Building Instance Aware Systems using Explicit Performance Modeling

by

Vikram Nathan

B.A., Harvard University (2013)
S.M., Massachusetts Institute of Technology (2018)

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

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

Certified by .....
Mohammad Alizadeh
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

Building Instance Aware Systems using Explicit Performance Modeling

by

Vikram Nathan

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

Abstract

Computer systems are often optimized to realize the best performance possible. However, these optimization techniques are typically blind to the system's actual workload or the particular environment in which the system is deployed. The lack of awareness of the full picture, or "instance", limits the extent to which a system's performance can be improved. However, achieving instance awareness is difficult because it involves optimizing over a large search space of configurable parameters. This thesis explores the use of explicit performance modeling in the context of three different systems to accelerate this optimization process and therefore make instance awareness practical:

Flood is a multidimensional database index that is tuned for both lowest latency and minimal space overhead on a query distribution known ahead of time. Flood uses a simple grid-based index system that adjusts the number of partitions in each dimension based on the workload. The "knobs" Flood turns are the parameters of this grid, making it more flexible than existing indexes, which have fewer such knobs. Flood models system performance as a combination of features determined by these grid parameters, and uses explicit measurement to train this model.

Cortex is a correlation index that allows databases to index attributes correlated to already-indexed attributes with minimal additional overhead but substantial performance improvement. Cortex decides which points should be considered outliers and inliers in the correlation; this fine grained control is the set of knobs that Cortex needs to instance optimize its performance: for the hardware its on, the host index on the database, and the distribution of queries.

Minerva is an end-to-end transport algorithm for video streaming, which aims to

achieve *Quality-of-Experience* fairness, so all clients sharing a bottleneck link in the network have roughly equal picture quality and minimal stalls. Importantly, this is achieved without compromising the bandwidth share of non-video traffic and without any client knowing about the others' existence. Minerva achieves fairness by making the network instance aware, using information about the client's state and the videos being streamed to model the client's performance and scale the aggressiveness of its congestion control algorithm.

Thesis Supervisor: Mohammad Alizadeh
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This work is a culmination of many years of effort and the support of countless individuals, to whom I am extremely grateful.

My advisor, Mohammad Alizadeh, gave me the freedom to explore my interests, even as they changed over time between networks and database systems. His in-depth grasp of diverse fields and willingness to dive into the weeds with his students is a model for an advisor and mentor that I aspire to. He has taken countless hours, out of both his days and nights, to discuss details of experiments and paper writing when a deadline was approaching. Because of him, I have grown in how I think about research and communicate my ideas, and I am extremely grateful for his advice and guidance during my time at MIT.

Tim Kraska has been generous with his time in guiding the development of both Flood and Cortex. I've been lucky to have his support and advice in the last couple years at MIT. I am also grateful to Sam Madden for lending himself and his thoughtful presence to my thesis committee.

My time at MIT would not have been possible without Hari Balakrishnan, who accepted me into the graduate program and gave me an academic home in the Networks and Mobile Systems lab. The NMS group boasts some of the most brilliant minds, reliable sounding boards, and stellar ping pong partners at MIT. I fondly remember many of our stimulating conversations, particularly with the other occupants of 32G-982, and I'm indebted to all the richness and depth the group has added to my years as a doctorate student.

My fellow collaborators in graduate school have been a crucial part of my journey and have taught me so much about the art of research: Anirudh Sivaraman, James Mickens, Jialin Ding, Mehrdad Khani, Prateesh Goyal, Ravi Netravali, Ravichandra

Addanki, Srinivas Narayana, Venkat Arun, and Vibhaalakshmi Sivaraman. I've enjoyed working with all of them, and I will hold close the lessons and research habits they've taught me. They have truly made me a better researcher.

To my extended family and friends: thank you for giving me a kind, caring, and fun community outside of graduate school. You always rejoice in my successes and commiserate with me in my setbacks, and have given me a supportive network that I have leaned on multiple times over the past several years.

Most importantly, I have an immense debt of gratitude to my sister, Shreya Nathan, and parents, Anoo Nathan and Vaidhi Nathan, for their undying love and support of me. This degree is truly the culmination of everything they have given me over the course of my life. I am indescribably grateful for how much they have believed in me, pushed me to challenge myself, and been my cheerleaders throughout all the ups and downs of my life. This thesis is theirs as much as it is mine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems service requests that involve interactions between multiple complex components. For example, computer networks coordinate robust communication between multiple devices; operating systems handle interactions between numerous modules, like the kernel, filesystem, and memory management units; and databases build upon query optimizers, indexes, and both persistent and transient storage systems.

Systems must maintain high performance while also being simple to manage. Many systems strike this balance with a "few sizes fits all" approach: they offer a limited set of tunable parameters or "knobs" that can be adjusted by a system administrator. These knobs offer a small number of degrees of freedom that an admin can explore to adapt performance of the system for their application. For example, a database management system (DBMS) may offer the ability to create an index on a subset of attributes to speed up query-time accesses. However, limiting the set of tunable parameters with this approach also limits the extent to which a system can be optimized for peak performance. In particular, allowing a system to adapt itself either to the specific environment it operates within (e.g. type of machine) or to the

workload it executes (e.g. type of queries) could produce significant performance benefits. We refer to a system that is capable of this type of adaptation as *instance aware*.

The key challenges to building instance aware systems are twofold. First, a system must define the correct set of knobs that exposes sufficiently many degrees of freedom to allow it to adapt to its environment. This is important because the limited degrees of freedom in traditional systems yield an optimization surface that is often too limited to allow any appreciable change in performance metrics, and they typically fail to capture the richness of diverse workloads, leaving substantial room for improvement. Determining what these parameters should be may involve redesigning system components entirely to surface more tunable knobs (see Chapter 2) or crossing traditional abstractions to expose parameters farther down the stack (see Chapter 4). However, simply increasing the degrees of freedom to increase the room for optimization is insufficient. The second challenge is to efficiently find the instance optimal settings for these knobs. Exposing more knobs goes against the "few sizes fits all" approach of traditional systems design: the number of possible parameter configurations may range from hundreds of millions to 10^{100} or more. The system must able to automatically tune these knobs without explicit user guidance, necessitating an efficient optimization procedure. Requiring user input to choose optimal settings is impractical since it places the burden on the user to understand the complex interaction between (possibly correlated) knobs and system performance metrics, creating a large barrier to deployment.

This thesis explores techniques for making systems more instance aware, given advance knowledge of the environment and / or workload distribution. Formally, if we let $\mathcal{Z}$ be the known input distribution, and $P(z;\theta)$ be the performance of the system on request $z \sim \mathcal{Z}$ with configurable parameter values θ, then we seek the

optimal parameters that maximize expected performance:

$$\theta^*(\mathcal{Z}) = \text{argmax}_\theta \, \mathbb{E}_{z \sim \mathcal{Z}} \left[P(z; \theta) \right] \tag{1.1}$$

Finding θ^* is typically referred to as *instance optimization*. Notably, θ^* is only optimal for the input distribution $\mathcal{Z}$, and no guarantees are made for inputs not drawn from $\mathcal{Z}$. Note also that although this work, and Equation 1.1, focuses on optimizing *average* performance, one might also reasonably choose optimize for other metrics, such as worst-case or 99th percentile performance.

Instance awareness has the potential to significantly improve performance on inputs drawn from $\mathcal{Z}$. For example, a DBMS with advance knowledge of the query workload may opt to lay out the data in storage such that records matching many queries are faster to access than records that are rarely touched. However, as previously mentioned, one of the major challenges of instance awareness is optimizing Equation 1.1 for θ^* efficiently and automatically. Unfortunately, performing the optimization exactly as in Equation 1.1 is often untenable in practice: since real systems may take a long time to execute an input z, evaluating P on many workloads over a variety of parameter configurations is infeasible. For example, running a single query on a database may take several minutes, so characterizing the performance of this database over many configurations and workloads will be slow. In order to speed up the optimization, our instance optimization techniques share a common design:

Two-phased Optimization. Most system deployments have components that change at different timescales. For example, many database engines are housed on the same machine(s) and underlying tables, which may not change often or change significantly. However, they may observe workloads that change frequently with time, over the span of minutes or hours. The difference in time scale allows instance

optimization to also occur in two different phases. In the first phase, we measure and collect information that is static or changes on longer timescales, such as properties of the machine. This needs to happen only once before processing workloads, or at least extremely infrequently, since these properties are relatively constant. In the second phase, we adapt to aspects of the instance that change on shorter timescales, like the workload. The second phase happens more regularly, whenever the workload shifts enough. Since we run the first phase infrequently, it is acceptable for the first phase to take a longer time than the second phase. Work performed in the first phase can also be utilized the second phase, accelerating the second phase even further. See Table 1.1 for a summary of the two-phased approach in the three systems we design.

System Measurement. The first phase of the two-phased optimization approach is mostly responsible for measuring the system's static environment. Since we care about the performance of the system *in its particular environment*, we often perform runs of the either the full system itself or its individual components to gather data to build the system's performance model (see below). Runs of the system are typically slow, but we can afford that latency because this phase is performed only once, in advance of running any workloads.

Performance Model. In the second phase, our instance aware systems approximate P quickly instead of evaluating it exactly, by using an *explicit model of the system's performance* that is a function of the tunable knobs θ. This model typically uses data collected in the first phase and can take many forms, from a closed-form equation to a machine learning model that uses θ as an input. Notably, the systems we implement in this work opt for *simple* models, usually linear. Simple models are easy to interpret and optimize, while still providing substantial performance gains.

Optimization Technique. Since the parameter search space is large, we can

reduce the number of times P is evaluated by using optimization techniques that lend themselves to the form of P. For example, a convex P can be optimized with gradient descent, while other simple forms may lend themselves well to analytical approximations and solutions.

This thesis designs and implements the aforementioned instance awareness framework in three systems, discussed here and summarized in Table 1.1:

Flood is a multidimensional database index that is tuned for both lowest latency and small space overhead on a given query distribution. Flood uses a simple grid-based index system that adjusts the number of partitions in each dimension based on the workload. The "knobs" Flood turns are the parameters of this grid, making it more flexible than existing indexes, which have fewer such knobs. In the first optimization phase, Flood measures itself on random settings of these knobs, fixing the dataset, to construct a linear model of system performance that includes the true time required to access and scan grid cells. The inputs to this model are parameters of the grid and statistics derived from the dataset and workload. In the second phase, Flood computes these statistics over a sample workload, and determines the minimum of the performance model using gradient descent.

Cortex is a correlation index that allows databases to accelerate queries on attributes correlated to other indexed attributes, with minimal additional overhead but substantial performance improvement. Cortex's main contribution is an algorithm to decide which points should be considered outliers and inliers in a correlation. The outlier assignments of these records are the degrees of freedom Cortex uses to optimize its performance. In the first phase, Cortex measures properties of the underlying machine and host index, computing the relative latencies of sequential range scans versus random-access point lookups. It uses these measurements in the second phase, when

it constructs a cost model as a function of outlier assignment, weighted by the relative latencies computed in the first phase. This equation can be solved analytically for the optimal outlier assignment, giving Cortex the fine-grained control it needs to instance optimize its performance: for the hardware its on, the host index on the database, and the distribution of queries.

Minerva is an end-to-end transport protocol for video streaming systems that helps clients streaming simultaneously over a shared link to achieve viewing experiences of similar qualities. Though networks are usually blind to the applications running on top of them, Minerva makes the network instance aware by feeding it information about the videos being streamed; the resulting system has the power to adjust the "knob" that governs the aggressiveness of the underlying congestion control algorithm. In the first phase, Minerva runs the ABR algorithm used by the clients, a key component used to decide which bitrate the client chooses, to produce a *normalization function* and sent to each client individually. This normalization function depends on the distribution of videos that the provider expects to be streamed and is sent to clients before any streaming sessions begin. In the second phase, Minerva generate performance models in the form of utility functions that estimate the quality of experience for a client based on its network throughput and buffer level. In conjunction with the normalization function, Minerva can then dynamically adjust the video download rate while streaming, achieving *Quality-of-Experience* fairness, so all clients sharing a bottleneck link in the network have watching sessions of roughly equal picture quality and minimal stalls. Importantly, this fairness is achieved without compromising non-video traffic sharing the same link.

We provide additional background on these systems and further details of our contributions in §2.1, §3.1, and §4.1.

	Minerva	**Flood**	**Cortex**
Knobs	Aggresiveness in Congestion Control Algorithm	Order and number of partitions along each grid dimension	Inlier and Outlier Assignment
Performance Metric	QoE Fairness	Query Latency	Query Latency
Phase 1	Compute normalization and value functions	Determine weights for linear cost model using a random forest	Measure relative access latencies for machine and host index
Phase 2	Adjust rate control continuously throughout video	Measure workload statistics, use gradient descent to find optimal grid parameters	Solve performance model to find optimal outlier assignment
Performance Model	Equation	Random Forest	Linear Regression, Equation
Optimization Method	Custom online algorithm	Simulated Annealing	Analytical

Table 1.1: A summary of the systems designed and implemented in this thesis, and how they fit the instance optimization paradigm.

It is important to note that instance awareness is a spectrum, not a binary classification. Some systems may be *more* instance aware than others, e.g., if they take more of their environment into account during optimization. Indeed, it is reasonable to consider many system at least somewhat instance aware; for example, several database indexes adapt to the dataset provided to them. Identifying a threshold at which a system becomes definitively instance aware is difficult and not well-defined, and this thesis does not attempt to do so. Instead, the goal of this work is to identify ways that systems can become *more* instance aware, and to provide examples of systems that implement that paradigm. We expect that these systems could be made *even more* instance aware with further careful design, and we discuss these possibilities further in Chapter 5.

1.1 Other Approaches to Optimization

A natural question to ask about the aforementioned instance awareness paradigm is: how does it differ from other approaches to system optimization? We contrast two other existing techniques: cost-based optimizers and reinforcement learning.

Cost-based optimizers use a cost model to estimate the performance of multiple strategies, with the goal of selecting the best performing one. One of the most prevalent examples of cost-based optimizers are database query optimizers. Given a query, a query optimizer returns an efficient execution plan for that query by ordering the operators that need to be applied to the relevant tables. Modern optimizers build their cost model using statistics of the underlying dataset, e.g., by computing cardinality estimates of selections to decide on an optimal join order [57, 84, 6]. However, they differ from the instance aware systems we design in two key ways.

First, existing query optimizers make no *measurements* of system performance, leaving them blind to the features of the machine they are running on. For example, an unclustered index lookup or lookup join may be substantially faster on a machine that uses NVMe, since random access latencies are typically orders of magnitude lower than conventional hard disks. These differences might lead to choosing an alternate join strategy. Additionally, adapting to cache dynamics, understanding how data is partitioned and distributed off-machine, and measuring intra-cluster communication latencies (in a distributed setting) could all give a query optimizer more knowledge of its instance that would let it make better-informed decisions. Although some optimizers may let the user configure hardware-specific parameters, these parameters are limited, coarse-grained, and require manual entry, since they have no automatic measurement built in [37].

Second, while existing query optimizers use a query as input, they often do not

consider the distribution of *all* queries in the current workload. This distribution is a key part of the instance that can change execution strategy. For example, optimizers may pre-materialize frequently occurring aggregations, or might choose to incur the overhead of sorting the data and caching it, if that would allow fast merge joins on similar queries in the future. The ability of the systems we design to adapt to query *distributions* allows them to unlock improvements to aggregate workload performance that would not be greedily optimal when considering each query alone, in a vacuum.

Query optimizers thus differ from the types of instance aware systems that we design because they do not measure either the underlying hardware and workload distribution, two key aspects of the environment that this thesis utilizes in the systems we implement. Taking these into account could make cost-based optimizers more instance aware.

Another emerging systems optimization technique uses reinforcement learning (RL) to guide system behavior. RL systems decide on actions for the system to take, which could include setting particular parameters or guiding execution, and update their decision model based on the resulting performance or "reward".

RL-augmented systems capture measurements of the system implicitly in their reward, since they are often derived from actual performance metrics [57, 56]. Therefore, these systems do adapt to their environment. However, they do not *explicitly model* the underlying system, and instead opt to train (often less interpretable models) from scratch. As a result, RL-based systems pay a large upfront training cost, since generating enough training data to build an accurate model typically requires running the full system end-to-end a large number of times. Explicit cost models require little to no learning to properly fit, so they drastically decrease optimization time, making them much more practical to deploy. In addition, the high sample complexity of RL models requires them to evaluate system performance a many times in response to

changes in the instance. As a result, they are unable to re-optimize themselves in a timely manner, making their deployment impractical.

Therefore, in contrast to existing systems optimization techniques, our instance awareness paradigm lets us design systems that both adapt to the particular environment they are in, while maintaining practical upfront training and optimization costs.

1.2 Trade-offs of Instance Optimality

Instance optimization offers the possibility of improved performance on relevant workload distributions; however, it may not always be practical, for several reasons.

First, the workload distribution must be known ahead of time. Several systems see predictable usage over time, e.g. similar movies are viewed in a region at a given time, similar queries are asked of databases to construct analytics reports, etc. For these systems, instance optimality may be beneficial. However, if the instance cannot be known or approximated ahead of time, or if there are significant numbers of ad hoc queries, the system will typically not see much improvement in performance, since the expected distribution Z would differ from the workload observed in practice.

Second, instance optimization requires time for both phases of optimization. If the instance changes substantially, the system often has to "re-optimize" and incur the cost of one or both of these phases. This may be impractical for some systems, e.g. if the instance changes too rapidly relative to the re-optimization time or if re-optimization would interfere with the live performance of the system. Various techniques can be employed to prevent optimization time from affecting system performance, e.g. by performing it on a separate machine. A discussion of these techniques is beyond the scope of this work.

Chapter 2

Flood: Learning Multi-Dimensional Indexes

2.1 Background

Analytical queries issued database engines, referred to as OLAP workloads, often require the database to filter data on multiple attributes. For example, a database may store a table T of orders that has attributes such as `order_id`, `time`, `subtotal`, `total`, `tax`, `store_id`, etc. A user interested in the total revenue generated from large orders from a particular store might write the following SQL query:

```
SELECT SUM(total) from T where subtotal > 100 AND store_id = 36
```

In this example, the predicates on `subtotal` and `store_id` are the relevant *filters*. To answer this query, a database management system (DBMS) may scan every record, and only aggregate those that match the filters. However, this incurs the overhead of accessing the data for each record in the table, even ones that don't match the filters, which results in a significant amount of work that does not change the aggregation result. Most modern DBMSes therefore resort to some form of *indexing*: organizing the dataset ahead of time and limiting the number of records that need to be accessed at query time. For example, a table may split records into two partitions: those with

`subtotal` > 75 and those with `subtotal` ≤ 75. When answering the above query, the query engine would access only the first partition, saving the access time incurred from records in the second partition. As a result, the *scan overhead* for this query, defined as the ratio of records accessed to records that match the query, is lower, which generally result in a faster query execution time. Techniques to reduce the access and filtering latency at query time differ in a number of ways, depending on the application. We discuss a few of the major differences here.

Clustered vs Secondary. With the growing popularity of column stores for large-scale analytical applications (OLAP), database engines often store data for a single column in one contiguous chunk in memory or on disk, "clustering" them together. Storing data contiguously results in faster sequential access due to better caching performance, as well as stronger compression efficiency, since similar values are more likely to appear closer together with a columnar layout. A clustered index is an index structure on top of this contiguously stored data. Figure 2-1 shows the difference between an attribute that uses a clustered index and one that does not.

