

Second, if $i + 1 < j$, then $f(i) \leq i + 2 < j + 1 \leq f(j)$. So we only need to prove $f(i) < f(j)$ when $i + 1 = j$. Now consider the only two possible values for $f(j)$, $j + 1$ and $j + 2$, when $i + 1 = j$. If $f(j) = j + 1 = i + 2$, by definition we know that case (2) is true for $f(j)$. That means $f(j-1) = j$ or 1. But we already know $f(j-1) = f(i) > 1$. So $f(i) = f(j-1) = j = i + 1 < i + 2 = f(j)$. If $f(j) = j + 2$, then $f(i) \leq i + 2 < j + 2 = f(j)$.

So far, we have proved that $f(i)$ is unique when $f(i) > 1$. Now we prove that the key $x_{f(i)}$ is not placed in $\lfloor y_{f(i)} \rfloor$ when $f(i) > 1$, i.e., either case (2) or case (3) is true for $f(i)$. In both cases, $y_{i+2} - y_i \leq 1$, and the rounded integers $\lfloor y_{i+2} \rfloor$ and $\lfloor y_i \rfloor$ must be either equal or adjacent: $\lfloor y_{i+2} \rfloor - \lfloor y_i \rfloor \leq 1$. That means $\lfloor y_{i+1} \rfloor$ must be equal to either $\lfloor y_{i+2} \rfloor$ or $\lfloor y_i \rfloor$.

We prove by mathematical induction. For the minimal i s.t. $f(i) > 1$, if case (2) is true, $\lfloor y_{i+1} \rfloor = \lfloor y_i \rfloor$. That means x_{i+1} cannot be placed at $\lfloor y_{i+1} \rfloor$ because that location is already occupied before x_{i+1} is inserted. And $f(i) = i + 1$ by definition. If case (2) is false, since we already know $y_{i+2} - y_i \leq 1$, $f(i-1) = 1$ or $i = 1$, it follows that

$\lfloor y_{i+1} \rfloor > \lfloor y_i \rfloor$. That implies $\lfloor y_{i+1} \rfloor = \lfloor y_{i+2} \rfloor$. So x_{i+2} cannot be placed at $\lfloor y_{i+2} \rfloor$. And $f(i) = i + 2$ because case (3) happens.

Given that the key $x_{f(i-1)}$ is not placed at $\lfloor y_{f(i-1)} \rfloor$ when $f(i-1) > 1$, we now prove it is also true for i. The proof for case (2) is the same as above. If case (2) is false, and $\lfloor y_{i+1} \rfloor > \lfloor y_i \rfloor$, the proof is also the same as above. The remaining possibility of case (3) is that $\lfloor y_{i+1} \rfloor = \lfloor y_i \rfloor$, and $f(i-1) = i + 1$. The inductive hypothesis states that x_{i+1} is not placed at $\lfloor y_{i+1} \rfloor$. That means x_{i+1} is placed at a location equal or larger than $\lfloor y_{i+1} \rfloor + 1 = \lfloor y_i \rfloor + 1$. But we also know that $\lfloor y_{i+2} \rfloor \leq \lfloor y_i \rfloor + 1$. So x_{i+2} cannot be placed at $\lfloor y_{i+2} \rfloor$ which is not on the right of x_{i+1}'s location. Since case (3) is false, $f(i) = i + 2$.

By induction, we show that when $f(i) > 1$, the key $x_{f(i)}$ cannot be placed at $\lfloor y_{f(i)} \rfloor$. That means when we look up $x_{f(i)}$, we cannot directly hit it from the model prediction. Since we also proved that $f(i)$ has a unique value when $f(i) > 1$, the number of misses from the model prediction is at least the size of $S = \{i \in [n-2] | f(i) > 1\}$. By the definition of $f(i)$, $S = \{i \in [n-2] | y_{i+2} - y_i \leq 1\}$. Therefore, the number of direct hits by the model is at most $n - |S| = 2 + |\{i \in [n-2] | y_{i+2} - y_i > 1\}| = 2 + \left|\{1 \leq i \leq n-2 | \Delta_i \geq \frac{1}{ca}\}\right|$.