Clustering allows for faster range scanning: since values are stored in order, a range scan between two attribute values A and B simply does a sequential scan of all records in storage between the first record with A and the last record with B. Additionally, it allows for smaller index sizes: clustered indexes can choose to group many adjacent records into a single "page" and index only the boundaries of the page. Single dimensional clustered indexes are commonly implemented as cache-optimized B-Trees as shown in Figure 2-1, while clustered multi-dimensional indexes may include quadtrees, R-trees, and kd-trees.

With an index on an attribute that is unclustered (and not a primary key), often called a secondary index, the query engine must follow a pointer from the index to the record it indicates. Accessing records within a range would then cause the engine

(a) Unclustered B-Tree index. Note that records with similar colors are not necessarily adjacent in storage, and requires extra overhead to store pointers.

(b) Clustered B-Tree index. Note that similar colors (values) are adjacent to each other in storage and can be grouped together in blocks or pages.

Figure 2-1: A comparison of clustered and unclustered indexes. Color strengths denotes attribute values. The same three adjacent values are highlighted in both indexes to illustrate that similar values have different relative positions.

to jump between non-sequential locations in storage, which is cache-inefficient and therefore slow. However, secondary indexes are useful when the sort order of records is already determined by an index on a different attribute: in this case, if the query is not *selective*, e.g. its filters do not match many records, a secondary index can still accelerate a query, since the index's cache-inefficiency is outweighed by its filtering power. This distinction between clustered and secondary indexes is critical to one of the contributions of this thesis, Cortex.

Single vs. Multi-dimensional. Indexes may incorporate multiple columns in order to provide lower scan overhead on queries with more than one filter, such as the SQL query in the previous section. Examples of commonly-used multi-dimensional indexes include quadtrees (or octrees in higher dimensions), kd-trees, R-trees, and Z-score indexes. Here, we contrast two of these approaches.

A *quadtree* recursively divides the space of two attributes into four quadrants. If

the number of records in a quadrant exceeds the tree's pre-defined *page size*, that quadrant is split into four child quadrants. The final set of quadrants are the *pages* of the index. Note that when the index is applied to $d > 2$ attributes, each quadrant recursively split into 2^d child quadrants, and the index structure is termed an *octree*. At query time, records in all cells that intersect the query region are fully scanned. The page size thus represents a trade-off between (a) query performance and (b) space usage and indexing time: a lower page size translates to smaller pages but a larger tree and more redirections to traverse it. Figurebg:indexing:octree shows an example of a quadtree on a sample dataset of 2 attributes, along with the region scanned at query time.

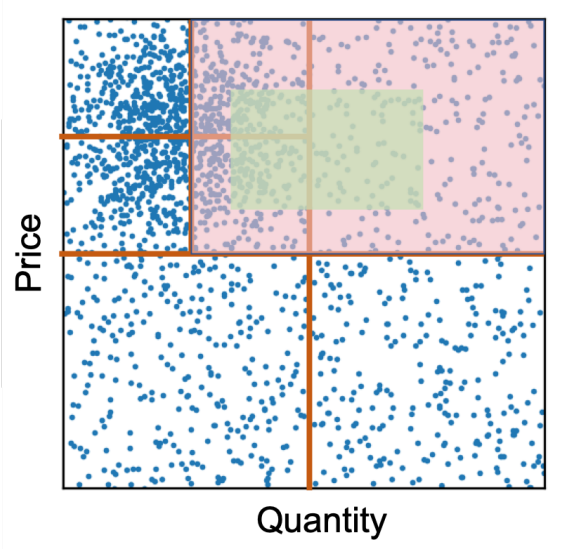


Figure 2-2: The octree (in this case, a quadtree) on two toy attributes, recursively divides space into equally sized quadrants. For each query (green), the index scans all intersected quadrant (red).

In contrast to a *quadtree*, which builds an explicit paged data structure, a Z-score index (or Z-index) is a sort order over multiple attributes simultaneously. It assigns

every point a Z-value based on its location along a space-filling Z-curve, shown in Figure 2-3.

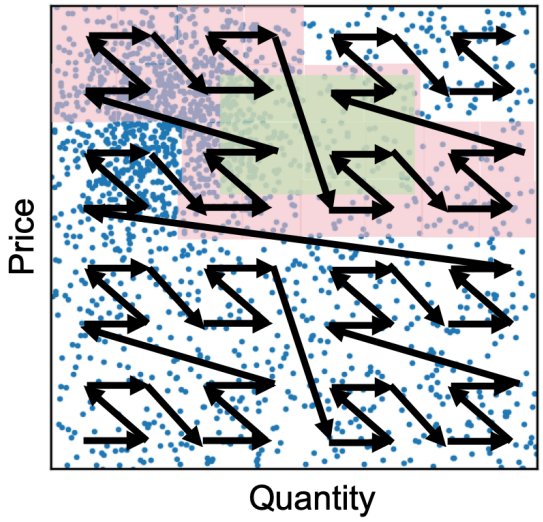


Figure 2-3: The z-index, on two toy attributes, maps points in many dimensions to a single number based on their location on the Z-curve. A sample query rectangle is shown in green, and the scanned region is shaded red.

The Z-value is then a function that maps multiple dimensions onto one, by interleaving the bits of its arguments in order. It is defined as a function $Z : \mathbb{Z}_+^n \to \mathbb{Z}$:

$$Z(x_1, \ldots x_n) = \sum_{i=1}^{n} \left[\sum_{k=0}^{d} (x_i[k]) \cdot 2^{nk+i} \right]$$

where $x_i[k] = x_i \;\&\; 2^k$ is the kth least-significant bit of x_i, and d is an integer such that $x_i < 2^{d+1}$ for all i.

When receiving a query rectangle for points x such that $A_i \leq x_i \leq B_i$, the index calculates:

$$Z_{\min} = Z(A_1, \ldots A_n) \qquad Z_{\max} = Z(B_1, \ldots, B_n)$$

At query time, the index locates the records with Z-values $Z_{\min}$ and $Z_{\max}$ and attempts to scan all points in between. It is important to note that the query may not correspond to a contiguous range of Z-values. Traversing Z-values in order may scan a large swath of points outside the query rectangle, as shown in Figure 2-3. To avoid scanning unnecessary ranges of points, whenever the index scans a point outside of the query rectangle, it computes the next Z-value inside the query rectangle, locates that point, and continues. In practice, continuously recomputing Z-values may be inefficient and result in high scan times. Therefore, many indexes may choose to chunk points sorted by Z-order into pages, similar to a B-Tree. If any of the Z-values in a page intersect the query rectangle, the entire page is scanned. This reduces the number of times the Z-value must be computed but may incur a larger scan overhead.

2.1.1 Our contribution: Flood

It is often the case that analytical workloads contain queries with filters in multiple dimensions and would benefit from using a multi-dimensional index. However, these indexes perform inconsistently across different datasets and workloads (or *instances*). For example, while a Z-order index may outperform an Octree on one workload, the opposite may be true on a different workload. Knowing which index is superior on a given instance is not straightforward, and typically falls to the database admin (DBA). Moreover, existing indexes offer few mechanisms to tune their performance, and when they do, the DBA must usually manually find the correct set of parameters to optimize performance.

We address this issue by designing and implementing *Flood*, the first multi-dimensional index with a grid-based index structure that is learned from *both* the underlying dataset and workload. Efficient optimization is made possible by sampling and a random forest model, which approximates query performance on the entire

dataset based on statistics derived from the grid parameters. Flood achieves up to three orders of magnitude faster performance for range scans with predicates than state-of-the-art multi-dimensional indexes or sort orders on real-world datasets and workloads. Our work serves as a building block towards an end-to-end learned database system.

2.2 Introduction

Scanning and filtering are the foundation of any analytical database engine, and several advances over the past several years specifically target database scan and filter performance. Most importantly, column stores [22] have been proposed to delay or entirely avoid accessing columns (i.e., attributes) which are not relevant to a query. Similarly, there exist many techniques to skip over records that do not match a query filter. For example, transactional database systems create a clustered B-Tree index on a single attribute, while column stores often sort the data by a single attribute. The idea behind both is the same: if the data is organized according to an attribute that is present in the query filter, the execution engine can either traverse the B-Tree or use binary search, respectively, to quickly narrow its search to the relevant range in that attribute. We refer to both approaches as clustered column indexes.

If data has to be filtered by more than one attribute, secondary indexes can be used. Unfortunately, their large storage overhead and the latency incurred by chasing pointers make them viable only for a rather narrow use case, namely when the predicate on the indexed attribute has a very high selectivity; in most other cases, scanning the entire table can be faster and more space efficient [21]. An alternative approach is to use *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 attributes (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. For example, both Redshift [5] and SparkSQL [16] use Z-ordering to lay out the data; Vertica can define a sort-order over multiple columns (e.g., first age, then date), while IBM Informix, along with other spatial database systems, uses an R-Tree [36].

However, multidimensional indexes still have significant drawbacks. First, these techniques are extremely hard to tune. For example, Vertica's ability to sort hierarchically on multiple attributes requires an admin to carefully pick the sort order. The admin must therefore know which columns are accessed together, and their selectivity, to make an informed decision. Second, there is no single approach (even if tuned correctly) that dominates all others. As our experiments will show, the best multidimensional index varies depending on the data distribution and query workload. Third, most existing techniques cannot be fully tailored for a specific data distribution and query workload. While all of them provide tunable parameters (e.g., page size), they do not allow finer-grained customization for a specific dataset and filter access pattern.

To address these shortcomings, we propose Flood, the first learned multi-dimensional in-memory index. Flood's goal is to locate records matching a query filter faster than existing indexes, by automatically co-optimizing the data layout and index structure for a particular data and query distribution.

Central to Flood are two key ideas. First, Flood uses a sample query filter workload to learn how often certain dimensions are used, which ones are used together, and which are more selective than others. Based on this information, Flood automatically customizes the entire layout to optimize query performance on the given workload. Second, Flood uses empirical CDF models to project the multi-dimensional and potentially skewed data distribution into a more uniform space. This "flattening"

step helps limit the number of points that are searched and is key to achieving good performance.

Flood's learning-based approach to layout optimization distinguishes it from other multi-dimensional index structures. It allows Flood to target its performance to a particular query workload, avoid the superlinear growth in index size that plagues some indexes [25], and locate relevant records quickly without the high traversal times incurred by k-d trees and hyperoctrees, especially for larger range scans.

While Flood's techniques are general and may benefit a wide range of systems, from OLTP in-memory transaction processing systems to disk-based data warehouses, this paper focuses on improving multi-dimensional index performance (i.e., reducing unnecessary scan and filter overhead) for an in-memory column store. In-memory stores are increasingly popular due to lower RAM prices [46] and the increasing amount of main memory which can be put into a single machine [23, 42]. In addition, Flood is optimized for reads (i.e., query speed) at the expense of writes (i.e., incremental index updates), making it most suitable for static analytical workloads, though our experiments show that adjusting to a new query workload is relatively fast. We envision that Flood 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) [73].

The ability to self-optimize allows Flood to outperform alternative state-of-the-art techniques by up to three orders of magnitude, while often having a significantly smaller storage overhead. More importantly though, Flood achieves *optimality across the board*: it has better, or at least on-par, performance compared to the next-fastest indexing technique on all our datasets and workloads. For example, on a real sales dataset, Flood achieves a boost of 3× over a tuned clustered column index and 72× over Amazon Redshift's Z-encoding method. On a different workload derived from

TPC-H, Flood is 61× faster than the clustered column index but only 3× faster than the Z-encoding. We make the following contributions:

1. We design and implement Flood, the first learned multi-dimensional index, on an in-memory column store. Flood targets its layout for a particular workload by learning from a sample filter predicate distribution.

2. We evaluate a wide range of multi-dimensional indexes on one synthetic and three real-world datasets, including one with a workload from an actual sales database at a major analytical database company. Our evaluation shows that Flood outperforms all other index structures.

3. We show that Flood achieves query speedups on different filter predicates and data sizes, and its index creation time is competitive with existing multi-dimensional indexes.

2.3 Related Work

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 [62], which maps multi-dimensional points onto a single dimension for sorting [5, 72, 95]. With spatial dimensions, SQL Server allows Z-ordering [59], and IBM Informix uses an R-Tree [36]. Other multi-dimensional indexes include K-d trees, octrees, R* trees, UB trees (which also make use of the Z-order), among many others (see [70, 82] for a survey). Flood's underlying index structure is perhaps most similar to Grid Files [69], which has many variants [87, 32, 35]. However, Grid Files do not automatically adjust to the query workload, yielding poorer performance (Section 2.8). In fact, Grid Files tend to have superlinear growth in index size even for uniformly distributed data [25].

Flood also differs from other adaptive indexing techniques such as database cracking [38, 39, 79]. The main goal of cracking is to build a query-adaptive incremental index by partitioning the data incrementally with each observed query. However, cracking produces only single dimensional clustered indexes, and does not jointly optimize the layout over multiple attributes. This limits its usefulness on queries with multi-dimensional filters. Furthermore, cracking does not take the data distribution into account and adapts only to queries; on the other hand, Flood adapts to both the queries *and* the underlying data.

Arguably most relevant to this work is automatic index selection [54, 12, 90]. However, these approaches mainly focus on creating secondary indexes, whereas Flood optimizes the storage and index itself for a given workload and data distribution.

For aggregation queries, data cubes [28] are an alternative to indexes. However, data cubes alone are insufficient for queries over arbitrary filter ranges, and they cannot support arbitrary actions over the queried records (e.g., returning the records themselves).

Finally, learned models have been used to replace/enhance traditional B-trees [49, 26, 17] and secondary indexes [92, 47]. Self-designing systems use learned cost models to synthesize the optimal algorithms for a data structure, resulting in a continuum of possible designs that form a "periodic table" of data structures [40]. Flood extends these works in two ways. First, Flood learns models for indexing *multiple* dimensions. Since there is no natural sort order for points in many dimensions, Flood requires a design tailored specifically to multi-dimensional data. Second, prior work focused solely on constructing models of the data, without taking queries into account. Flood optimizes its layout by learning from the query workload as well. Also unlike [40], Flood embeds models into the data structure itself.

SageDB [48] proposed the idea of a learned multi-dimensional index but did not

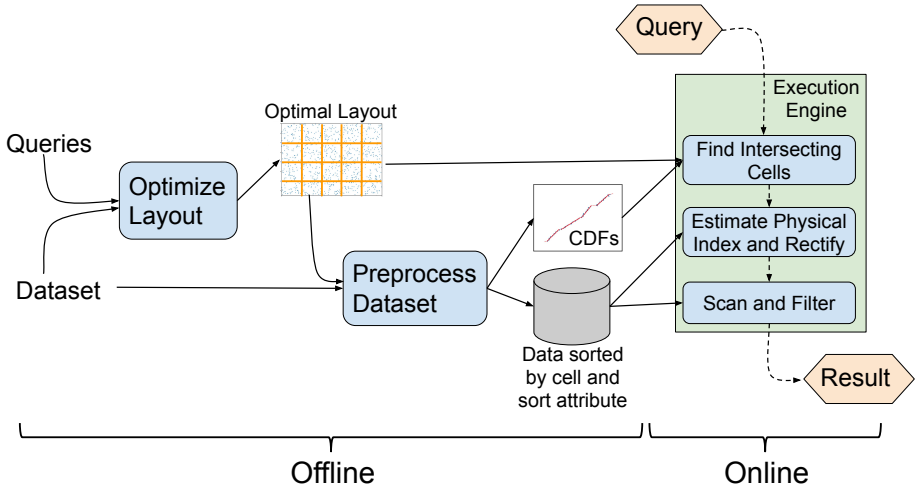


Figure 2-4: Flood's system architecture.

describe any details.

2.4 Index Overview

Flood is a multi-dimensional clustered index that speeds up the processing of relational queries that select a range over one or more attributes. For example:

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

Note that equality predicates of the form `R.Z == f` can be rewritten as `f` $\leq$ `R.Z` $\leq$ `f`. Typical selections generally also include disjunctions (i.e. `OR` clauses). However, these can be decomposed into multiple queries over disjoint attribute ranges; hence our focus on `AND`s.

Flood consists of two parts: (1) an offline preprocessing step that chooses an optimal layout, creating an index based on that layout, and (2) an online component responsible for executing queries as they arrive (see Figure 2-4).

At a high level, Flood is a variant of a basic grid index that divides d-dimensional data space into a d-dimensional grid of contiguous cells, so that data in each cell is stored together. We describe Flood's grid layout and online operation in Section 2.4.1 and Section 2.4.2. We then discuss Flood's central idea: how to automatically optimize the grid layout's parameters for a particular query workload (Section 2.5). The rest of this paper uses the terms *attribute* and *dimension* interchangeably, as well as the terms *record* and *point*.

2.4.1 Data Layout

Consider an index on d dimensions. Unlike the single dimensional case, points in multiple dimensions have no natural sort order. Our first goal is then to impose an ordering over the data.

We first rank the d attributes. The details of how to choose a ranking are discussed in Section 2.5, but for the purposes of illustration, we assume it is given. Next, we use the first $d-1$ dimensions in the ordering to overlay a $(d-1)$-dimensional grid on the data, where the ith dimension in the ordering is divided into c_i equally spaced columns between its minimum and maximum values. Every point maps to a particular *cell* in this grid, i.e. a tuple with $d-1$ attributes. In particular, if M_i and m_i are the maximum and minimum values of the data along the ith dimension, then define the dimension's range as $r_i = M_i - m_i + 1$. Then the cell for point $p = (p_1, \ldots, p_d)$ is:

$$\text{cell}(p) = \left(\left\lfloor \frac{p_1 - m_1}{r_1} \cdot c_1 \right\rfloor, \ldots, \left\lfloor \frac{p_{d-1} - m_{d-1}}{r_{d-1}} \cdot c_{d-1} \right\rfloor \right)$$

Note that the cell is determined only by the first $d-1$ dimensions; the dth dimension, the *sort dimension*, will be used to order points within a cell.

Flood orders the points using a depth-first traversal of the cells along the dimension

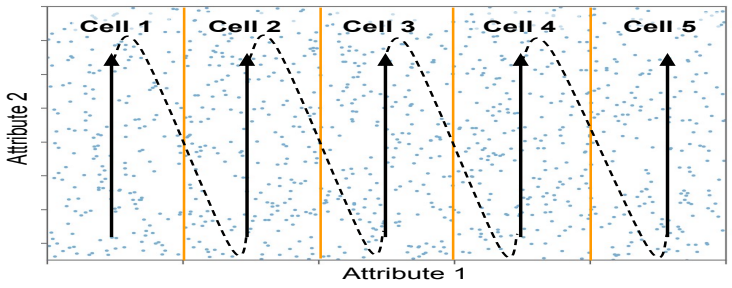


Figure 2-5: A basic layout in 2D, with dimension order (x, y) and $c_0 = 5$. Points are bucketed into columns along x and then sorted by their y-values, creating the seriliaziation order indicated by the arrows.

ordering, i.e. cells are sorted by the first value in the tuple, then the second, etc. Within each cell, points are sorted by their value in the dth dimension. Figure 2-5 illustrates the sort order for a dataset with two attributes.

Flood then sorts the data by this traversal. In other words, points in cell 0 (sorted by their sort dimension) come first, followed by cell 1, etc. Ties are broken arbitrarily.

2.4.2 Basic Operation

Flood receives as input a filter predicate consisting of ranges over one or more attributes, joined by ANDs. The intersection of these ranges defines a hyper-rectangle, and Flood's goal is to find and process exactly the points within this hyper-rectangle (e.g., by aggregating them). At a high level, Flood executes the following workflow (Figure 2-6):

1. **Projection**: Identify the cells in the grid layout that intersect with the predicate's hyper-rectangle. For each such cell, identify the range of positions in storage, i.e. the *physical index range*, that contains that cell's points (Section 2.4.2).

2. **Refinement**: If applicable, take advantage of the ordering of points within

each cell to shorten (or *refine*) each physical index range that must be scanned (Section 2.4.2).

3. **Scan**: For each refined physical index range, scan and process the records that match the filter.

Projection

In order to determine which points match a filter, Flood first determines which cells contain the matching points. Since the query defines a "hyper-rectangle" in the $(d-1)$-dimensional grid, computing intersections is straightforward. Suppose that each filter in the query is a range of the form $[q_i^s, q_i^e]$ for each indexed dimension i. If an indexed dimension is not present in the query, we simply take the start and end points of the range to be $-\infty$ and $+\infty$, respectively. Conversely, if the query includes a dimension not in the index, that filter is ignored at this stage of query processing.

The "lower-left" corner of the hyper-rectangle is $q^s = (q_0^s, \ldots, q_{d-1}^s)$ and likewise for the "upper-right" corner q^e. Both are shown in Figure 2-6. Then, we define the set of *intersecting cells* as $\{C_i \mid \text{cell}(q^s)_i \leq C_i \leq \text{cell}(q^e)_i\}$. Flood keeps a *cell table* which records the physical index of the first point in each cell. Knowing the intersecting cells then easily translates to a set of physical index ranges to scan.

Refinement

When the query includes a filter over the sort dimension, Flood uses the fact that points in each cell are ordered by the sort dimension to further refine the physical index ranges to scan. In particular, suppose the query includes a filter over the sort dimension $R.S$ of the form $a \leq R.S \leq b$. For each cell, Flood finds the physical indices of both the first point I_1 having $R.S \geq a$ and the last point I_2 such that $R.S \leq b$. This narrows the physical index range for that cell down to $[I_1, I_2]$. The simplest

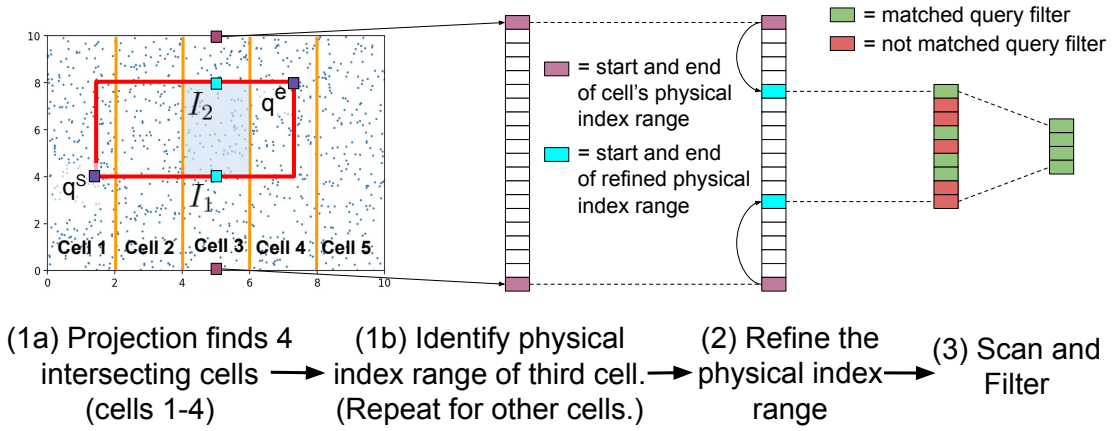


Figure 2-6: Basic flow of Flood's operation

way to find $[I_1, I_2]$ is by performing binary search within C on the values in the sort dimension. This is possible only because the points in C are stored contiguously in sorted order by the sort dimension. We discuss a faster way to refine, using models, in Section 2.6.2. If the query does not filter over the sort dimension, Flood skips the refinement step.

2.5 Optimizing the Grid

Flood's grid layout has several parameters that can be tuned, namely the number of columns allocated to each of the $d-1$ dimensions that form the grid, and which dimension to use as the sort dimension. Adjusting these parameters is the key way in which Flood optimizes performance on a given query workload. We found that the ordering of the $d-1$ grid dimensions did not significantly impact performance.

Adding more columns in each dimension allows Flood to scan a rectangle that more tightly bounds the true query filter, which reduces the number of points that must be scanned (Figure 2-7). However, adding more columns also increases the number of sub-ranges, which incurs extra cost for projection and refinement. Striking

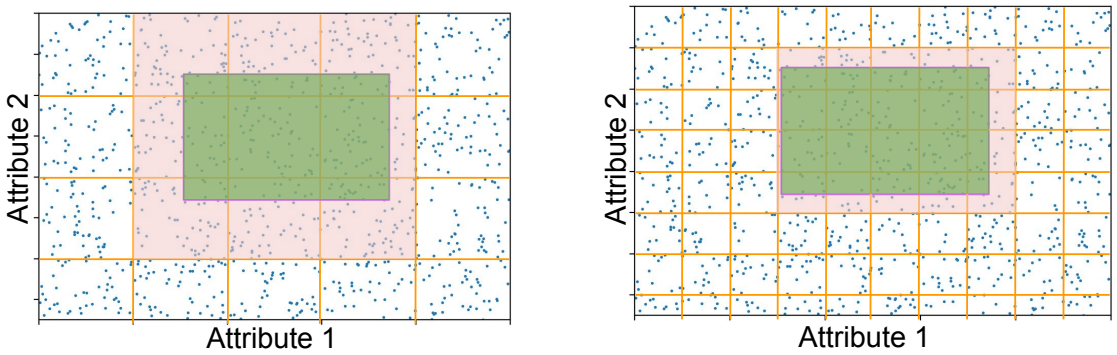


Figure 2-7: Doubling the number of columns can increase the number of visited cells but decreases the number of scanned points that don't match the filter (light red).

the right balance requires choosing a layout with an optimal number of columns in each dimension.

Flood can also select the sort dimension. The sort dimension is special because it will incur no scan overhead; given a query, Flood finds the precise sub-ranges to scan in the refinement step, so that the values in the sort dimension for scanned points are guaranteed to lie in the desired range. On the other hand, the grid dimensions do incur scan overhead because a certain column might only lie partially within the query rectangle. Therefore, the choice of sort dimension can have a significant impact on performance.

It is hard to select the optimal number of columns in each dimension because it depends on many interacting factors, including the frequency of queries filtering on that dimension, the average and variance of filter selectivities on that dimension, and correlations with other dimensions in both the data and query workload. The optimal sort dimension is also hard to select for similar reasons. Therefore, we optimize layout parameters using a cost model based approach. We first describe the cost model, then present the procedure that Flood uses to optimize the layout.

2.5.1 Cost Model

Define a layout over d dimensions as $L = (O, \{c_i\}_{0 \le i < d-1})$, where O is an ordering of the d dimensions, in which the dth dimension is the sort dimension and $\{c_i\}_{0 \le i < d-1}$ is the number of columns in the remaining $d-1$ grid dimensions.

Given a dataset D and a layout L, we model the query time of any query q as a sum of three parts, which correspond to the steps of the query flow from Section 2.4.2. Each part consists of some measurable statistic N, which is multiplied by a variable weight w which is a function of the dataset D, query q, and layout L, to produce an estimate of time taken on that step:

1. **Projection** contributes $w_p N_c$ to the query time, where N_c is the number of cells that fall within the query rectangle, and w_p is the average time to perform projection on a single cell. The weight w_p is not constant across all datasets, queries, and layouts. For example, it is faster to identify a block of cells along a single grid dimension, which are adjacent on linear storage media, than a hypercube of cells along multiple grid dimensions which are non-adjacent.

2. **Refinement** contributes $w_r N_c$ to the query time, where w_r is the average time to perform refinement on a cell. If the query q does not filter on the sort dimension, refinement is skipped and w_r is zero. Also, w_r is lower if the cell is smaller, because the piecewise linear CDF for that cell (explained in Section 2.6.2) is likely less complex and makes predictions more quickly.

3. **Scan** contributes $w_s N_s$ to the query time, where N_s is the number of scanned points, and w_s is the average time to perform each scan. The weight w_s depends on the number of dimensions filtered (fewer dimensions means fewer lookups for each scanned point), the run length of the scan (longer runs have better locality),

and how many scans fall within exact sub-ranges (explained in Section 2.8.1).

Putting everything together, our model for query time is:

$$Time(D, q, L) = w_p N_c + w_r N_c + w_s N_s \tag{2.1}$$

Given a dataset D and a workload of queries $\{q_i\}$, we find the layout L that minimizes the average of Equation 2.1 for all $q \in \{q_i\}$.

Calibrating the Cost Model Weights

Since the four weight parameters $w = \{w_p, w_r, w_s\}$ vary based on the data, query and layout, Flood uses models to predict w. The features of these weight models are statistics that can be measured when running the query on a dataset with a certain layout. These statistics include $N = \{N_c, N_s\}$, the total number of cells, the average, median, and tail quantiles of the sizes of the filterable cells, the number of dimensions filtered by the query, the average number of visited points in each cell, and the number of points visited in exact sub-ranges.

As we show in Section 2.8.7, the weight models are accurate across different datasets and query workloads. In particular, when new data arrives or the query distribution changes, Flood needs only to evaluate the existing models, instead of training new ones. Flood therefore trains the weight models once to calibrate to the underlying hardware. To produce training examples, Flood uses an arbitrary dataset and query workload, which can be synthetic. Flood generates random layouts by randomly selecting an ordering of the d dimensions, then randomly selecting the number of columns in the grid dimensions to achieve a random target number of total cells. Flood then runs the query workload on each layout, and measures the weights w and aforementioned statistics for each query. Each query for each random layout

will produce a single training example. In our evaluation, we found that 10 random layouts produces a sufficient number of training examples to create accurate models. Flood then trains a random forest regression model to predict the weights based on the statistics.

One natural question to ask is whether a single random forest model can be trained to predict query time, instead of factoring the query time as weighted linear terms and training a model for each weight. However, a single model is inadequate because we want to accurately predict query times across a range of magnitudes; a model for query time would optimize for accuracy of slow queries at the detriment of fast queries. On the other hand, the weights span a relatively narrow range (e.g., the average time to scan a point will not vary across orders of magnitude), so are more amenable to our goal.

Why Use Machine Learning?

We model the cost using machine learning because query time is a function of many interdependent variables with potentially non-linear relationships that are difficult to model analytically. For instance, on 10k training examples, Figure 2-8 shows not only that the empirical average time to scan a point (w_s) is not constant, but also that its dependence on two related features (number of scanned points and average scan run length, which affects locality) is non-linear and does not follow an obvious pattern.

Indeed, we found that query time predicted using a simple analytical model that replaces the weight parameters of Equation 2.1 with fine-tuned constants has on average 9× larger difference from the true query time than our machine-learning based cost model. Furthermore, predicting the weight parameters using a linear regression model that uses the same input features as our random forest produces query time predictions with 4× larger difference from the true query time, which

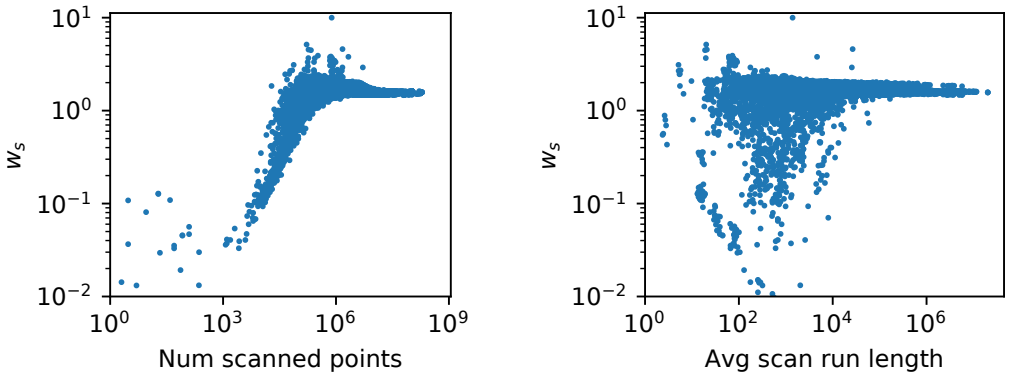


Figure 2-8: w_s (and by extension, scan time) is not constant and is difficult to model analytically because of its non-linear dependence on related features.

confirms that the features are interdependent and/or have non-linear relation with query time.

2.5.2 Optimizing the Layout

Given a calibrated cost model, Flood optimizes its layout for a specific dataset and query workload as follows:

1. Sample the dataset and query workload, then flatten the data sample and workload sample using RMIs trained on each dimension.

2. Iteratively select each of the d dimensions to be the sort dimension. Order the remaining $d-1$ dimensions that form the grid by the average selectivity on that dimension across all queries in the workload. This gives us O.

3. For each of these d possible orderings, run a gradient descent search algorithm to find the optimal number of columns $\{c_i\}_{0 \leq i < d-1}$ for the $d-1$ grid dimensions. The objective function is Equation 2.1. For each call to the cost model, Flood computes the statistics $N = \{N_c, N_s\}$ and the input features of the weight

models using the data sample instead of the full dataset D.

4. Select the layout with the lowest objective function cost amongst the d layouts.

Algorithm 1 provides pseudocode for this procedure, i.e. optimizing the layout using a calibrated cost model. Optimizing the layout is efficient (Section 2.8.7) because each iteration of gradient descent does not require building the layout, sorting the dataset, or running the query. Instead, statistics are either estimated using a sample of D or computed exactly from the query rectangle and layout parameters.

2.6 Learning from the Data

The simple index presented in Section 2.4 does not consider or adapt to the underlying distribution of the data. Here, we present two ways that Flood learns its layout from the data. First, *Flood uses a model of each attribute to better determine column spacing.* Second, *Flood accelerates refinement within each cell using a model of the underlying data.*

2.6.1 Flattening

The index in Section 2.4 spaces columns equally, but this type of layout is inefficient when indexing highly skewed data: some grid cells will have a large number of points, causing Flood to scan too many superfluous points.

If we were to have an accurate model of each attribute's distribution, i.e. its CDF, we could choose columns such that for each attribute, each column is responsible for approximately the same number of points. In practice, Flood models each attribute using a Recursive Model Index (RMI), a hierarchy of models, e.g. linear models in our case, that is quick to evaluate [49]. The input to the model is the attribute value v; the output is the fraction of points with values $\leq v$. At query time, suppose that we would like to split the kth dimension into n columns. A point with value v in the

Algorithm 1 Layout Optimization

Inputs: d-dimensional dataset D, query workload $Q = \{q_i\}$, cost model $T : (D, q, L) \rightarrow$ query time
Output: layout $L = (O, C)$, where O is the order of dimensions and $C = \{c_i\}_{0 \leq i < d-1}$ is the number columns in each grid dimension
procedure FindOptimalLayout(D, Q, T)
    $\widehat{D}$ = Sample(D)
    $\widehat{Q}$ = Sample(Q)
    /* RMIs trained on each dimension of $\widehat{D}$ are used to flatten the data and query workload samples
        by replacing each value v in the i-th dimension of a point or query with $\text{CDF}_i(v)$ */
    $\widehat{D}, \widehat{Q}$ = Flatten($\widehat{D}, \widehat{Q}$)
    dims = /* dimensions ordered by decreasing average selectivity of $q \in \widehat{Q}$ on $\widehat{D}$ */
    best_cost = ∞
    best_L = null
    for i in 0:d **do**
        O = {dims[0:i], dims[i+1,:], dims[i]} /* use i-th dimension as sort dimension */
        /* search for minimum cost $T(\widehat{D}, q, (O, C))$ averaged over $q \in \widehat{Q}$, assuming fixed order O, by varying C */
        C, cost = GradientDescent(T, O, $\widehat{D}$, $\widehat{Q}$) /* returns lowest found cost and the C that achieves it */
        L = (O, C)
        if cost < best_cost **then**
            best_cost = cost
            best_L = L
        end if
    end for
    return best_L
end procedure

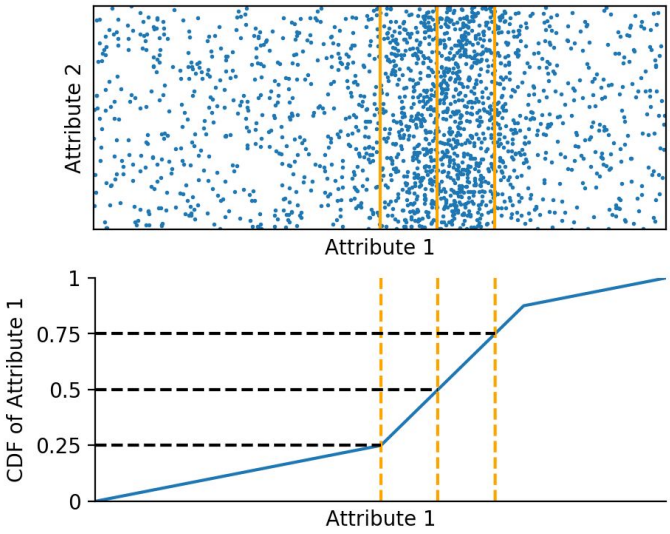


Figure 2-9: By flattening, each of the four columns in a dimension will contain a fourth of the points.

kth dimension will be placed into column $\lfloor \text{CDF}(v) \cdot n \rfloor$. Since evaluating the RMI is efficient, we can efficiently determine the columns in each dimension that the query intersects.

Figure 2-9 shows example 2-D data and the result of applying this transformation to Attribute 1. The column boundaries are no longer equally spaced across the range of Attribute 1's values. Instead, they are equally spaced in terms of the CDF of Attribute 1. This means that each of the four columns has around $1/4$ of the points. Since the number of points in each column is evened out, we call this a *flattened* layout. Flood applies this flattening transformation for each grid dimension. Skewness is abundantly present in real-world data; on two of the datasets used in our evaluation (Section 2.8), flattening provides a performance boost of $20-30\times$ over a non-flattened layout.

Note that while flattening may assign an equal number of points to each column of a single attribute, it does not guarantee that each cell in the final grid has a

similar number of points. In particular, if two attributes are correlated, flattening each attribute independently will not yield uniformly sized cells. This may lead to some cells incurring a high scan overhead. In practice, we found that modeling single attributes, i.e. assuming each dimension is independent, was sufficient. If necessary, adding more columns per dimension can further reduce the per-cell scan overhead. Flood's layout training procedure (Section 2.5) will choose the number of columns per dimension to trade off scan overhead with projection and refinement cost, mitigating the effect of non-uniform cell sizes. Additionally, if the correlation results in some cells being empty, those cells can be easily pruned using the cell table, incurring very little overhead. However, recent work [92, 47] suggests that it might be possible to further reduce the index size by taking advantage of the correlation. Exploring such techniques for multi-dimensional clustered indexes remains future work.