□

This result presents an upper bound on the number of direct hits from the model, which is positively correlated with c. This upper bound also applies to the Learned Index, which has $c = 1$. This explains why the Gapped Array has the potential to dramatically decrease the search time. Similarly, we can lower bound the number of direct hits.

Theorem 4. *The number of keys placed in the predicted location is no smaller than $l + 1$, where l is the largest integer such that $\forall 1 \leq i \leq l, \delta_i \geq \frac{1}{ca}$, i.e., the number of consecutive δ_i's from the beginning equal or larger than $\frac{1}{ca}$.*

The proof is not hard based on the ideas from the previous two proofs.

Appendix B

Supplementary Material for SageDB

B.1 Data Schemas and Workloads

Here, we define the schemas for the three datasets by displaying their `CREATE TABLE` commands. We also define the three workloads by displaying the prepared statements. All of these use SageDB's SQL dialect, which is similar to but not entirely the same as any commercial SQL dialect.

B.1.1 Gaming

Schema

```
create table dim1 (
    d1_label text,
    d1_id int64 UNIQUE
);
create table dim2 (
    d2_type text,
    d2_duration int64,
    d2_label text,
    d2_d1_id int64,
    d2_id int64 UNIQUE
);
create table dim3 (
    d3_label text,
    d3_joined int64,
    d3_loc text,
    d3_p1 float64,
    d3_p2 float64,
    d3_p3 float64,
    d3_p4 float64,
    d3_p5 float64,
    d3_id int64 UNIQUE
```

```
);
create table fact (
    f_time INT64,
    f_d3_id INT64,
    f_d1_id INT64,
    f_amt INT64,
    f_type TEXT,
    f_p1 INT64,
    f_p2 INT64,
    f_p3 INT64,
    f_p4 INT64,
    f_p5 INT64,
    f_p6 INT64,
    f_p7 INT64,
    f_p8 float64,
    f_p9 float64,
    f_p10 float64,
    f_p11 float64,
    f_p12 float64,
    f_id int64 UNIQUE
);
create table attrib (
    attrib_f_id int64,
    attrib_d2_id int64,
    attrib_share float64
);
```

Workload

Q1: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND f_p1 < ?:INT64 AND f_p8 < ?:FLOAT64 GROUP BY d1_label ORDER BY cnt;`

Q2: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND f_p2 < ?:INT64 AND f_p9 < ?:FLOAT64 GROUP BY d1_label ORDER BY cnt;`

Q3: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND f_p3 < ?:INT64 AND f_p10 < ?:FLOAT64 GROUP BY d1_label ORDER BY cnt;`

Q4: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND (f_p4 < ?:INT64 OR f_p5 < ?:INT64) AND (f_p6 < ?:INT64 OR f_p7 < ?:INT64) AND (f_p11 < ?:FLOAT64 OR f_p12 < ?:FLOAT64) GROUP BY d1_label ORDER BY cnt;`

Q5: `select d3_loc, sum(f_amt) as total from fact, dim3 where d3_id = f_d3_id and f_type=?:TEXT group by d3_loc order by total desc limit 20;`

Q6: `Select d2_label, sum(attrib_share * f_amt) as total from attrib, fact, dim2 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_amt > ?:INT64 group by d2_label order by total desc;`

Q7: `Select d2_type, sum(attrib_share * f_amt) as total from fact, attrib, dim2 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_amt > ?:INT64 group by d2_type order by total desc;`

Q8: `Select d2_label, d2_type, sum(attrib_share * f_amt) as total from fact, attrib, dim2, dim3 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_d3_id = d3_id and d3_loc IN (?:TEXT, ?:TEXT, ?:TEXT, ?:TEXT, ?:TEXT, ?:TEXT) group by d2_label, d2_type order by total desc;`

Q9: `Select d3_loc, sum(f_amt) as total from fact, dim3 where f_d3_id = d3_id and (d3_p1 > ?:FLOAT64 or d3_p2 > ?:FLOAT64) group by d3_loc order by total desc;`

Q10: `Select d3_loc, sum(f_amt) as total from fact, dim3 where f_d3_id = d3_id and (d3_p3 > ?:FLOAT64 or d3_p4 > ?:FLOAT64) and d3_p5 > ?:FLOAT64 group by d3_loc order by total desc;`