2.6.2 Faster Refinement

The simple index from Section 2.4.2 uses binary search over the sort dimension to refine the physical index range of each cell. In practice, since we may have to refine in every cell, binary search is too slow. Instead, Flood builds a CDF model over the sort dimension values for each cell. Flood uses a cell's model to estimate the endpoints of the refined physical index range, and then corrects any misprediction through a local search.

We want a model that can achieve a low average absolute error, while keeping the maximum error bounded to a reasonable value, in order for local search to be fast. Unfortunately, it is difficult to build an RMI with a target error bound. Instead, the model Flood uses is a *piecewise linear model* (PLM).

A PLM models a CDF by partitioning a sorted list of values V into slices, each of which is modeled by a linear segment. Let $P(v)$ be the predicted index of value

$v \in V$, determined by the segment responsible for the slice containing v, and let $D(v)$ be the index of the first occurrence of v. We require that the linear segments serve as a *lower bound* on the true CDF values, i.e. $P(v) \leq D(v)$, with the property that for every segment, the average absolute error is less than a given threshold δ:

$$\frac{1}{|V|} \sum_{v \in V} |D(v) - P(v)| \leq \delta$$

The lower bound property allows us to turn this condition into: $\frac{1}{|V|} \sum_{v \in V} D(v) - P(v) \leq \delta$, which is much easier to achieve.

Flood uses a greedy algorithm to partition V into slices: for each $v \in V$ in increasing order, it adds $(v, D(v))$ to the segment for the current slice. If the segment's average error over the values in current slice exceeds δ, it begins a new slice. The model records the smallest v in each slice and forms a cache-optimized B-Tree over those values. At inference time, Flood uses the B-Tree to find the appropriate segment and thus $P(v)$. The parameter δ encodes a tradeoff between size and speed (lower δ is faster): see Section 2.8.8 for experiments tuning δ and a comparison of the PLM to other methods.

2.7 Discussion

Tuning of traditional indexes. Existing multi-dimensional indexes strive for fewer hyperparameters, typically only a page size, to lower the overhead of tuning. However, fewer parameters also restricts the search space over which Flood can optimize its layout, limiting the speedups it can achieve over existing indexes. Indeed, Figure 2-11 demonstrates that simply tuning page size does not offer substantial performance improvement. In contrast, Flood's grid layout intentionally offers a larger number of parameters over which to optimize, all of which can be automatically tuned by Flood's

learning procedure. As a result, Flood can customize the layout for a particular query workload better than existing indexes.

Alternatives to grids. Flood is a learning-enhanced version of a basic grid index, but many alternative techniques to divide a multi-dimensional space exist (R-Tree, k-d tree, Z-order, etc.). We decided to use a grid structure for several reasons. First, it has a small space overhead: other multi-dimensional indexes use between 10MB and 1GB, but Flood's grid uses less than 1kB (Section 2.8), leaving ample room to add per-cell models. Second, the grid has low lookup latency, since it avoids pointer chasing. On the TPC-H dataset in Section 2.8, Flood with flattening takes 0.46ms to identify relevant grid cells (excluding refinement), while the k-d tree and hyperoctree take 8.9ms (20×) and 1.8ms (4×) to identify matching pages, respectively. This trend is consistent across the datasets we evaluate on. Z-order based indexes have low lookup times but expose no obvious parameters that can be tuned for the query workload. Note that our flattening approach is necessary to keep scan times low by making sure the grid is not highly imbalanced.

Nearest Neighbor Queries. Tree-based indexes that are used for geospatial data, such as k-d trees and R-trees, support $k-$nearest neighbor (kNN) queries. For example, a k-d tree locates the page with the query point and checks adjacent pages until all k neighbors are found. Flood can easily locate adjacent cells in its grid layout, allowing a similar kNN algorithm. However, since this paper does not focus on geospatial analytics, we exclude kNN queries from our evaluation.

Multi-dimensional CDFs. In Section 2.6.1, we mentioned that correlated dimensions can yield non-uniform data after flattening. To address this issue, for each pair of correlated dimensions, one could train a 2-dimensional joint CDF, or train a conditional CDF that creates a 1-D model for attribute A within each column of attribute B. However, it is difficult to ensure that a multi-dimensional RMI model

gives monotonic predictions along each dimension, which is a necessary property for partitioning points into columns; and conditional CDFs did not significantly improve performance in our benchmarks, but did significantly increase index size. Therefore, Flood does not use multi-dimensional CDFs. Efficiently modeling correlations between more dimensions is an active area of research [89, 75].

2.8 Evaluation

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

1. Flood achieves optimality *across the board*: it is faster than, or on par with, every other index on the tested workloads. However, the next best index changes depending on the dataset. On our datasets, Flood is up to 187× faster than a single-dimensional clustered column index, up to 62× faster than a Grid File, up to 72× faster than a Z-order index, up to 250× faster than an UB-tree, up to 43× faster than a hyperoctree, and up to 48× faster than a k-d tree or R-tree.

2. Flood's index can take up to 50× less space than the next fastest index.

3. Even though we did not optimize Flood for dynamic workloads, Flood can train its layout and reorganize the records quickly for a new query distribution, typically in under a minute for a 300 million record dataset.

4. Flood's performance over baseline indexes improves with larger datasets and higher selectivity queries.

2.8.1 Implementation

We implement Flood in C++ on a custom column store that uses *block-delta* compression: in each column, the data is divided into consecutive blocks of 128 values, and each value is encoded as the delta to the minimum value in its block. Our encoding scheme allows constant-time element access and is able to compress the datasets used in our evaluation by 77%.

Our implementation uses 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. We include two scan-time optimizations:

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

2. Similar to the idea of [51], our implementation allows indexes to speed up common aggregations like `SUM` by including a column in which the ith value is the cumulative aggregation of all elements up to index i. In the case of an exact range, the final aggregation result is simply the difference between the cumulative aggregations at the range endpoints. Note that this is not a data cube as we can support arbitrary ranges instead of only pre-aggregated ranges.

These additions are meant to demonstrate that Flood can take advantage of features that existing indexes enjoy. We show in Section 2.8.5 that these optimizations are not required for Flood: its performance benefits are due primarily to the optimality of

the layout and not the details of the underlying implementation. Our random forest regression uses Python's Scipy library [80].

We benchmark our column store with MonetDB, an open-source column store [60], by executing a query workload with full scans. Both MonetDB and our implementation were run single-threaded, with identical bit widths for each attribute, and without compression. Note that MonetDB does not support compression on numerical columns. Averaged over 150 aggregation queries on the TPC-H dataset (Section 2.8.3), our scan times are within 5% of MonetDB, showing that our column store implementation is on par with existing systems.

2.8.2 Baselines

We compare Flood to other solutions implemented on the same column store, with the same optimizations, if applicable. When querying any solution, the user provides two arguments: (1) the start and end value of the filter range in each dimension (set to negative and positive infinity if the dimension is not filtered in the query), and (2) a Visitor object which will accumulate the statistic of the aggregation. All of our experiments are performed on aggregation queries. Indexes only scan the columns for dimensions that appear in the query filter. The baseline indexes we implemented are:

1. **Clustered Single-Dimensional Index:** We use an RMI with three layers, where all models are linear models. Models in the non-leaf layers are linear spline models to ensure that the models accessed in the following layer are monotonic; the models in the leaf layer are linear regressions. The numbers of experts in each layer are $1, \sqrt{n}$, and n, respectively, with n tuned to minimize query time on the target workload.

2. **Grid File:** The d-dimensional space is divided into *blocks* by a grid (each block is one grid cell). Multiple adjacent blocks constitute a *bucket*. All points

in a bucket are stored contiguously and not sorted: if a record in a bucket needs to be accessed, the entire bucket must be scanned. The grid is built incrementally, starting with a single block that contains the entire space. Each point is added to its corresponding bucket; once the number of points in a bucket hits a user-defined page size, that bucket is split to form a new bucket by either (1) splitting points along an existing block boundary, if it exists in any dimension, or (2) adding a grid column (and therefore more blocks) that divides the existing bucket at its midpoint along a particular dimension. The dimension along which the block is split is cycled through in a round robin fashion. The page size is tuned to minimize query time on the target workload. When a query arrives, the grid file scans all the buckets that intersect the query rectangle.

3. **Z-Order Index:** We use 64-bit Z-order values. When indexing d dimensions, we compute the Z-order value for a point by taking the first $\lfloor 64/d \rfloor$ bits of each dimension's value and interleaving them, ordered by selectivity (e.g., the most selective dimension's LSB is the Z-order value's LSB). We order points by their Z-order value and group contiguous chunks into pages. For each page, we store the min and max value in each dimension for points in the page. Given a query, the index finds the smallest and largest Z-order value contained in the query rectangle (conceptually the bottom-left and top-right vertices of the query rectangle), uses binary search to find the physical indexes that correspond to those Z-order values, and iterates through every page that falls between those physical indexes. The points in a page are only scanned if the metadata min/max values indicate that it is possible for points in the page to match the query filter (i.e., we only scan a page if the rectangle formed by the page's

min/max values intersects with the query rectangle).

4. **UB-tree:** Z-order values are computed in the same way as the Z-Order Index. We order points by their Z-order value and group contiguous chunks into pages. For each page, we store the minimum Z-order value contained in that page. Given a query, the index finds the smallest and largest Z-order value contained in the query rectangle (conceptually the bottom-left and top-right vertices of the query rectangle), uses binary search to find the physical indexes that correspond to those Z-order values, and iterates through every physical index in this range. If we reach a Z-order value that is outside the query rectangle (the Z-order curve might enter and exit the query rectangle many times), we compute the next Z-order value that falls within the query rectangle. We then "skip ahead" to the page that contains this Z-order value, by comparing with each page's minimum Z-order value.

5. **Hyperoctree:** We recursively split into d-dimensional hyperoctants until each page has below the page size number of points. Points within a page are stored contiguously, and pages are ordered by an in-order traversal of the tree index. Each node in the hyperoctree contains an array of points to 2^d child nodes, the min and max value in each dimension for points in the page, and the start and end physical index for points in the page. Given a query, the index finds all pages that intersect with the query rectangle, uses the node's metadata to identify the physical index range for each page, and scans all physical index ranges.

6. **K-d tree:** We recursively partition space using the median value along each dimension, until the number of points in each page has below the page size

number of points. The dimensions are used for partitioning in a round robin fashion, in order of decreasing selectivity. If the remaining points all have the same value in a particular dimension, that dimension is no longer used for further partitioning. Points within a page are stored contiguously, and pages are ordered by an in-order traversal of the tree index. Each node in the k-d tree contains the pointers to its two children, the dimension that is split on, the split value, and the start and end physical index for points in the page. Given a query, the index finds all pages that intersect with the query rectangle, uses the node's metadata to identify the physical index range for each page, and scans all physical index ranges.

7. The *R*-Tree* is a read-optimized variant of the R-Tree that is bulk loaded to optimize for read query performance. We benchmark the R*-Tree implementation from libspatialindex [31]. On larger datasets, the R*-Tree was prone to out-of-memory errors and was not included in benchmarks.

While some techniques are read-optimized like Flood (e.g., the Z-Order and *R**-Tree), others are inherently more write-friendly (e.g., UB-tree); we still include them for the sake of comparison while optimizing them for reads as much as possible (e.g., using dense cache-aligned pages).

Our primary goal is to evaluate the performance of our multidimensional index as a fundamental building block for improving range request with predicates (e.g., filters) over one or more attributes. We therefore do not evaluate against other full-fledged database systems or queries with joins, group-bys, or other complex query operators. While the impact of our multi-dimensional clustered index on a full query workload for an in-memory column-store database system would be interesting, it requires major changes to any available open-source column-store and is beyond the scope of this

	sales	tpc-h	osm	perfmon
records	30M	300M	105M	230M
queries	1000	700	1000	800
dimensions	6	7	6	6
size (GB)	1.44	16.8	5.04	11

Table 2.1: Dataset and query characteristics.

paper. However, it should be noted that that even in its current form, Flood could be directly used as a component to build useful services, such as a multi-dimensional key-value store.

For a fair comparison, all benchmarks use a single thread without SIMD instructions. We exclude multiple threads mainly because our baselines were not optimized for it. We discuss how Flood could take advantage of parallelism and concurrency in Section 2.9. All experiments are run on an Ubuntu Linux machine with an Intel Core i9 3.6GHz CPU and 64GB RAM.

2.8.3 Datasets

We evaluate indexes on three real-world and one synthetic dataset, summarized in Table 2.1. Queries are either real workloads or synthesized for each dataset, and include a mix of range filters and equality filters. The **Sales** dataset is a 6-attribute dataset and corresponding query workload drawn directly from a sales database at a commercial technology company. It was donated to us by a large corporation on the condition of anonymity. The dataset consists of 30 million records, with an anonymizing transformation applied to each dimension. Each query in this workload was submitted by an analyst as part of report generation and analysis at the corporation.

Our second real-world dataset, **OSM**, consists of all 105 million records catalogued

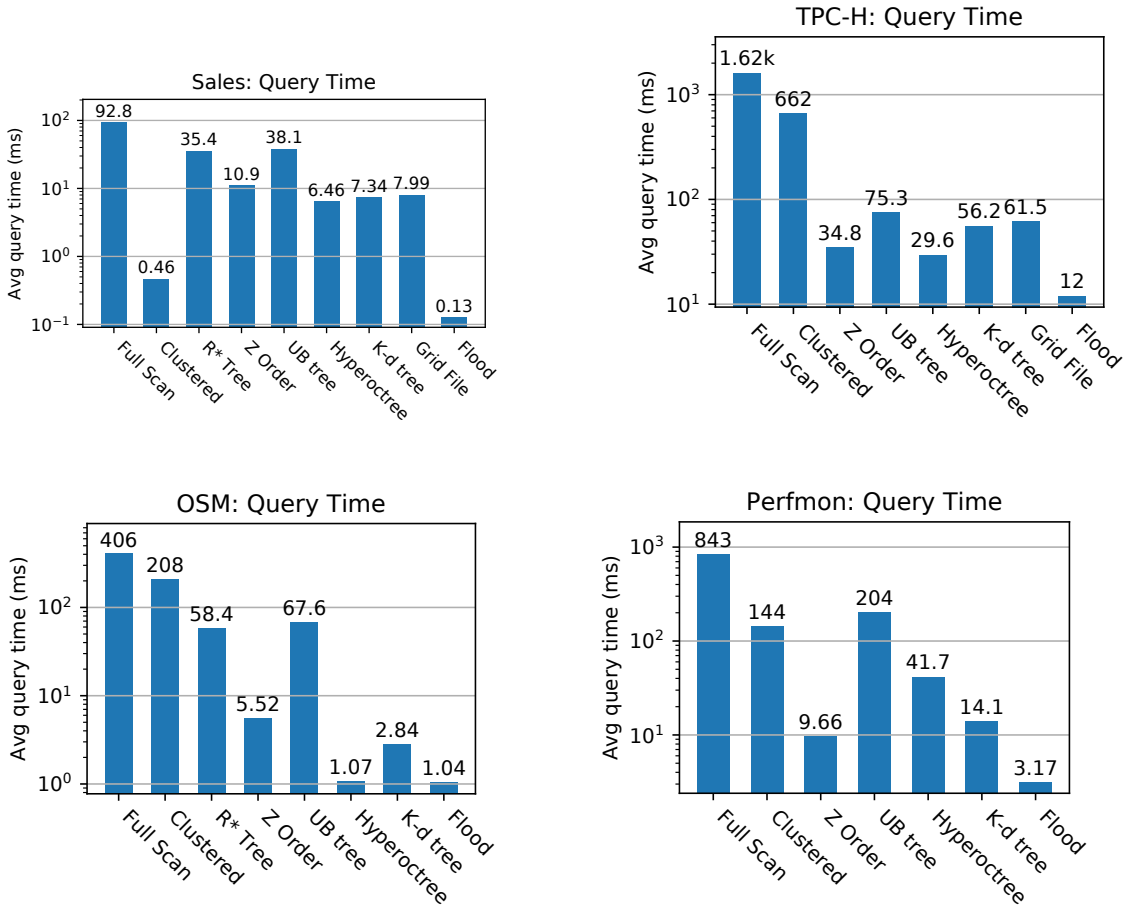


Figure 2-10: Query latency of Flood on all datasets. Flood's index is trained automatically, while other indexes are manually tuned for optimal performance on each workload. We exclude the R*-tree when it ran out of memory. Note the log scale.

by the OpenStreetMap [71] project in the US Northeast. All elements contain 6 attributes, including an ID and timestamp, and 90% of the records contain GPS coordinates. Our queries answer relevant analytics questions, such as "How many nodes were added to the database in a particular time interval?" and "How many buildings are in a given lat-lon rectangle?" Queries use between 1 and 3 dimensions,

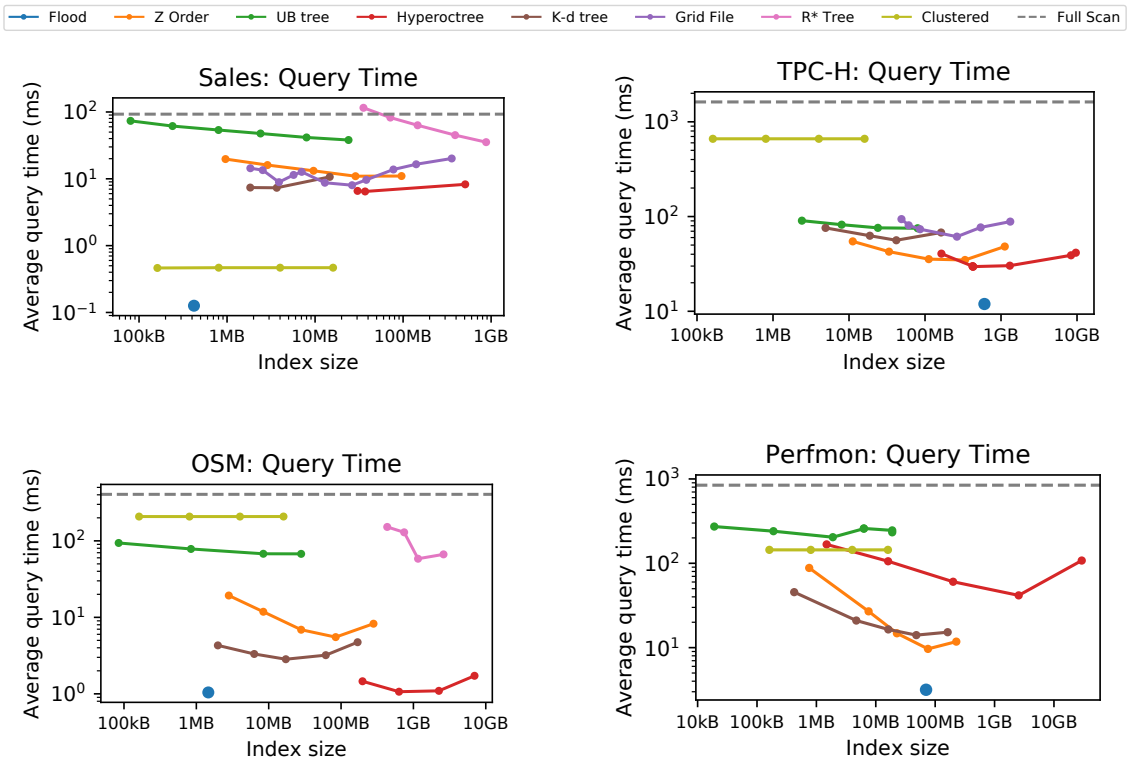


Figure 2-11: Flood (blue) sees faster performance with a smaller index, pushing the pareto frontier. Note the log scale.

with range filters on timestamp, latitude, and longtitude, and equality filters on type of record and landmark category. Each query is scaled so that the average selectivity is $0.1\% \pm 0.013\%$.