Q11: `Select d2_type, sum(attrib_share * f_amt) as total from fact, attrib, dim2 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_amt > ?:INT64 and attrib_share > 0.10 and f_p4 - 5500 > f_p7 group by d2_type order by total desc;`

Q12: `Select d3_loc, sum(f_p9) from fact, dim3 where f_d3_id = d3_id and (f_p2 = ?:INT64 or f_p4 = ?:INT64) group by d3_loc order by d3_loc;`

Q13: `Select d3_loc, sum(f_p9) from fact, dim3 where f_d3_id = d3_id and f_p2 > ?:INT64 and f_p2 < ?:INT64 and f_p4 > ?:INT64 and f_p4 < ?:INT64 group by d3_loc order by d3_loc;`

B.1.2 Stack Overflow

Schema

```
create table stack_overflow (
  id UINT64,
  site_name TEXT,
  post_date DATE,
  poster_name TEXT,
  poster_reputation INT32,
  poster_join_date DATE,
```

```
  score INT32,
  view_count UINT64,
  favorite_count UINT64,
  answered UINT8,
  highest_score_answer INT32,
  comment_count UINT32,
  comment_max_score INT32,
  tag_count UINT32,
  tag_top25 UINT8,
  tag_top20 UINT8,
  tag_top15 UINT8,
  tag_top10 UINT8,
  tag_top5 UINT8,
  tag_rust UINT8,
  tag_cpp UINT8,
  tag_gpu UINT8
)
```

Workload

Q1: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE answered = 1 AND comment_count <= ?:UINT32 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q2: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE answered = 1 AND score >= ?:INT32 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q3: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE highest_score_answer >= score AND view_count >= ?:UINT64 AND comment_max_score >= ?:INT32 AND answered = 1 AND comment_count >= 0 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q4: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE answered = 0 AND comment_max_score >= ?:INT32 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q5: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE view_count >= ?:UINT64 AND comment_count >= ?:UINT32 GROUP BY EXTRACT(YEAR FROM post_date);`

Q6: `SELECT poster_name, COUNT(*) FROM denorm_so WHERE tag_rust = 1 AND poster_join_date <= ?:FLOAT64 AND view_count >= ?:UINT64 GROUP BY poster_name;`

Q7: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE favorite_count <= ?:UINT64 AND post_date >= ?:FLOAT64 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q8: `SELECT COUNT(*) FROM denorm_so WHERE poster_reputation >= ?:INT32 AND score >= ?:INT32 AND tag_top5 = 1;`

Q9: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE score >= ?:INT32 AND favorite_count >= ?:UINT64 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q10: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE score <= ?:INT32 AND comment_count <= ?:UINT32 GROUP BY EXTRACT(YEAR FROM post_date);`

Q11: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE answered = 0 AND score >= ?:INT32 AND tag_count >= 4 GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q12: `SELECT COUNT(*) FROM denorm_so WHERE answered = 1 AND post_date >= ?:FLOAT64;`

Q13: `SELECT COUNT(*) FROM denorm_so WHERE view_count >= ?:UINT64 AND (tag_rust = ?:UINT8 OR tag_cpp = ?:UINT8 OR tag_gpu = ?:UINT8);`

B.1.3 TPC-H

Schema

We use the same TPC-H schema in the official specification.

Workload

Q1: `select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice as sum_base_price, sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from lineitem where l_shipdate <= ?:DATE group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;`

Q2: `select s_name, sum(s_acctbal) as balance from part, supplier, partsupp, nation, region where part.p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and p_size = ?:INT32 and region.r_name = ?:TEXT and ps_supplycost = (select min(ps_supplycost)`

```
from partsupp, supplier, nation, region where p_partkey = ps_partkey and
s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey
and region.r_name = ?:TEXT ) group by s_name order by balance limit 100;
```

Q3:
```
select sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate,
o_shippriority from lineitem, orders, customer where c_custkey = o_custkey
and l_orderkey = o_orderkey and c_mktsegment = ?:TEXT and o_orderdate <
?:DATE and l_shipdate > ?:DATE group by o_orderdate, o_shippriority order
by revenue, o_orderdate limit 10;
```