The performance monitoring dataset **Perfmon** contains logs of all machines managed by a major US university over the course of a year. Our queries include filters over time, machine name, CPU usage, memory usage, swap usage, and load average. The data in each dimension is non-uniform and often highly skewed. The original dataset has 23M records, but we use a scaled dataset with 230M records.

Our last dataset is **TPC-H** [88]. For our evaluation, we use only the fact table,

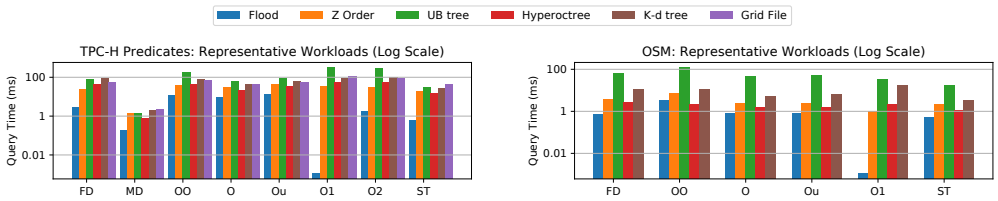


Figure 2-12: Flood and other indexes on workloads that have: fewer dimensions than the index (FD), as many dimensions as the index (MD), a skewed OLAP workload (O), a uniform OLAP workload (Ou), an OLTP workload over a single primary key (i.e., point lookups) (O1) and two keys (O2), a mixed OLTP + OLAP workload (OO), and a single query type (ST). Note the log scale.

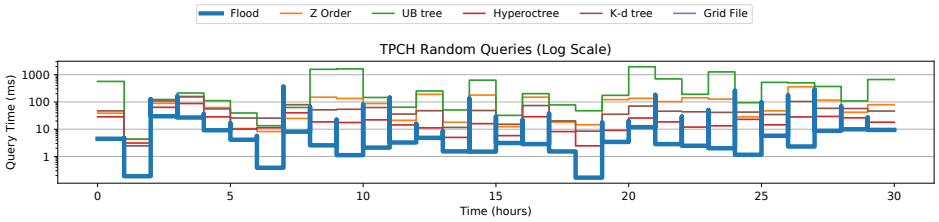


Figure 2-13: Flood vs. other indexes on 30 random query workloads, each for one hour. At the start of each hour, Flood's performance degrades, since it is not trained for the new workload; however, it recovers in 5 minutes on average once the layout is re-learned, and beats the next best index by 5× at the median. Note the log scale.

`lineitem`, with 300M records (scale factor 50) and create queries by using filters commonly found in the TPC-H query workload, with filter ranges scaled so that the average query selectivity is 0.1%. Our queries include filters over ship date, receipt date, quantity, discount, order key, and supplier key, and either perform a `SUM` or `COUNT` aggregation.

For each dataset, we generate a train and test query workload from the same distribution. Flood's layout is optimized on the training set, and we only report results on the test set.

2.8.4 Results

Overall Performance. We first benchmark how well Flood can optimize for a query workload compared to baseline indexes that are also optimized for the same query workload. Figure 2-10 shows the query time for each optimized index on each dataset. Flood uses the layout learned using the algorithm in Section 2.5, while we tuned the baseline approaches as much as possible per workload (e.g., ordered dimensions by selectivity and tuned the page sizes). This represents the best case scenario for the other indexes: that the database administrator had the time and ability to tune the index parameters.

On three of the datasets, Flood achieves between 2.4× and 3.3× speedup on query time compared to the next closest index, and is always at least on-par, thus achieving the best performance *across-the-board*. However, the next best system changes across datasets. Thus, depending on the dataset and workload, Flood can outperform each baseline by orders of magnitude. For example, on the real-world sales dataset, Flood is at least 43× faster than each multi-dimensional index, but only 3× faster than a clustered index. However, on the TPC-H dataset, Flood is 187× faster than a clustered index.

On every dataset, Figure 2-11 shows that Flood beats the Pareto frontier set by the other multi-dimensional indexes. In particular, even though Flood's performance on OSM is on par with the hyperoctree, its index size is more than 20× smaller. The hyperoctree thus has to spend much more memory for its performance than Flood. Flood's space overhead comes partially from the grid layout metadata, but mostly (over 95%) from the models of the sort attribute it maintains per cell.

Different Workload Characteristics. In practical settings, it is unlikely that a database administrator will be able to manually tune the index for every workload

change. The ability of Flood to automatically configure its index for the current query workload is thus a significant advantage. We measure this advantage by tuning all indexes for the workloads in Figure 2-10, and then changing the query workload characteristics to:

1. Single record filters, i.e. point lookups, using one or two ID attributes, as commonly found in OLTP systems.

2. An OLAP workload, similar to the ones in Figure 2-10, that answer reasonable business questions about the underlying dataset. Some types of queries occur more often than others, skewing the workload.

3. An OLAP workload where each query type is equally likely.

4. An equal split of workloads (1) and (2), i.e., combined OLTP and OLAP queries.

5. A workload with a single type of query, using the same dimensions with the same selectivities.

6. A workload with fewer dimensions (a strict subset) than indexed by the baseline indexes.

Figure 2-12 shows the potential advantages Flood can achieve over more static alternatives. Flood consistently beats other indexes, though the magnitude of improvement depends on the dataset and query workload. For example, on TPC-H, Flood achieves a speedup of more than 20× on half the workloads, while on OSM, the median improvement is 2.2×.

Dynamic Query Workload Changes. Here, we demonstrate how the performance of Flood varies over several random workloads, when the administrator does not tune the other indexes. We created 30 random workloads for the TPC-H dataset. Each

		SO	TPS	ST	IT	TT
Sales	**Full Scan**	644	3.09	92.6	0	92.8
	Clustered	3.18	3.09	0.462	6.76e-4	0.463
	Z Order	57.9	4.00	10.9	0.0161	10.9
	UB tree	55.7	14.5	38.0	0.0175	38.1
	Hyperoctree	38.8	3.34	6.11	0.353	6.46
	K-d tree	38.2	3.40	6.13	1.21	7.34
	Grid File	37.4	4.53	7.99	0.0594	7.99
	Flood	1.82	1.26	0.108	0.0182	0.128
TPC-H	**Full Scan**	965	5.27	1580	0	1620
	Clustered	447	4.71	655	7.15e-4	662
	Z Order	14.9	7.63	34.80	0.0267	34.8
	UB tree	15.3	16.1	75.2	0.0284	75.3
	Hyperoctree	20.8	4.38	27.8	1.77	29.6
	K-d tree	36.4	4.26	47.3	8.85	56.2
	Grid File	36.9	5.28	59.5	1.88	61.5
	Flood	5.90	5.53	9.96	2.02	12.0
OSM	**Full Scan**	1090	3.83	403	0	406
	Clustered	478	4.50	207	8.92e-4	208
	Z Order	6.85	8.37	5.5	0.0164	5.52
	UB tree	22.5	31.3	67.5	0.0171	67.6
	Hyperoctree	2.36	3.59	0.812	0.253	1.07
	K-d tree	6.60	3.51	2.22	0.611	2.84
	Grid File	N/A	N/A	N/A	N/A	N/A
	Flood	3.13	2.39	0.717	0.328	1.05
Perfmon	**Full Scan**	990	3.52	833	0	843
	Clustered	186	3.32	144	1.20e-3	144
	Z Order	9.08	4.42	9.64	0.0146	9.66
	UB tree	38.8	21.9	204	0.0120	204
	Hyperoctree	33.8	3.47	28.2	13.4	41.7
	K-d tree	15.7	3.07	11.6	2.51	14.1
	Grid File	N/A	N/A	N/A	N/A	N/A
	Flood	4.26	2.77	2.84	0.327	3.17

Table 2.2: Performance breakdown: scan overhead (SO), i.e. the ratio between points scanned and result size; average time (ns) scanning per scanned point (TPS); average time (ms) scanning (ST); average time (ms) indexing (for Flood this includes projection and refinement) (IT); total query time, in milliseconds (TT). SO×TPS is proportional to ST, and ST+IT≈ TT.R^*-tree omitted because instrumentation for collecting statistics was inadequate in [31].

workload runs for one hour and consists of at most 10 distinct query types, and each query type in turn consists of up to 6 dimensions, both chosen uniformly at random. The selectivities of each dimension are chosen randomly, with the constraint that all queries have an average total selectivity of around 0.1% and are more selective on key attributes.

Figure 2-13 shows the results over time with Flood being the only index that changes from one hour to the next (all others were kept fixed and tuned for the workload in Figure 2-10). At the start of each hour, a new query workload is introduced, and we trigger Flood's retraining. During the retraining phase, which we assume happens on a separate instance, Flood runs the new queries on its old layout, causing brief performance degradation and producing a spike at the start of each hour. It only switches to the new, more performant layout once retraining is finished. Flood outperforms all other indexes, showing a median improvement of more than 5× over the closest competitor, with 30% of queries achieving more than a 10× speedup. The results suggest that Flood is able to generalize well by adapting to new and unforeseen workloads.

Figure 2-13 also highlights the importance of learning from the query workload. When transitioning to the next query workload, Flood's performance often worsens, since the current layout is usually not suitable for the new workload. Re-learning the layout based on the new workload lowers query time back lower than other indexes. Learning a layout is therefore (a) effective at adapting to new query workloads and (b) crucial to Flood's performance improvement over other indexes. We leave the detection of workload changes to future work (Section 2.9).

While the results are encouraging, it is also important to consider the time it takes to adjust to a new query workload. Flood takes at most around 1 minute to adapt to a new query workload, but it more than makes up for this adjustment period through

improved performance on the subsequent workload. We evaluate index creation time in further detail in Section 2.8.7.

Performance Breakdown. Where does Flood's advantage over baseline indexes come from? We look at the *scan overhead*, the ratio of total points scanned by the index to points matching the query. The scan overhead is implementation agnostic: it relies neither on the machine nor on the implementation of the underlying column store. A high scan overhead suggests that the index wastes time scanning unnecessary points. Since all indexes spend the vast majority of their time scanning, the scan overhead is a good proxy for overall query performance.

Table 2.2 shows that Flood achieves the lowest scan overhead (SO) on three out of four datasets , which confirms that Flood's optimized layout is able to better isolate records that match a query filter. Additionally, Flood usually spends less time per scanned point (TPS) because Flood avoids accessing the sort dimension. As a result, Flood consistently achieves the lowest scan time (ST), which is proportional to the product of SO and TPS. This more than makes up for Flood's higher index time (IT), which includes the time to project and refine. Indexes based on Z-order must compute Z-values and thus have a higher time per scanned point. Tree-based indexes have the highest index time due to the overhead of tree traversal.

Which of Flood's components is responsible for its performance? Figure 2-14 shows the incremental benefit of (1) sorting the last indexed dimension instead of creating a d-dimensional histogram, (2) flattening the data instead of using columns of fixed width, and (3) adapting to the query workload using the training procedure from Section 2.5. The baseline system is a "Simple Grid" on all d dimensions, with the number of columns in each dimension proportional to that dimension's selectivity. (1) offers marginal benefits: it allows more columns to be allocated to the first $d-1$ dimensions, increasing the resolution of the index along those dimensions without

increasing the total number of cells. The biggest improvements come from (2) and (3). However, the effect of each varies across datasets. Flattening benefits OSM and Perfmon since both datasets have heavily skewed attributes. Since Sales and TPC-H data are fairly uniform, using a non-flattened layout performs equally well. Finally, learning from queries provides major performance gains on all datasets, corroborating results from Figure 2-13.

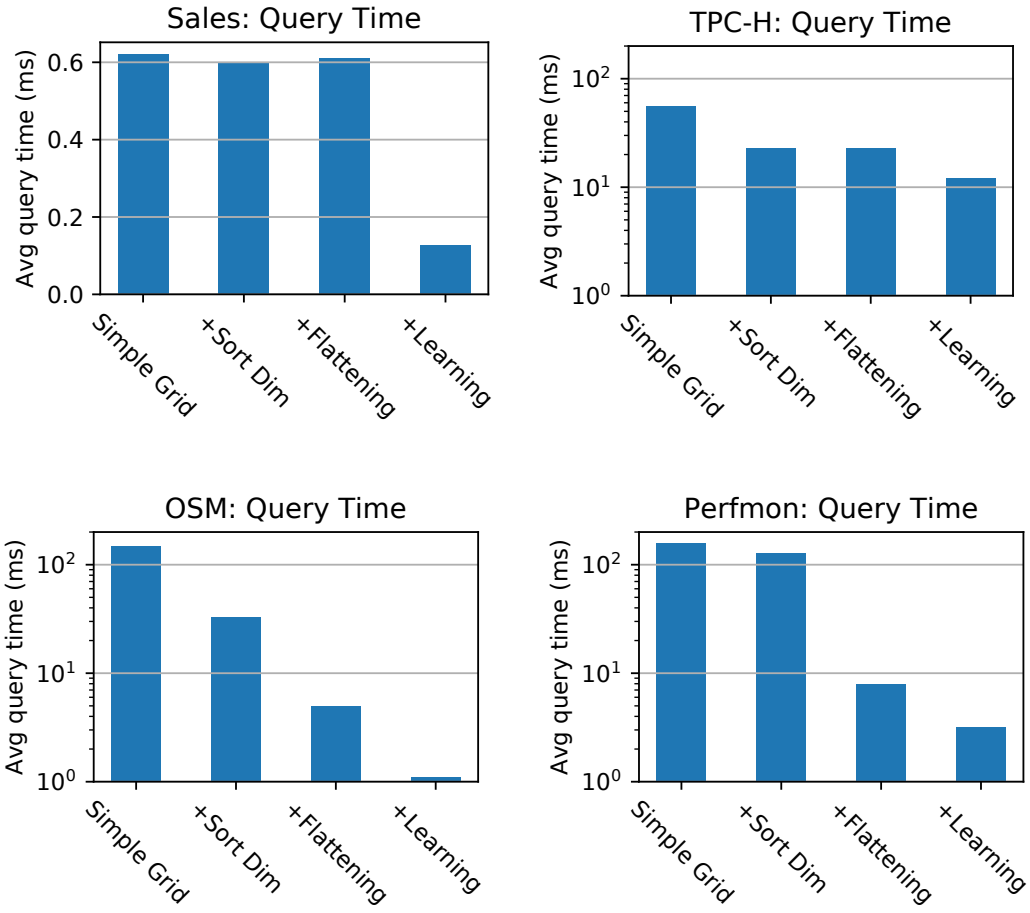
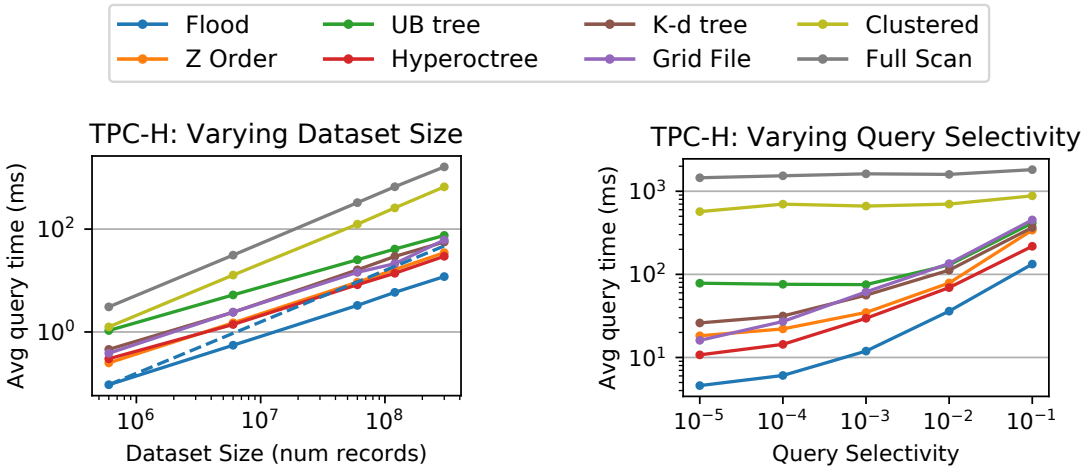


Figure 2-14: Flattening and learning help Flood achieve low query times but is workload dependent.

2.8.5 Scalability



(a) Breakdown by dataset size.

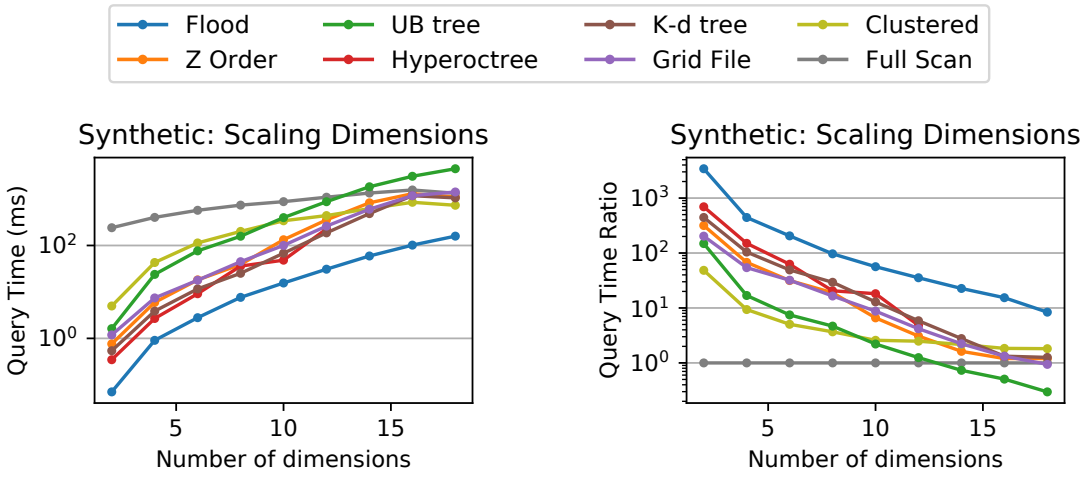
(b) Breakdown by query selectivity.

Figure 2-15: Flood's performance scales both with dataset size and query selectivity. The dashed blue line depicts what linear scaling would like like.

Dataset Size. To show how Flood scales with dataset size, we sample records from the TPCH dataset to create smaller datasets. We train and evaluate these smaller datasets with the same train and test workloads as the full dataset. Figure 2-15a shows that the query time of Flood grows sub-linearly. As the number of records grows, the layout learned by Flood uses more columns in each dimension, which results in more cells. The extra overhead incurred by processing more cells is outweighed by the benefit of lowering scan overhead.

Query Selectivity. To show how Flood performs at different query selectivities, we scale the filter ranges of the queries in the original TPC-H workloads up and down equally in each dimension in order to achieve between 0.001% and 10% selectivity. Figure 2-15b shows that Flood performs well at all selectivities. The performance benefit of Flood is less apparent at 10% selectivity because all indexes are able to

incur lower scan overhead when more points fall into the query rectangle.