Q4:
```
select o_orderpriority, count(*) from orders where o_orderdate >= ?:DATE
and o_orderdate < ?:DATE and exists ( select * from lineitem where l_orderkey
= o_orderkey and l_commitdate < l_receiptdate ) group by o_orderpriority
order by o_orderpriority;
```

Q5:
```
select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue from
lineitem, orders, customer, supplier, nation, region where c_custkey = o_custkey
and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey =
s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey
and r_name = ?:TEXT and o_orderdate >= ?:DATE and o_orderdate < ?:DATE group
by n_name order by revenue desc;
```

Q6:
```
select sum(l_extendedprice*l_discount) from lineitem where l_shipdate
>= ?:DATE and l_shipdate < ?:DATE and l_discount >= ?:FLOAT64 and l_discount
<= ?:FLOAT64 and l_quantity < ?:FLOAT64;
```

Q7:
```
select n1.n_name as supp_nation, n2.n_name as cust_nation, sum(l_extendedprice
* (1 - l_discount)) from lineitem, orders, supplier, customer, nation n1,
nation n2 where s_suppkey = l_suppkey and o_orderkey = l_orderkey and c_custkey
= o_custkey and s_nationkey = n1.n_nationkey and c_nationkey = n2.n_nationkey
and ((n1.n_name = ?:TEXT and n2.n_name = ?:TEXT) or (n1.n_name = ?:TEXT
and n2.n_name = ?:TEXT)) and l_shipdate >= 19950101 and l_shipdate <= 19961231
group by n1.n_name, n2.n_name order by supp_nation, cust_nation;
```

Q8:
```
select n2.n_name as nation, sum(l_extendedprice * (1-l_discount)) from
lineitem, orders, part, supplier, customer, nation n1, nation n2, region
where p_partkey = l_partkey and s_suppkey = l_suppkey and l_orderkey = o_orderkey
and o_custkey = c_custkey and c_nationkey = n1.n_nationkey and n1.n_regionkey
= r_regionkey and r_name = ?:TEXT and s_nationkey = n2.n_nationkey and o_orderdate
>= 19950101 and o_orderdate <= 19961231 and p_type = ?:TEXT group by n2.n_name;
```

Q10:
```
select c_custkey, n_name, sum(l_extendedprice * (1 - l_discount))
as revenue from lineitem, orders, customer, nation where c_custkey = o_custkey
and l_orderkey = o_orderkey and o_orderdate >= ?:DATE and o_orderdate <
?:DATE and l_returnflag = 'R' and c_nationkey = n_nationkey group by c_custkey,
```

```
n_name order by revenue desc limit 20;
```

Q11:
```
select ps_partkey, sum(ps_supplycost * ps_availqty) as value from
partsupp, supplier, nation where ps_suppkey = s_suppkey and s_nationkey
= nation.n_nationkey and n_name = ?:TEXT group by ps_partkey order by value
limit 10;
```

Q12:
```
select l_shipmode, sum(case when o_orderpriority = '1-URGENT' or o_orderpriority
= '2-HIGH' then 1 else 0 end) as high_line_count, sum(case when o_orderpriority
!= '1-URGENT' and o_orderpriority != '2-HIGH' then 1 else 0 end) as low_line_count
from orders, lineitem where o_orderkey = l_orderkey and l_shipmode = ?:TEXT
and l_commitdate < l_receiptdate and l_shipdate < l_commitdate and l_receiptdate
>= ?:DATE and l_receiptdate < ?:DATE group by l_shipmode order by l_shipmode;
```

Q14:
```
select sum(case when p_size <= 5 then l_extendedprice * (1 - l_discount)
else 0.0 end), sum(l_extendedprice * (1 - l_discount)) from lineitem, part
where l_partkey = p_partkey and l_shipdate >= ?:DATE and l_shipdate < ?:DATE;
```

Q15:
```
select l_suppkey, sum(l_extendedprice * (1 - l_discount)) as total_revenue
from lineitem where l_shipdate >= ?:DATE and l_shipdate < ?:DATE group by
l_suppkey order by total_revenue desc limit 10;
```