(a) Absolute query time

(b) Throughput relative to a full scan.

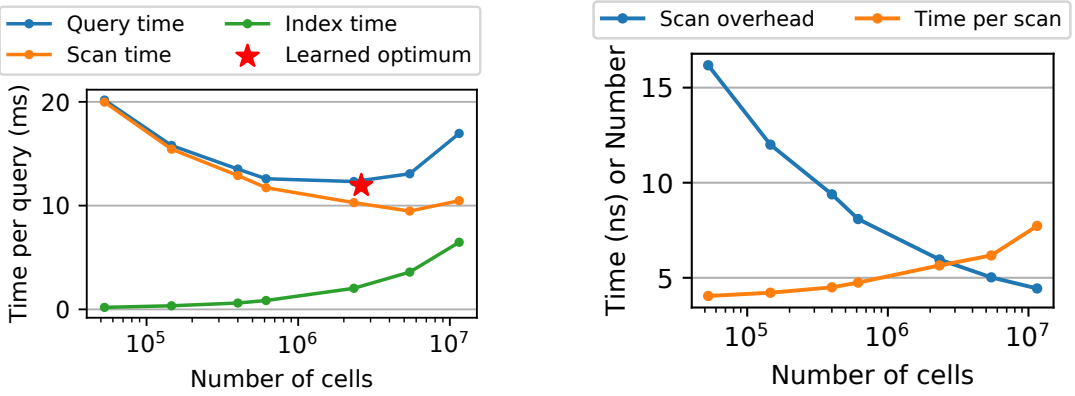
Figure 2-16: Query time, both absolute and relative to a full scan, as the number of dimensions varies.

Number of Dimensions. To show how Flood scales with dimensions, we create synthetic d-dimensional datasets ($d \leq 18$) of 100 million records with each dimension sampled from i.i.d. uniform distributions. For each dataset, we create a query workload of 1000 queries. The number of dimensions filtered in the queries varies uniformly from 1 to d. If a query has a filter on k dimensions, they are the first k dimensions in the dataset. For each query, the filter selectivity along each dimension is the same and is set so that the overall selectivity is 0.1%. For example, for a 2-dimensional dataset, 500 queries will select 0.1% of the domain of dimension 1, and 500 queries will select around 3.2% of the domains of dimensions 1 and 2.

Figure 2-16a shows that Flood continues to outperform the baseline indexes at higher dimensions. Note that the clustered index's relative performance also improves, since the baseline indexes spend resources on dimensions which are not frequently

filtered on. By contrast, Flood learns which dimensions to prioritize and excludes the least frequently filtered dimensions from the index on higher-dimensional datasets. Yet, Flood is also impacted by the curse of dimensionality (Figure 2-16b), which depicts the speedup of each index compared to a full scan. However, Flood can dampen this effect through its self-optimization, degrading more slowly than other indexes.

2.8.6 The Cost Model



(a) Number of Flood cells affects query time: there is an optimal point, denoted by the star.

(b) Though the scan overhead falls with more cells, there is extra time incurred in bookkeeping.

Figure 2-17: TPC-H: Adding cells reduces scan overhead but incurs a higher indexing cost and worse locality.

Finding the Optimum. Choosing an optimal layout requires balancing two competing factors: reducing the latency of locating both the relevant cells and physical index ranges within each cell (index time), and reducing the scan time by lowering scan overhead. Figure 2-17a illustrates this trade-off as the grid size changes (we fix a layout and scale the number of columns in each dimension proportionally): as the number of cells grows, scan time decreases because scan overhead decreases, but

		Layout learned for sales	tpc-h	osm	perfmon
Models trained on	**sales**	0.128	10.8 (-8%)	0.975 (-7%)	3.49 (+17%)
	tpch	0.132 (+3%)	11.7	0.986 (-6%)	3.18 (+6%)
	osm	0.134 (+5%)	11.7 (+0%)	1.05	3.14 (+5%)
	perfmon	0.137 (+7%)	11.6 (-1%)	0.964 (-8%)	2.99

Table 2.3: Query time (ms) when layouts are learned using cost models trained on different examples.

index time increases because there are more cells to process.

Figure 2-17a shows that Flood finds the number of cells that minimizes the total query time (red star), i.e., the best trade-off between scan time and index time. Note that we show the cost surface along only a single axis for visual clarity.

Robustness of the model. As per Section 2.5.1, our cost model needs to learn the weights $\{w_p, w_l, w_r, w_s\}$. This calibration step should happen once per dataset and machine. On our server, it took around 10 minutes, most of which was spent generating training examples. However, maybe surprisingly, the weights are robust to the data itself, so the cost model does not need to be retrained for every dataset. To show this, we trained our cost model on each of our four datasets, used each model to learn layouts for all four datasets, and then ran all 16 layouts on the corresponding query workloads. Table 2.3 shows that, no matter which dataset is used to learn the layout, the resulting layouts have similar performances, often with less than a 10% difference between them. Therefore, Flood can use the same cost model regardless of changes to the dataset or query workload. This makes calibration a one-time cost of 10 minutes.

	sales	tpc-h	osm	perfmon
Flood Learning	10.3	33.4	44.5	33.3
Flood Loading	4.12	29.6	8.03	22.0
Flood Total	14.4	63.0	52.5	55.3
Clustered	2.11	16.2	4.85	11.6
Z Order	7.82	86.7	24.9	72.6
UB tree	8.28	81.9	26.0	69.5
Hyperoctree	2.47	42.2	31.4	54.8
K-d tree	8.45	140	36.9	250
Grid File	10.6	121	N/A	N/A
R* tree	259	N/A	1340	N/A

Table 2.4: Index Creation Time in Seconds

2.8.7 Index Creation

Table 2.4 shows the time to create each index. We separate index creation time for Flood into learning time, which is the time taken to learn the layout (Section 2.5.2); and loading time, which is the time to build the primary index. The reported learning times use sampling of the dataset and query workload, described next. The total index creation time of Flood is competitive with the creation time of the baseline indexes.

Sampling records. Optimizing the layout using the entire dataset and query workload can take prohibitively long and does not scale well. However, Flood can reduce learning time without a significant performance loss by sampling the data. Figure 2-18 shows that Flood maintains low query times when estimating features with a 0.01–1% sample. This is because the main purpose of the sample is to estimate the number of records scanned per query. With selectivities around 0.1% or higher, a sample of 1% records is sufficient. Yet, this alone is not enough to match the creation time of the hyperoctree, the fastest of our multi-dimensional baselines.

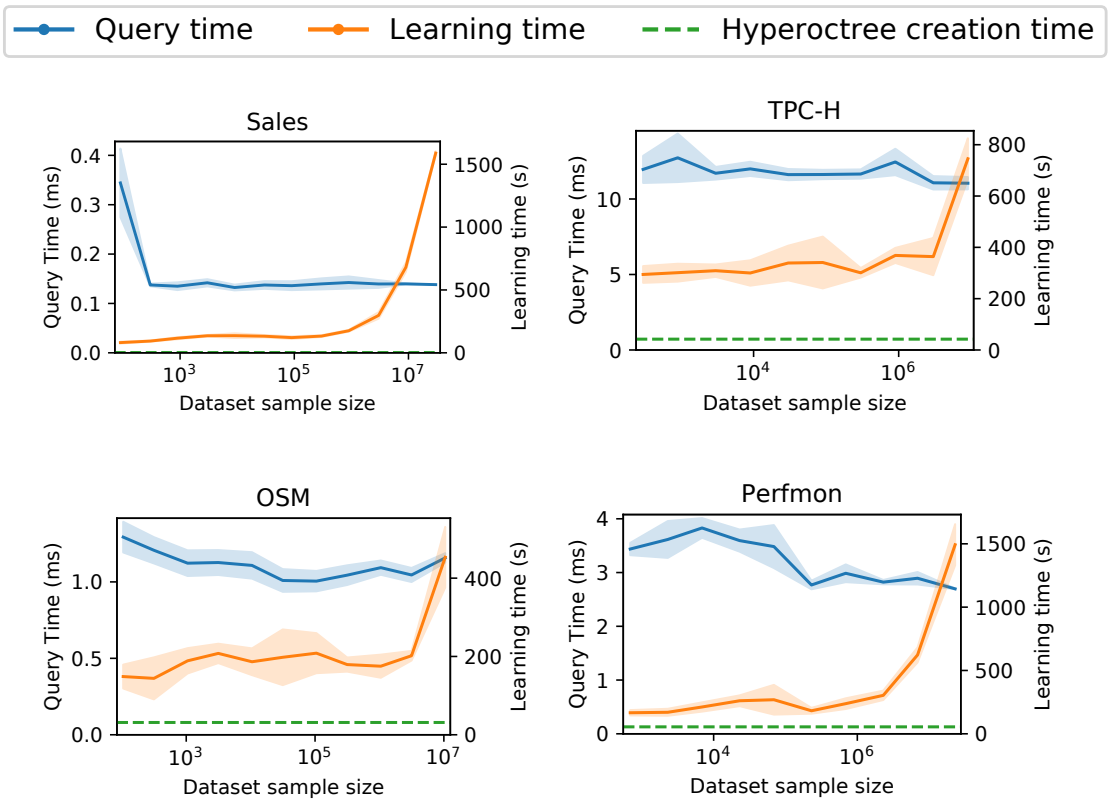


Figure 2-18: Learning time and resulting query time when sampling the dataset over several trials. One standard deviation from the mean is shaded. For comparison, we show the index creation time for the hyperoctree.

Sampling queries. Sampling the query workload can further reduce Flood's creation time. Here we conservatively use a data sample size of 100k records and vary the query sample size. As Figure 2-19 shows, Flood maintains low query times when using only 5% of queries. This is because the query workloads contain limited number of query types. Since queries within each type have similar characteristics with respect to selectivity and which dimensions are filtered, Flood only requires a few queries of each type to learn a good layout. However, the variance in performance increases as

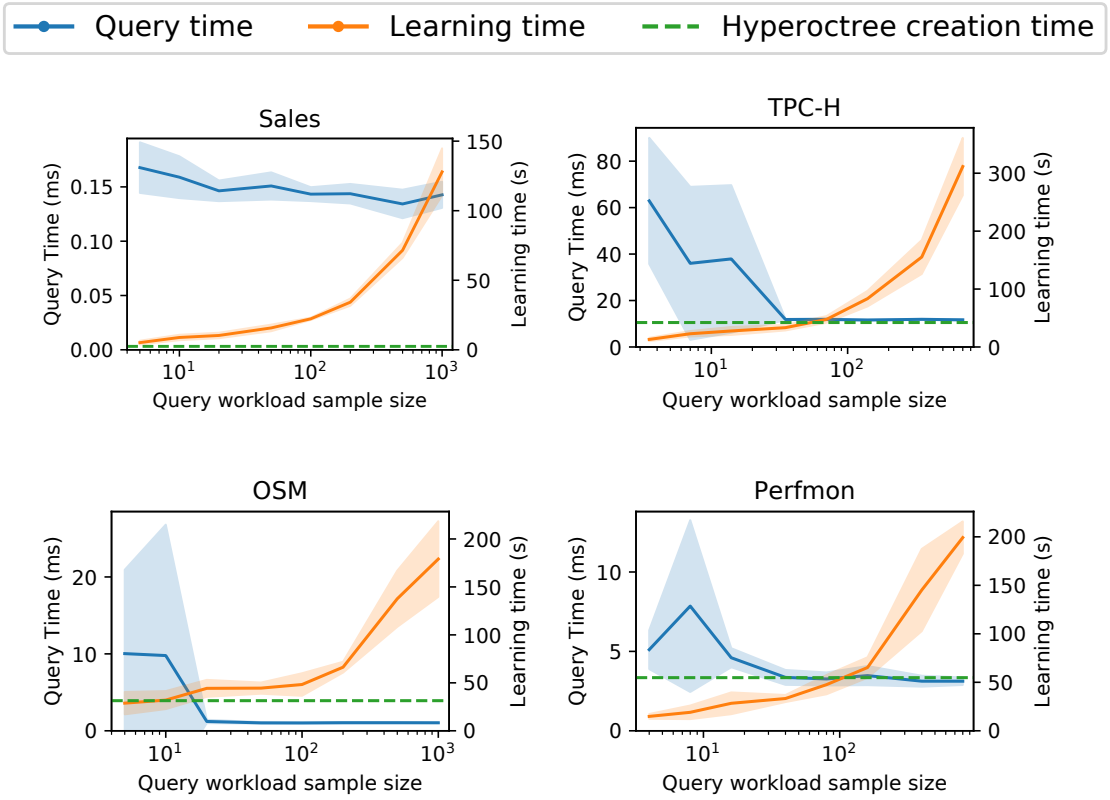
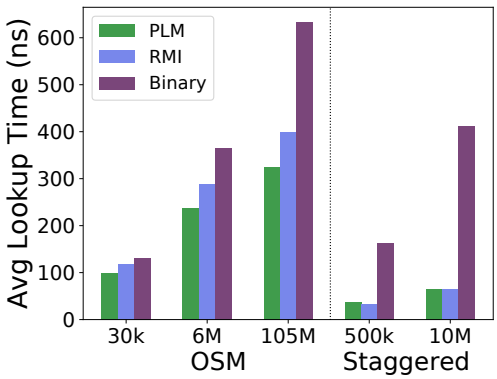


Figure 2-19: Learning time and resulting query time when sampling the queries over several trials. One standard deviation from the mean is shaded. For comparison, we show the index creation time for the hyperoctree.

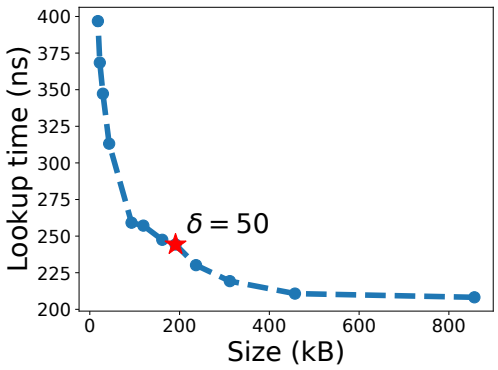
the query workload sample size decreases. With both optimizations (data and query samples), Flood achieves a learning time on par with the hyperoctree creation time without sacrificing performance.

2.8.8 Per-cell Models

Section 2.6.2 discusses CDF models to accelerate refinement. Since these CDFs are evaluated twice for each visited cell (beginning and end of the range), small speedups in lookup time may be noticeable on overall query time. Figure 2-20a benchmarks



(a) Three per-cell CDF models on two 1-D datasets.



(b) Size-speed tradeoff for the PLM, with our configuration marked.

Figure 2-20: Benchmarks for Flood's per-cell models.

the lookup time, including inference and an exponential search rectification phase, of three options we considered: the PLM (our approach), the learned B-tree [49], and binary search. We used real data (timestamps from the OSM dataset) and synthetic staggered uniform data (uniform over identically sized but disjoint intervals), with query points sampled from each dataset. The PLM and RMI perform comparably, and both beat binary search by up to 4×. We use the PLM since it requires only a single tuning parameter δ, which encodes the tradeoff between accuracy and size (Figure 2-20b). By contrast, the learned B-tree [49] requires extensive tuning of the number of experts per layer. The choice of δ depends on how much the administrator prioritizes speed over space: Figure 2-20b shows that $\delta = 50$ strikes a reasonable balance.

2.9 Future Work

Shifting workloads. Flood can quickly adapt to workload changes but cannot detect when the query distribution has changed sufficiently to merit a new layout. To do this, Flood could periodically evaluate the cost (Section 2.5) of the current

layout on queries over a recent time window. If the cost exceeds a threshold, Flood can replace the layout.

Additionally, Flood is completely rebuilt for each new workload. However, Flood could also be incrementally adjusted, e.g. by coalescing adjacent columns or splitting a column, or by incorporating aspects of incremental layout creation from database cracking [38], to avoid rebuilding the index.

Insertions. Flood currently only supports read-only workloads. To support insertions, each cell could maintain gaps, similar to the fill factor of a B+Tree. It could also maintain a delta index [81] in which updates are buffered and periodically merged into the data store, similar to Bigtable [11].

Concurrency and parallelism. Flood is currently single-threaded, but it can be extended to take advantage of concurrency and parallelism. Different cells can be refined and scanned simultaneously; within a cell, records can be scanned in parallel, allowing Flood to benefit from multithreading. Additionally, since Flood stores each column in the column store as a dense array, it can also take advantage of SIMD.

2.10 Conclusion

Despite the shift of OLAP workloads towards in-memory databases, state-of-the-art systems fail to take advantage of multi-dimensional indexes to accelerate their queries. Many instead opt for simple 1-D clustered indexes with bulky secondary indexes. We design Flood, a new multi-dimensional primary index that is jointly optimized using both the underlying data and query workloads. Learning from the query workload allows Flood to beat existing approaches: Flood outperforms optimally tuned state-of-the-art approaches by $30 - 400\times$, while using a fraction of the space. Our results suggest that learned primary multi-dimensional indexes can serve as useful building blocks in larger in-memory database systems.

Chapter 3

Cortex: Harnessing Correlations to Boost Database Query Performance

3.1 Background

Multi-dimensional indexes, including Flood, almost universally struggle when indexing even a handful of attributes. After around 5 attributes, most multi-dimensional indexes incur large scan overheads, scanning a significant portion of the dataset even if the query rectangle is small. This shortcoming is referred to as the *curse of dimensionality* and limits the effectiveness of multi-dimensional indexes in practice on datasets with several columns. Other approaches to indexing more columns, such as secondary indexes, are too restrictive: they have a large memory footprint and can only speed up queries that access a small number of records.

Most multi-dimensional indexes ignore *correlations* between columns, and therefore cannot leverage the fact that values in one column may provide information about another. We present Cortex, an approach that takes advantage of correlations to extend the reach of clustered indexes to more attributes, which has the potential to delay the curse of dimensionality: if columns A and B are correlated, an index that

includes column A might be able to achieve substantial pruning power on B without explicitly including B in the index. The index can therefore extend its "reach" to many additional correlated columns without incurring performance degradation from the curse of dimensionality. Note that Cortex cannot prevent the curse of dimensionality, but it allows additional columns to be filtered by the index, improving pruning power, without degrading the performance of the index as a whole.

Cortex harnesses correlations by dividing points into an inlier and outlier group, based on how strongly they adhere to the correlation in question, the selectivity of the query workload, the desired size−speed tradeoff of the index, and the relative latency of random versus sequential memory accesses in the underlying hardware. Cortex fits a linear performance model to compute this relative memory latency and determines outliers by analytically optimizing its model of scan overhead.

Unlike prior work, Cortex can adapt itself to any existing clustered index on a table, whether single or multi-dimensional, to harness a broad variety of correlations, such as those that exist between more than two attributes or have a large number of outliers. We demonstrate that on real datasets exhibiting these diverse types of correlations, Cortex matches or outperforms traditional secondary indexes with $5\times$ less space, and it is $2-8\times$ faster than existing approaches to indexing correlations.

3.2 Introduction

Data storage and access optimizations are at the core of any database system. Their main purpose is ensuring that records are efficiently looked up and filtered to maximize query performance. For example, many databases use clustered B-Tree indexes on a single attribute or – in the case of column stores – sort the data on a single column. As long as queries filter on the single clustered attribute, both techniques allow the database to return all relevant records without scanning the entire table.