Q17:
```
select sum(0.7 * l_extendedprice) from lineitem, part where p_partkey
= lineitem.l_partkey and p_brand = ?:TEXT and p_container = ?:TEXT and l_quantity
< ( select 0.2 * avg(l_quantity) from lineitem where l_partkey = p_partkey
);
```

Q18:
```
select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice, sum(l_quantity)
from customer, orders, lineitem where c_custkey = o_custkey and o_orderkey
= l_orderkey and o_orderkey in ( select l_orderkey from lineitem group by
l_orderkey having sum(l_quantity) > ?:FLOAT64 ) group by c_name, c_custkey,
o_orderkey, o_orderdate, o_totalprice order by o_totalprice, o_orderdate
limit 100;
```

Bibliography

[1] PostgreSQL database, http://www.postgresql.org/.

[2] Hussam Abu-Libdeh, Deniz Altinbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. Learned indexes for a google-scale disk-based database. *CoRR*, abs/2012.12501, 2020.

[3] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Jim Chen, Min Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh Sr, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Jun Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divy Agrawal, Jeffrey F. Naughton, Sujata Kosalge, and Hakan Hacigümüs. Napa: Powering scalable data warehousing with robust query performance at google. *Proc. VLDB Endow.*, 14(12):2986–2998, 2021.

[4] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005: Demo. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 930–932, New York, NY, USA, 2005. Association for Computing Machinery.

[5] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[6] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370, 2004.

[7] Amazon. Amazon Redshift Automatic Table Optimization.

[8] Amazon. Amazon Redshift AutoMV.

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

[10] Manos Athanassoulis and Anastasia Ailamaki. Bf-tree: Approximate tree indexing. *Proc. VLDB Endow.*, 7(14):1881–1892, October 2014.

[11] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. Optimal column layout for hybrid workloads. *Proc. VLDB Endow.*, 12(13):2393–2407, September 2019.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Amazon AWS. Openstreetmap on aws. https://registry.opendata.aws/osm/.

[14] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990.

[15] Michael Bender, Martin Farach-Colton, and Miguel Mosteiro. Insertion sort is o(n log n). *Theory of Computing Systems*, 39, 06 2006.

[16] Michael A Bender and Haodong Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)*, 32(4):26, 2007.

[17] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[18] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, January 1981.

[19] Vivek Bharathan, Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Chuck Bear, and Ariel Cary. Materialization strategies in the vertica analytic database: Lessons learned. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 1196–1207, USA, 2013. IEEE Computer Society.

[20] Timo Bingmann. Stx b+ tree. https://panthema.net/2007/stx-btree/.

[21] Paul Brown and Peter J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *VLDB*, 2003.

[22] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016.

[23] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment*, 15(1):112–126, 2021.

[24] Surajit Chaudhuri and Vivek Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 1997.

[25] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14, 2007.

[26] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[27] Surajit Chaudhuri and Gerhard Weikum. Self-management technology in databases. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[28] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. *SIGMOD Rec.*, 30(2):235–246, May 2001.

[29] Zach Christopherson. Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys, 2016.

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[32] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57, sep 2010.

[33] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[34] Armon Dadgar. Adaptive radix trees implemented in c.

[35] Benoît Dageville and Mohamed Zait. Sql memory management in oracle9i. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 962–973. Elsevier, 2002.

[36] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 155–171. USENIX Association, 2020.

[37] Zhenwei Dai and Anshumali Shrivastava. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier. *arXiv preprint arXiv:1910.09131*, 2019.

[38] Databricks. Databricks Delta Lake Z-Ordering.

[39] Databricks. Data skipping index, 2020.

[40] Databricks Delta Engine. Z-Ordering (multi-dimensional clustering), 2020.

[41] Databricks Engineering Blog. Processing Petabytes of Data in Seconds with Databricks Delta. `https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html`.

[42] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *2020 Conference on Extending Database Technology (EDBT*, 2020.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Ani Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. SageDB: An Instance-Optimized Data Analytics System. Technical report, MIT, 2022.

[45] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. *Instance-Optimized Data Layouts for Cloud Analytics Workloads*, page 418–431. Association for Computing Machinery, New York, NY, USA, 2021.

[46] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data*, 2020.

[47] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.*, 14(2):74–86, oct 2020.

[49] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads, 2020.