For queries that do not filter on the clustered column, secondary indexes can help boost performance, but at a steep cost: they have a notoriously large memory footprint and incur high latencies due to their lookup cost and non-sequential access patterns [**?**, **?**]. This makes secondary indexes worth the high storage cost only if the queries touch a small number of records (i.e., have *low selectivity*).

As a result, multi-dimensional clustered indexes (and sort orders) are becoming increasingly popular, since they allow databases to improve the lookup performance for multiple attributes simultaneously without the storage or access overhead of a secondary index [95, 72, 16, 66, **?**, 36]. Unfortunately, multi-dimensional indexes typically do not scale past a handful of columns; the *curse of dimensionality* causes the performance of these indexes to quickly degrade as more columns are added [**?**, 66]. This limits the usefulness of multi-dimensional indexes on large tables, where query workloads may filter on many different columns.

This work explores how *correlations* between attributes can extend the reach of single or multi-dimensional clustered indexes to more attributes. The basic concept is to map a correlated column to a *host* column that is already part of the clustered index. Then, to answer a query on the correlated column, we can use the clustered index to filter the relevant ranges of the host column. This approach takes advantage of the locality in the data layout: since accessing ranges of sequential records (using the clustered index) is much faster than issuing point lookups (e.g., using a secondary index), we can in principle accelerate queries over the correlated column.

While encoding queries on one column in terms of a correlated column is not new [8, **?**], the benefits of doing so have been realized only for tight correlations. In real data, outliers weaken the strength of the correlation and force scans of large ranges on the host column, hurting performance. A natural question is then: *can we index outliers separately, leaving only the strong components of the correlation to be*

handled by the clustered index?

We present **Cortex**, an approach for indexing correlations that operates jointly with a clustered index (or sort order), leveraging it for fast range scans whenever possible while issuing point lookups to a small secondary index "stash" for outliers. Cortex's key idea is an algorithm for identifying outliers whose inclusion in the stash maximizes query performance. Cortex automatically adapts outliers to both the data's geometry and the underlying database. It uses a cost model informed by the scan and lookup performance, as well as the data layout, to select the optimal set of outliers.

Cortex's self-optimizing outlier detection algorithm helps it achieve better query times and handle more general types of correlations than previous approaches. For example, Correlation Maps (CMs) [8] do not detect outliers and can therefore only support strong correlations. In the presence of outliers, CMs scan a large fraction of the table and show poor performance. Hermit [**?**] can only handle outliers for strong (i.e., *soft-functional*) correlations between two real-valued attributes, a restrictive subset of the possible correlation types (Section 3.3). These shortcomings limit the benefits of prior solutions in real-world settings, where correlations in data are complex and varied.

To summarize, this paper makes the following contributions:

1. We design and implement Cortex, a system that indexes correlated columns by leveraging the table's clustered index. Cortex can use any type of clustered index or sort order, including those that are multi-dimensional. In line with the shift to in-memory databases [46], Cortex is implemented on top of a read-optimized compression-enabled in-memory column store, and supports insertions, updates, and deletions.

2. We introduce a new notion of an outlier that caters specifically to database performance, along with an algorithm to classify points as outliers using this definition. This algorithm allows Cortex to handle multiple types of correlations and adapt to different primary indexes.

3. We use three real datasets to evaluate Cortex against an optimized secondary B-Tree index and previous correlation indexes. We show that Cortex is the most performant index on a wide range of selectivities. Notably, it matches a B-Tree's performance on low selectivity queries while using more than $5\times$ less space, and outperforms it on high selectivity queries by more than $3\times$. Additionally, Cortex outperforms both CMs and Hermit across the board by $2-8\times$.

3.3 Related Work

Cortex's goal is to use correlations to boost query performance specifically by accelerating data access. There is a substantial body of work on discovering correlations in a dataset and using them for query plan selection or selectivity estimation [**?**, **?**, **?**]. Both are outside the scope of this paper: while an algorithm for finding correlations would complement Cortex by helping a DB admin choose attributes to index, our focus is on limiting the number of records accessed to reduce scan overhead at query time. In this section, we examine past work on indexing correlations, why using correlations to speed up queries is difficult, and the gaps that remain between the state-of-the-art and a practical, efficient solution.

Suppose a table has two columns, A and B, with an index on only B. In a typical RDMS, a database administrator wishing to handle queries over A might use a secondary index structure, such as a B-Tree [**?**], to map values of A to the location of the corresponding records. However, secondary indexes are well-known to be

extremely memory intensive and inefficient for even moderately selective queries, since the random point accesses they incur are much slower than scanning a contiguous range of records [**?**]. Efforts to reduce their memory footprint, such as compression, generally incur high overhead and deteriorate lookup performance [**?**, **?**]. Other approaches, like partial indexes, index only a subset of records, based on whether they satisfy a particular condition; however, these approaches only support simple predicates, have to be manually specified, and can only be utilized for a limited number of queries [**?**, **?**, **?**].

Another alternative is to capture A and B together in a multi-dimensional index. Many production systems employ multi-dimensional indexes [95, 72, 16], and recent work automatically tunes multi-dimensional indexes to retain fast performance on any type of query workload [66]. However, it is well known that this approach does not scale past a small number of columns, even if those columns are correlated [**?**]. This phenomenon is commonly referred to as the *curse of dimensionality* and afflicts all multi-dimensional indexes, including learned variants like [66] and [18]. Moreover, indexing additional columns worsens performance on queries over already indexed columns, a drawback that secondary indexes do not have.

Both secondary indexes and standard multi-dimensional indexes are unaware of the relationship between A and B, and so cannot capitalize on it. The typical solution to this problem is to harness the correlation by indexing one of the columns, say B, using a separate *host index*, which may or may not be a primary index, and map values of A onto B. At query time, the execution engine consults the mapping to figure out which values of B to scan.

Correlation Maps (CMs) [8] do precisely this: for every distinct value (or range of values) in A, a CM lists the values (or range of values) of B present in the table. CMs are guaranteed to produce values of B that are a superset of the ranges that

need scanning and work well for tight correlations with no outliers. However, they will trigger a scan of an entire primary index page if even a single point lies in it. As a result, their performance degrades substantially in the presence of outliers (see Section 3.6) [**?**].

Since most real datasets can reasonably be expected to have outliers, recent work has developed solutions to handle them without degrading query performance. BHUNT [**?**] finds algebraic constraints, where two columns are related to each other via simple arithmetic operations. However, this class of correlation is extremely restrictive. Hermit [**?**] further handles *arbitrary* soft functional dependencies by learning a piecewise linear function that maps A to B. In each piecewise component, a manually set error bound around the fitted function determines which records are considered outliers. The outliers are stored in a separate outlier index (a secondary B-Tree), while the inliers are mapped onto ranges of B directly, like a CM, and scanned using the host index.

Hermit suffers from two major drawbacks. First, it assumes the host index is single-dimensional by design. Trying to operate Hermit on a multi-dimensional index yields poor performance, since it does not take into account the index's complex partitioning of attribute values (Section 3.6). Second, it performs poorly on correlations that are not soft-functional (Section 3.6), due to its manual and heuristic-driven classification of outliers. Cortex solves this problem with an outlier detection scheme that calibrates itself to the data's geometry and the underlying system to maximize query performance, removing the need for heuristics.

3.4 Overview

Cortex adopts a hybrid design that consists of two key components (Figure 3-1): a dedicated outlier index using a secondary index structure that supports fast point

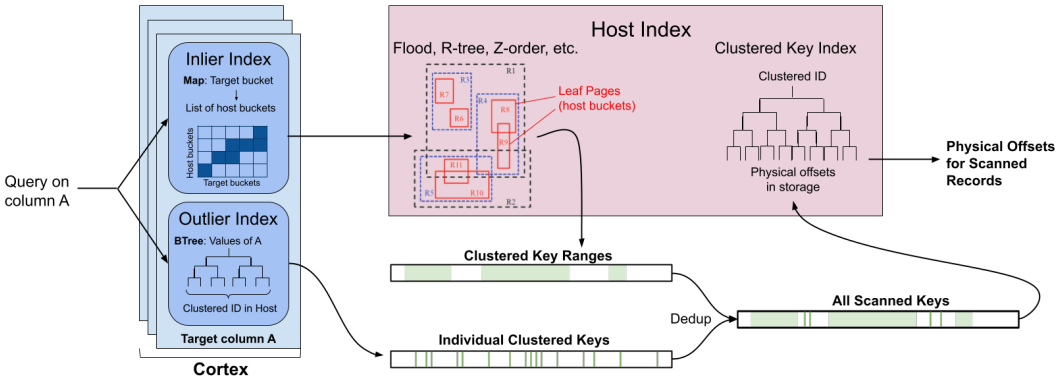


Figure 3-1: Query flow in Cortex. Cortex's inlier index indicates the leaf pages of the host index to scan, while the outlier index returns individual outlier records. After deduplicating the scanned pages with the outlier keys, keys are translated into physical offsets using the clustered index.

lookups (B-Tree), and a structure similar to a Correlation Map [8] to perform range scans. Cortex operates in conjunction with the clustered index (or *host index*) that already exists on the table, which indexes the *host columns* and dictates how the database is sorted in physical storage.[1] Each correlation in Cortex maps values of the column to be indexed, the *target column*, to a combination of one or more host columns. By combining the data structures from Correlation Maps and B-Trees, Cortex is able to avoid the pitfalls of each. In particular, while Correlation Maps alone cannot index outliers, a B-Tree for outliers fills this void. Conversely, while B-Trees alone would require substantial space overhead, using it *only for outliers* limits their memory footprint (Section 3.6). The following sections describe Cortex's design in detail, following the architecture shown in Figure 3-1.

[1]Cortex can also use secondary host indexes in case the clustered index does not contain the desired host columns; however, this approach will not take advantage of the clustered index's spatial locality.

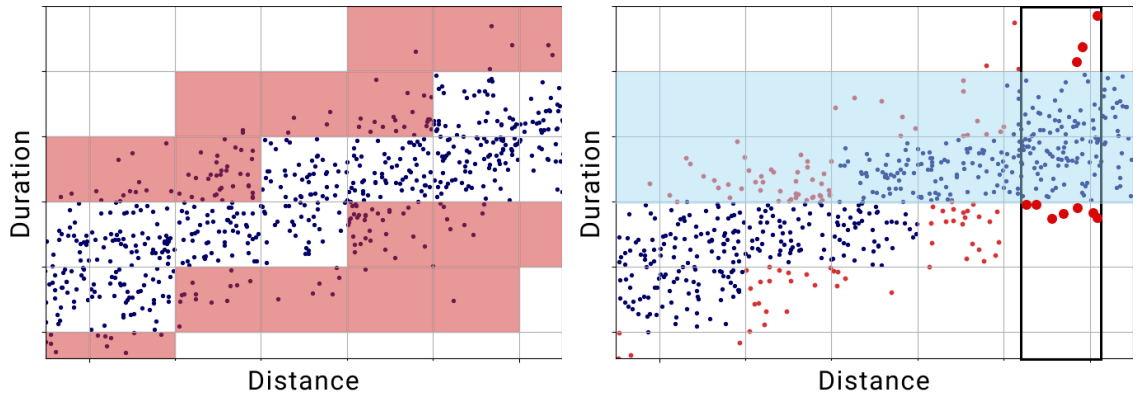


Figure 3-2: (a) Cortex assigns outliers (red area) by bucketing the host and target columns (grid lines). (b) For a query over the target column (bold box), Cortex scans the relevant inlier host buckets (blue region) and individually looks up outliers (large red dots).

3.4.1 The Host Index

Cortex works with any host index (single or multi-dimensional) that organizes records into pages, or *host buckets*. This method of organization is already the natural representation of tree-based or grid-based multi-dimensional indexes, such as octrees, kd-trees, and learned variants [66, 18], where the leaves themselves are the pages (host buckets). Note that since host buckets cover multiple columns in such indexes, they may be of different sizes and may not have a natural ordering. Likewise, most single-dimensional indexes partition data into pages of consecutive records. To maintain generality, Cortex assumes that if pages are accessed, the entire page is scanned. Cortex can also handle host indexes that sort records within a page, but it does not specifically optimize for such indexes.

Each record is given a key that is unique and orders records by their host buckets (or location in the sort order); for example, a key may consist of the clustered bucket ID with a unique identifier. A multi-dimensional index then comprises two parts (Figure 3-1): an algorithm, such as Flood [66], that maps a record to a host bucket

(which in turn corresponds to a contiguous range of clustered keys), and a clustered index, typically a B-Tree, that maps the clustered keys to physical locations in storage. The use of a clustered key as an intermediate is standard and necessary for efficient inserts [**?**].

Cortex is more versatile than prior approaches to indexing correlations, which assume the host index is single dimensional, because it can index correlations between a target column and *any subset* of the host columns. For example, in a table with column A correlated to B and C correlated to D, a host index on (B, D) allows Cortex to capture both correlations. This host index would also allow Cortex to capture any *multi-way correlations*, e.g., between another column F and (B, D) (or any other subset of host columns).

3.4.2 Indexing and Query Flow

To use Cortex, a database admin (DBA) specifies the target columns to index. This can be done using standard statistical techniques, such as evaluating mutual information with the host columns, and is outside the scope of this paper. Cortex consists of three data structures per target column: an inlier index, outlier index, and correlation tracker. The inlier and outlier indexes participate in query execution, while the correlation tracker maintains the information necessary to handle updates.

At a high level, Cortex divides the points in a host bucket into several *target buckets* based on the values of the target attribute, detailed further in Section 3.5.2. The target and host buckets partition the table, with each record belonging to a single (target bucket, host bucket) pair, or *cell*. Figure 3-2 shows the cells for an example correlation in a flight dataset, where the target column ('Distance') is correlated to the single host column ('Duration'). Cortex then runs its custom *stashing algorithm* to determine which cells are inliers and outliers (Section 3.5.3). In Figure 3-2a, the

outlier cells are highlighted in red.

Inliers. The inlier cells are indexed by a structure resembling a Correlation Map [8]: the inlier index maps each target bucket t_i to a list of host buckets h_j for which the cell (t_i, h_j) is an inlier. At query time, if the query has a filter over the target column, Cortex scans the matching host buckets, i.e., the union of all host buckets that are inliers for the target buckets that overlap with the filter. Specifically, for a range R on the target column, Cortex will scan all points in $\{h \mid \exists t \text{ s. t. } (t, h) \text{ is an inlier and } t \cap R \neq \emptyset\}$. In the example in Figure 3-2b, these scanned points correspond to the 'Duration' buckets highlighted in blue.

A host bucket h can include points matching multiple target buckets. If at least one of these target buckets intersect the query range R, Cortex scans all of h, potentially scanning unnecessary points. The inlier index therefore has a low storage footprint, but this simplicity comes at the cost of a higher scan overhead.

Outliers. For each correlation, each point in an outlier cell is indexed on its target column value by a secondary B-Tree index. The outlier index makes the opposite tradeoff from the inlier index: it requires a larger space overhead to maintain but guarantees that the only outliers scanned are those in target buckets touched by the query filter, avoiding superfluous lookups. In Figure 3-2b, all the red dots (small and large) are added to the outlier index.

Cortex then processes a range query over the target column in four steps, summarized in Figure 3-1:

1. Query the inlier index to find the host buckets h_j that intersect with the target buckets touched by the range (the blue buckets in Figure 3-2b). Each host bucket corresponds to a single range of clustered keys.

2. Query the outlier index to find the individual clustered keys for points that

belong to the same target buckets (the large red dots in Figure 3-2b).

3. Deduplicate the results from (1) and (2). This is a superset of all points that match the query filter. Deduplication is necessary because inliers and outliers are only guaranteed to be unique *per target bucket*. If a query covers multiple target buckets, the outliers of one bucket may fall within the inlier ranges of another.

4. Scan the given ranges and individual points, pruning out those that do not match the query filter.

Correlation Tracking. As records are inserted into or deleted from the table, Cortex's determination of which cells are inliers and outliers will change. Instead of running the stashing algorithm anew on the entire dataset, it maintains a small data structure to track when each cell needs to be reclassified. The tracker also handles changes to the host index as a result of updates, e.g. if a host bucket is split into multiple children or if many buckets are merged into one. The tracker's operation is detailed in Section 3.5.4.

3.4.3 Outlier Detection

To index a correlation, Cortex uses the leaf pages of the host index, or host buckets, as a natural partition of the host columns' values. Therefore, Cortex's outlier stashing algorithm depends on two important aspects of the underlying host index:

1. The host bucket boundaries. The cost of an inlier cell is that every query that touches that cell must scan the corresponding host bucket. Since Cortex's stashing algorithm considers this cost in its optimization, its selection of outliers depends on the host buckets.

2. The time spent scanning a page from the host index relative to an outlier lookup in the stash. The faster a sequential scan on the host index is, the more Cortex should rely on the host index and the fewer records Cortex should assign as outliers. This determination depends on multiple facets of the host index, including how large the host buckets are, the indexing overhead, if data is compressed, row vs. column store, etc.

As a result, Cortex's outlier detection algorithm, detailed in Section 3.5, must be closely tied to the underlying system and host index properties. Using off-the-shelf outlier detection algorithms, or simple heuristic-based thresholds like in Hermit, are unable to capture the complexities specific to the host index. We explore the impact of using these untuned approaches in Section 3.6.

3.5 Stashing Algorithm

In this section, we describe Cortex's algorithm for deciding the *outlier assignment* for a single correlation; that is, which (target bucket, host bucket) cells to classify as outliers. If a cell is an outlier, all the points in that cell are stashed in the outlier index. We discuss how Cortex trades off between space and speed (Section 3.5.1), decides on target buckets (Section 3.5.2), formulates and optimizes the cost model (Section 3.5.3), and adapts the outlier assignment in the presence of insertions and deletions (Section 3.5.4). Table 3.1 describes the key parameters used in the following sections.

3.5.1 Speed vs. Space

Choosing the number of outliers is a tradeoff between space overhead and query speed: stashing a record in the outlier index may reduce scan overhead but increases the memory footprint of the stash. Navigating this tradeoff requires some form of

storage constraint by the DBA, which Cortex allows in one of two forms:

1. A hard space limit (equivalently, a maximum number of outliers) that Cortex will not exceed when choosing outliers. This is an intuitive constraint but cannot automatically adapt itself to varied correlation types.

2. The relative value of storage and performance, i.e. how much of a performance improvement would you like in exchange for using extra storage?

Cortex supports both formulations, but this section will focus primarily on (2), since we believe that its adaptability to adjust to different strengths of correlations is valuable.

Cortex captures the relative value of storage and performance in a parameter α, set by the DBA and defined as the percentage improvement in scan time relative to T_0 (the scan time when all cells are classified as inliers) that is equivalent to a 1% increase in storage for the DBA, relative to the dataset size D. This is an intuitive definition: if $\alpha = 10$, Cortex would stash 1% of points as outliers only if doing so improved performance by more than 10%. Cortex's cost function is then:

$$\text{Cost} = (\text{query time}) + \alpha \cdot \frac{T_0}{D} \cdot (\text{space}) \tag{3.1}$$

A large value of α selects very few outliers and thus does not improve query speed substantially, while setting $\alpha = 0$ does the opposite, but requires more space for the outlier index. In practice, we approximate the query time T_0 with the scan overhead, i.e. the number of points scanned. This works because the scan overhead can be represented algebraically in terms of known constants and scales linearly with scan time. Note that α is Cortex's only user-defined parameter.