[50] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[51] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.*, 12(9):1044–1057, May 2019.

[52] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. Supporting table partitioning by reference in oracle. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1111–1122, New York, NY, USA, 2008. Association for Computing Machinery.

[53] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind M. Joshi. Execution strategies for sql subqueries. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 993–1004, New York, NY, USA, 2007. Association for Computing Machinery.

[54] Mike Stonebraker et al. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st VLDB Conference*. VLDB Endowment, 2005.

[55] Evan Hallmark. Daily Historical Stock Prices (1970 - 2018). `https://www.kaggle.com/ehallmar/daily-historical-stock-prices-1970-2018`, 2020.

[56] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. Why are learned indexes so effective? In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3123–3132. PMLR, 13–18 Jul 2020.

[57] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.

[58] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In Philippe Jacquet, editor, *AofA: Analysis of Algorithms*, volume DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, pages 137–156, Juan les Pins, France, June 2007. Discrete Mathematics and Theoretical Computer Science.

[59] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30:170–231, 1998.

[60] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1189–1206, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1189–1206, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Cesar A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. Optimizing star join queries for data warehousing in microsoft sql server. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, page 1190–1199, USA, 2008. IEEE Computer Society.

[63] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 213–231, Carlsbad, CA, October 2018. USENIX Association.

[64] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, may 2001.

[65] Goetz Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*, DaMoN '06, page 5–es, New York, NY, USA, 2006. Association for Computing Machinery.

[66] Goetz Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3), July 2010.

[67] Goetz Graefe. Modern b-tree techniques. *Found. Trends Databases*, 3(4):203–402, April 2011.

[68] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 152–159, 1996.

[69] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proc. VLDB Endow.*, 7(12):1259–1270, aug 2014.

[70] Michael Hammer and Arvola Chan. Index selection in a self-adaptive data base management system. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of data*, pages 1–8, 1976.

[71] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, page 283–294, New York, NY, USA, 2003. Association for Computing Machinery.

[72] Benjamin Hilprecht, Carsten Binnig, Tiemo Bang, Muhammad El-Hindi, Benjamin Hättasch, Aditya Khanna, Robin Rehrmann, Uwe Röhm, Andreas Schmidt, Lasse Thostrup, and Tobias Ziegler. DBMS fitting: Why should we learn what we already know? In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[73] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 143–157, New York, NY, USA, 2020. Association for Computing Machinery.

[74] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. In *International Conference on Learning Representations*, 2019.

[75] Dawei Huang, Dong Young Yoon, Seth Pettie, and Barzan Mozafari. Joins on samples: A theoretical guide for practitioners. *Proc. VLDB Endow.*, 13(4):547–560, December 2019.

[76] IBM. The Spatial Index. `https://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.spatial.doc/ids_spat_024.htm`.

[77] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International*

Conference on Management of Data, SIGMOD '18, page 535–550, New York, NY, USA, 2018. Association for Computing Machinery.

[78] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. Deepsqueeze: Deep semantic compression for tabular data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1733–1746, New York, NY, USA, 2020. Association for Computing Machinery.

[79] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 647–658, New York, NY, USA, 2004. Association for Computing Machinery.

[80] Piotr Indyk, Yaron Singer, Ali Vakilian, and Sergei Vassilvitskii. STOC'20 Workshop on Algorithms with Predictions.

[81] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An architecture for recycling intermediates in a column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 309–320, New York, NY, USA, 2009. Association for Computing Machinery.

[82] Zachary G. Ives and Nicholas E. Taylor. Sideways information passing for push-style query processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, page 774–783, USA, 2008. IEEE Computer Society.

[83] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. Idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, June 2005.

[84] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, March 2018.

[85] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *Proc. VLDB Endow.*, 13(3):252–265, November 2019.

[86] Irfan Khan. Falling ram prices drive in-memory database surge. `https://www.itworld.com/article/2718428/falling-ram-prices-drive-in-memory-database-surge.html`, 2012.

[87] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In

Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, page 339–350, New York, NY, USA, 2010. Association for Computing Machinery.

[88] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation maps: A compressed access method for exploiting soft functional dependencies. *Proc. VLDB Endow.*, 2(1):1222–1233, August 2009.

[89] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Coradd: Correlation aware database designer for materialized views and indexes. *Proc. VLDB Endow.*, 3(1–2):1103–1113, September 2010.

[90] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[91] Andreas Kipf, Ryan Marcus, Alexander Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: A single-pass learned index. In *aiDM 2020*, 04 2020.

[92] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. Rl-cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.

[93] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[94] Yannis Kotidis and Nick Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 371–382, New York, NY, USA, 1999. Association for Computing Machinery.

[95] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[96] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[97] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[98] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[99] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[100] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[101] Hans-Peter Kriegel and Bernhard Seeger. Plop-hashing: A grid file without directory. In *Proceedings of the Fourth International Conference on Data Engineering*, page 369–376, USA, 1988. IEEE Computer Society.

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

[103] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[104] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *Proc. VLDB Endow.*, 11(7):759–772, March 2018.

[105] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. Automatic configuration for ibm db2 universal database. *Proc. of IBM Perf Technical Report*, 2002.

[106] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to sql server column stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1159–1168, New York, NY, USA, 2013. Association for Computing Machinery.

[107] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 38–49, USA, 2013. IEEE Computer Society.

[108] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 International Conference on Management of Data*, 2020.

[109] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2119–2133, New York, NY, USA, 2020. Association for Computing Machinery.

[110] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. Opportunistic view materialization with deep reinforcement learning, 2019.

[111] David B Lomet. Digital b-trees. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, pages 333–344. VLDB Endowment, 1981.

[112] David B. Lomet. Partial expansions for file organizations with an index. *ACM Trans. Database Syst.*, 12(1):65–84, March 1987.

[113] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. APEX: A high-performance learned index on persistent memory. *Proc. VLDB Endow.*, 15(3):597–610, 2021.

[114] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. Adaptdb: Adaptive partitioning for distributed joins. *Proc. VLDB Endow.*, 10(5):589–600, January 2017.

[115] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4), jul 2021.

[116] Lin Ma, Dana Van Aken, Amed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD*. ACM, 2018.

[117] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 631–645, New York, NY, USA, 2018. Association for Computing Machinery.

[118] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, 2018.

[119] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. Mb2: Decomposed behavior modeling for self-driving database management systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1248–1261, 2021.

[120] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy G. Mattson, and Nesime Tatbul. Self-organizing data containers. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[121] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[122] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[123] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[124] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. *Bao: Making Learned Query Optimization Practical*, page 1275–1288. Association for Computing Machinery, New York, NY, USA, 2021.

[125] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[126] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *CoRR*, abs/1904.03711, 2019.

[127] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[128] Materialize. Materialize.

[129] Donald Meagher. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. Technical report, Rensselaer Polytechnic Institute, 1980.

[130] Microsoft. Columnstore indexes - Query performance, 2019.

[131] Microsoft SQL Server. Spatial indexes overview. https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-indexes-overview?view=sql-server-2017, 2016.

[132] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[133] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing (pdf). Technical report, IBM, 1966.

[134] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing (pdf). Technical report, IBM, 1966.

[135] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. Cliffguard: A principled framework for finding robust database designs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1167–1182, New York, NY, USA, 2015. Association for Computing Machinery.

[136] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. Recycling in pipelined query evaluation. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 338–349, 2013.

[137] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 985–1000, New York, NY, USA, 2020. Association for Computing Machinery.

[138] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 International Conference on Management of Data*, SIGMOD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[139] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1137–1148, New York, NY, USA, 2011. Association for Computing Machinery.

[140] Thomas Neumann. The case for b-tree index structures. http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html, 2017.

[141] Thomas Neumann and Michael J Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[142] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. In Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 383–402. GI, 2015.

[143] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, March 1984.

[144] NYC Taxi & Limousine Commission. TLC Trip Record Data. `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`, 2020.

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

[146] Beng Chin Ooi, Ron Sacks-davis, and Jiawei Han. Indexing in spatial databases, 2019.

[147] Oracle. Oracle Automatic Materialized Views.

[148] Oracle. Bitmap Join Indexes, 2020.

[149] Oracle. Database Data Warehousing Guide: Using Zone Maps, 2020.

[150] Oracle Database Data Warehousing Guide. Attribute Clustering. `https://docs.oracle.com/database/121/DWHSG/attcluster.htm`, 2017.

[151] Oracle, Inc. Oracle Database In-Memory. `https://www.oracle.com/database/technologies/in-memory.html`.

[152] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, page 4. ACM, 2018.

[153] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow.*, 11(6):663–676, February 2018.

[154] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. internal: An essay on machine learning agents for autonomous database management systems. *IEEE Data Engineering Bulletin*, pages 32–46, June 2019.

[155] Andrew Pavlo, Matthew Butrovich, Lin Ma, Wan Shen Lim, Prashanth Menon, Dana Van Aken, and William Zhang. Make your database system dream of electric sheep: Towards self-driving operation. *Proc. VLDB Endow.*, 14(12):3211–3221, 2021.

[156] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[157] Luis L. Perez and Christopher M. Jermaine. History-aware query optimization with materialized intermediate views. In *2014 IEEE 30th International Conference on Data Engineering*, pages 520–531, 2014.

[158] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 430–441, New York, NY, USA, 2013. Association for Computing Machinery.

[159] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002.

[160] Frank Ramsak1, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel . In *Proceedings of the 26th International Conference on Very Large Databases*. VLDB Endowment, 2000.

[161] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[162] Jun Rao and Kenneth A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 475–486, New York, NY, USA, 2000. Association for Computing Machinery.

[163] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 558–569, New York, NY, USA, 2002. Association for Computing Machinery.

[164] RocksDB. RocksDB. `https://rocksdb.org/`, 2020.

[165] Moni Naor Ronald Fagin, Amnon Lotem. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[166] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 249–260, New York, NY, USA, 2000. Association for Computing Machinery.

[167] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When are learned models better than hash functions? *CoRR*, abs/2107.01464, 2021.

[168] Scipy.org. scipy.optimize.basinhopping. `https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.basinhopping.html`.

[169] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[170] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 435–446, New York, NY, USA, 1996. Association for Computing Machinery.

[171] Dennis G. Severance and Guy M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, September 1976.

[172] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 229–241, New York, NY, USA, 2017. Association for Computing Machinery.

[173] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 01 1973.

[174] Hari Singh and Seema Bawa. A survey of traditional and mapreducebased spatial query processing approaches. *SIGMOD Rec.*, 46(2):18–29, September 2017.

[175] Sivaprasad Sudhir, Michael Cafarella, and Samuel Madden. Replicated layout for in-memory database systems. *Proc. VLDB Endow.*, 15(4):984–997, dec 2021.

[176] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1115–1126, New York, NY, USA, 2014. Association for Computing Machinery.

[177] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. Skipping-oriented partitioning for columnar layouts. *Proc. VLDB Endow.*, 10(4):421–432, November 2016.

[178] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 308–320, New York, NY, USA, 2020. Association for Computing Machinery.

[179] TPC. Tpc-c. `http://www.tpc.org/tpcc/`.

[180] TPC. TPC-H. `http://www.tpc.org/tpch/`, 2019.

[181] TPC. TPC-DS. `http://www.tpc.org/tpcds/`, 2020.

[182] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. Partitioned learned bloom filter. *CoRR*, abs/2006.03176, 2020.

[183] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *Proceedings of the 16th International Conference on Data Engineering*. IEEE, 2000.

[184] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, 2017.

[185] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[186] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 36–53, New York, NY, USA, 2019. Association for Computing Machinery.

[187] H. Wang, X. Fu, J. Xu, and H. Lu. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pages 569–574, 2019.

[188] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 993–1008, New York, NY, USA, 2017. Association for Computing Machinery.

[189] Wikipedia. Correlated Subquery, 2020.

[190] Wikipedia. Referential Integrity, 2021.

[191] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. Are updatable learned indexes ready?, 2022.

[192] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1223–1240, New York, NY, USA, 2019. Association for Computing Machinery.

[193] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1223–1240, New York, NY, USA, 2019. Association for Computing Machinery.

[194] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar F. Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 International Conference on Management of Data*, 2020.

[195] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. *CoRR*, abs/2004.10898, 2020.

[196] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables, 2020.

[197] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, November 2019.

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

[199] zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[200] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.