Cortex's optimization consists of two parts: (a) choosing the target buckets, and

Parameter	Description	How is it set?
N_t	Number of target buckets	Automatic
α	Space / speed tradeoff	User-defined
β	Cost of non-sequential lookup	Automatic

Table 3.1: The parameters in Cortex's optimization problem.

(b) given the target and host buckets, deciding which cells are outliers. Ideally, both would be co-optimized; however, co-optimization is complex and, in our exploration, took too long to be practical. Therefore, Cortex first optimizes the number of target buckets independent of the outlier assignment; once the target buckets are fixed, it then assigns outlier cells.

3.5.2 Choosing Target Buckets

Choosing the number of target buckets for Cortex to use is also a tradeoff between size and speed; having more target buckets requires more metadata and adds to the index size but lets Cortex prune out records with higher granularity. Cortex aims for targets buckets that contain approximately the same number of records.

Since Cortex calculates the number of target buckets independently from the outlier assignment, it does not have any information about outliers. In this setting, we can formulate the choice of target buckets T as an optimization problem that depends on α. Suppose there are t target buckets and H host buckets, producing $t \cdot H$ total cells. Let the selectivity of query q in the target column be $s_q \in (0, 1)$, defined as the fraction of the table that matches the query. It can be shown[2] that in this setting, when $s_q \ll 1$, the expected scan overhead (the ratio of points scanned to result size) of a query with selectivity s_q, chosen uniformly at random over the target column's value range, is approximately $1 + \frac{1}{ts_q}$. The average scan overhead of a set of

[2]See Appendix A for a proof.

queries Q is then:

$$S(t) \approx \frac{1}{|Q|} \sum_{q \in Q} 1 + \frac{1}{ts_q} \tag{3.2}$$

We validate this approximation experimentally in Section 3.6.5. Let m be the metadata (in bytes) required for each cell. Then Cortex's bucket optimization problem is:

$$\min_t S(t) + \alpha \cdot \frac{S(1)}{D} \cdot mtH$$

We solve analytically to get the optimal number of target buckets:

$$N_t = \sqrt{\frac{D}{\alpha m H |Q| S(1)} \sum \frac{1}{s_q}}$$

3.5.3 Outlier Assignment

Once the target buckets have been decided, Cortex must now determine which points are outliers. For this, Cortex must understand the cost of stashing an outlier. Cortex captures this cost with a parameter β that encodes the average cost of a non-sequential point lookup, relative to the cost incurred by a point accessed via a range scan. This includes the time required to deduplicate, look up the point's unique clustered key from the secondary index, and fetch the corresponding value from storage using the clustered index. Unlike α, β is determined automatically: Cortex runs 1000 queries with various selectivities, and fits a linear model of the form:

$$\text{Query time} = c_1(\#\text{ records in range scans}) + c_2(\#\text{ point lookups}) + c_3$$

Then, $\beta \equiv {}^{c_2}/_{c_1}$. This formulation makes the simplifying assumption that the query time has a linear dependence on the number of scanned points, which we found to be

accurate in practice: the correlation coefficient of the resulting fit is $R^2 \gtrsim 0.97$. Note that c_3 is a fixed cost incurred regardless of outlier assignment, so it has no effect on this optimization problem.

Tuning β automatically is how Cortex adapts itself to varied host indexes and storage formats. For example, a compressed table may hurt point lookup performance more than range scans; this would be reflected in a larger β. Additionally, if the host index is a secondary index, then $\beta \approx 1$, reflecting that there is no significant advantage to range scans over the host as compared with point lookups in the the outlier stash.

With the target column partitioned into buckets, Cortex computes the number of points per cell. We denote the number of points in a cell (t, h) as $|(t, h)|$. Let $\mathcal{B}$ be the set of all (target bucket, host bucket) cells that contain at least one point. Define an outlier assignment as $\mathcal{A} = \{\mathcal{O}, \mathcal{I}\}$, where $\mathcal{O} \subset \mathcal{B}$ are the outlier cells, and $\mathcal{I} = \mathcal{B} \setminus \mathcal{O}$ are the inlier cells. We use $\mathcal{I}(t)$ and $\mathcal{I}(h)$ to denote the inlier cells with target bucket t or host bucket h, respectively, and likewise for outliers. It will be clear from context which is being used. Cortex formulates the cost of $\mathcal{A}$, denoted $C(\mathcal{A})$, and chooses $\mathcal{O}$ to minimize this cost. Initially, all cells are considered inliers.

$C(\mathcal{A})$ comprises two terms: one for performance overhead and one for space overhead. Space overhead is simply the total number of points in the outlier index:

$$\mathrm{SO}(\mathcal{A}) = \sum_{(t,h)\in\mathcal{O}} |(t, h)|$$

Performance overhead for a query that intersects target bucket t is the cost of (a) doing a range scan over the inlier points for that query, namely all the points in the host buckets h for which (t, h) is an inlier, and (b) doing point scans on all the outlier

points in t. In other words:

$$PO(\mathcal{A}, t) = \sum_{h \in \mathcal{I}(t)} |h| + \beta \sum_{h \in \mathcal{O}(t)} |(t, h)|$$

where $|h|$ refers to the number of points in host bucket h. The total performance overhead is then $PO(\mathcal{A}) = \sum_t PO(\mathcal{A}, t)$.

Before putting everything together, recall that α was defined as the *percentage* decrease in performance overhead that is equivalent in value to a 1% increase in storage overhead. In other words, α is defined using relative changes in performance and storage overhead, so it needs to be scaled appropriately before assembling the complete cost function. Let s be the per-item space overhead of the outlier index, and let $P_0 = PO(\{\emptyset, \mathcal{B}\})$ be the *initial scan overhead*, i.e. the number of records scanned, assuming all cells are inliers (recall from Section 3.5.1 that P_0 is an approximation for T_0). The total cost from Equation 3.1 is then:

$$\begin{aligned} C(\mathcal{A}) &= \sum_t \left[\sum_{h \in \mathcal{I}(t)} |h| + \beta \sum_{h \in \mathcal{O}(t)} |(t, h)| \right] + \alpha \frac{P_0 s}{D} \sum_{(t,h) \in \mathcal{O}} |(t, h)| \\ &= \sum_t \left[\sum_{h \in \mathcal{I}(t)} |h| + \left(\beta + \alpha \frac{P_0 s}{D} \right) \sum_{h \in \mathcal{O}(t)} |(t, h)| \right] \end{aligned} \tag{3.3}$$

Since host buckets do not change as points are stashed, this problem is independent for each target bucket, and can be solved easily: a cell should be an outlier if the first summand in Equation 3.3 is larger than the second, and an inlier otherwise. Specifically, Cortex assigns (t, h) to be an outlier if:

$$\left(\beta + \alpha \frac{P_0 s}{D} \right) |(t, h)| < |h| \tag{3.4}$$

The fact that the solution to this optimization problem is independent for each target bucket is particularly important, since it means that Cortex's outlier assignment is correct *regardless of query distribution.* Any query intersecting target bucket t will maximize $C(\mathcal{A})$ using the above assignment, independent of which other buckets it intersects. Note, however, that this is not true if the storage constraint takes the form of a hard cap, instead of the parameter α. In this case, target buckets would need to be weighted by how frequently they are covered, to properly prioritize those that reduce cost the most.

3.5.4 Handling Inserts and Deletions

Cortex supports dynamic updates while maintaining the correctness of its outlier assignment. For brevity, we will focus this section on inserts; deletions and updates operate analogously.

As records are inserted, Cortex's correlation tracker must keep track of when a cell needs to switch from inlier to outlier or vice versa. In addition, it is possible, even likely, that the host buckets will change: a page may be split into multiple smaller pages if filled past a certain capacity, or merged into another cell if sufficiently empty. Cortex needs to handle these changes to the underlying host index as well as changes in cell assignments. Note that the target buckets remain largely unchanged, except for when points appear outside the existing range of the target column. If a target bucket becomes disproportionally crowded due to skewed inserts, it may be split into two. However, this isn't required for correctness of the outlier assignment.

Figure 3-3 illustrates the workflow for insertions. Cortex passes all insertions through the host index, and requires that the host index return the clustered keys of the new points, as well as a pointer to (or ID of) the host bucket it was added to. With this information, Cortex updates its internal count of points in each host

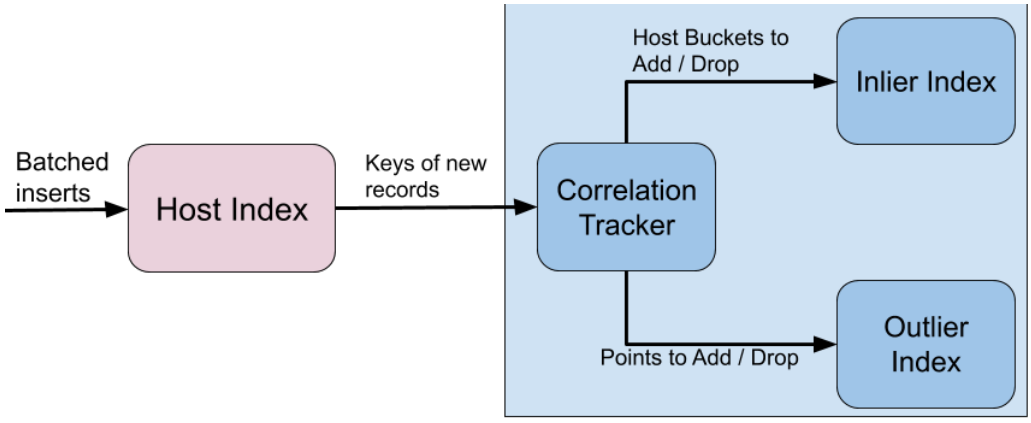


Figure 3-3: Cortex's insertion flow.

bucket and determines whether any cells in that host bucket have a new assignment. Any of the following may happen:

The outlier assignment is unchanged. Some inserted points belong to inlier cells, and no action needs to be taken for them. The inlier index will include them when it scans the corresponding host bucket. Points that fall into outlier buckets must be inserted into the outlier index. Unlike a traditional secondary index, only a fraction of points typically are outliers and need to be inserted.

A cell (t, h) changes from outlier to inlier. Cortex adds h to the entry for t in the inlier index. In addition, the tracker signals to the outlier index that the points lying within t's value range and h's clustered key range must be removed. This operation scales with the number of outlier points in target bucket t.

A cell (t, h) changes from inlier to outlier. This is the most costly operation. Cortex must scan the host bucket h in the underlying table to find the physical location of the records that are also in t; these points are all inserted into the outlier index based on their target column value. The entry for (t, h) is then removed from the inlier index.

	Records	Columns	Correlations	Host Index	Host Columns	Target Columns
Stocks	165M	7	Linear functional	1-D Clustered	Daily Open	Daily High, Daily Low, Daily Close
				Octree, Flood	Date, Daily Open	Daily High, Daily Low, Daily Close
Chicago Taxi	194M	9	Weak functional Non-functional Multi-way	1-D Clustered	Total Fare	Distance, Metered Fare, Tips
				Octree, Flood	Start time, Duration, Total Fare	End Time, Distance, Metered Fare, Tips
WISE	198M	15	Weak functional Non-functional Multi-way	1-D Clustered	W1 Magnitude	W1 σ, W1 SNR
				Octree, Flood	RA, Declension, W1 Magnitude	Galactic Lon/Lat, Ecliptic Lon/Lat, W2 Magnitude, W1 σ, W1 SNR

Table 3.2: Three real datasets used in our evaluation. We evaluate each dataset on both single dimensional and multi-dimenisonal host indexes; each setting uses different host and target columns. See Section 3.6.2 for further details on correlation types.

A host bucket is split or merged. This case occurs least frequently of the four. Cortex must recompute the new cells for all points in the original host bucket(s) and recluster the underlying records. Since there are often a large number of host buckets, this operation typically touches only a small fraction of the data.

If the inserts have the same distribution as existing records, cells switching between inlier and outlier will be rare. Inserts in order of clustered key are also fast since they will update fewer host buckets. We evaluate Cortex's insert performance in Section 3.6.6.

3.6 Evaluation

This section benchmarks Cortex against various other correlation index baselines, as well as a secondary B-Tree index. Our results demonstrate that across our real datasets, and across both single and multi-dimensional host indexes, Cortex can outperform the B-Tree while occupying $5 - 70\times$ less space. At the same time, Cortex outperforms prior approaches by $2 - 8\times$ on a variety of datasets and workloads.

3.6.1 Implementation

Cortex is implemented in C++. We run all experiments on an AWS r5a.4xlarge instance, with 16 2.2GHz AMD EPYC 7571 processors and 64GB DRAM. All experiments use 64-bit integer-valued attributes. Floating point values are multiplied by the smallest power of 10 that makes them integers, and categorical variables are assigned integer values using a dictionary encoding [**?**]. Unless indicated otherwise, each query in our query workloads has a range filter over a single target column, and returns the IDs of the records that match the filter[3]. Each query range is sampled uniformly at random from the value range of the target column. Each query workload has a target selectivity $s = \frac{\text{points in result}}{\text{table size}}$. Target selectivities are approximate, and the actual selectivity of queries in the workload lies within $[\frac{s}{2}, 2s]$. To clear ambiguity, we say that workloads with larger s have *higher selectivity* than those with smaller values of s.

Cortex runs on a custom in-memory column store with a bit-packing compression scheme, in which the value in each compression block is encoded as a low bit-width offset from the block's base value. With this setup, Cortex measures $\beta = 17.88$ (the relative cost of a point lookup vs. a range scan; see Section 3.5.3) with correlation coefficient of $R^2 = 0.97$. We use this column store for all baselines:

1. *Full Scan* scans all records in the table.

2. *Secondary* builds a B-Tree secondary index for every column in the query workload. The B-Tree returns physical pointers to all matching records. The B-Tree we use is cache-optimized with 256-byte nodes [**?**].

3. *CM* builds a Correlation Map for every column in the query workload, adapting

[3]We find that Cortex maintains similar relative performance to the baselines regardless of which aggregation is used.

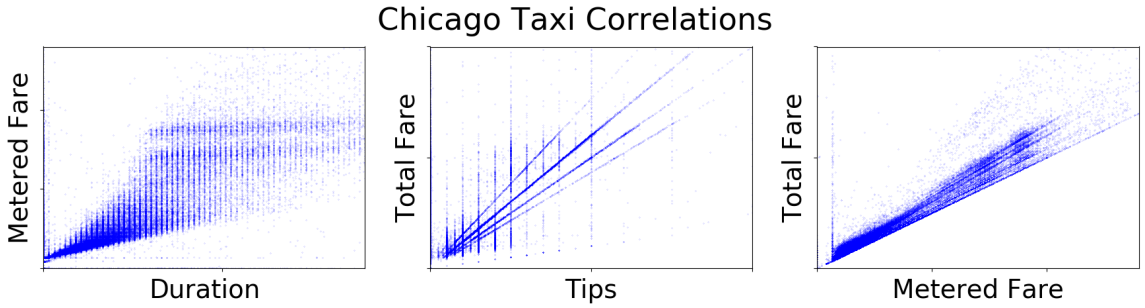


Figure 3-4: Examples of weak and non-functional correlations in the Chicago Taxi dataset. Some outliers may not be easily visible.

it to a multi-dimensional setting by mapping buckets along the target dimension to the minimal set of host buckets that are guaranteed to contain all matching points. The CM blocks along the target dimension are chosen identically to Cortex's target buckets.

4. *Hermit* implements the TRS-Tree from Hermit [**?**], which consists of four manually set parameters: the error bound, which corresponds to an inlier confidence interval for each piecewise linear segment, a maximum outlier fraction, and the maximum depth and fanout of the TRS Tree. We use the recommended parameters from [**?**]: 2, 0.1, 10, and 8, respectively. Since Hermit can only map the target column to a single host column, in situations with a multi-dimensional host index, we choose the host column with the highest correlation to the target column being indexed.

5. *Cortex* is our solution for $\alpha = 1$ unless otherwise specified. We found empirically that this value of α produces a good tradeoff between performance and speed across datasets and workload selectivities. We evaluate the effect of changing α later in this section.

We evaluate Cortex and all baselines on three host indexes:

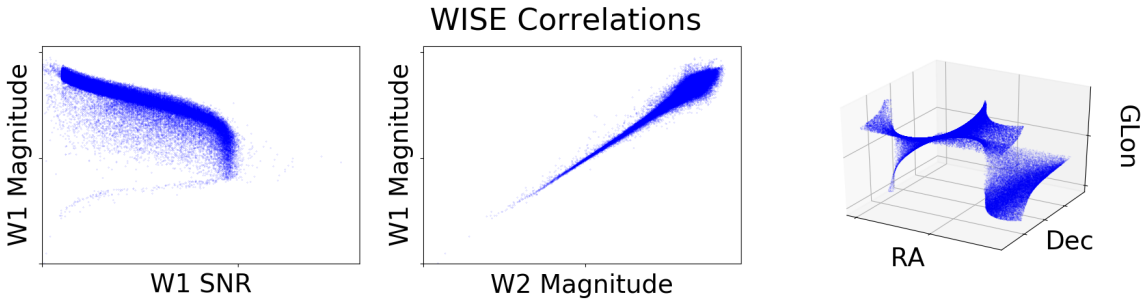


Figure 3-5: Examples of non-functional, weak, and multi-way correlations in the WISE dataset. Some outliers may not be easily visible.

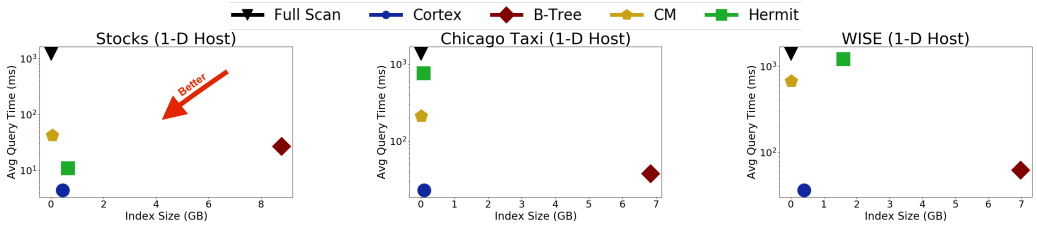


Figure 3-6: Performance of Cortex and baselines on three datasets using a clustered 1-D host index (0.1% selectivity). Note the log scale.

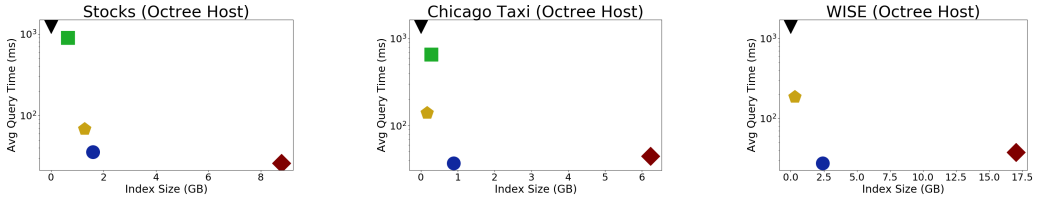


Figure 3-7: Performance of Cortex and baselines on three datasets, using an Octree host index (0.1% selectivity). Note the log scale.

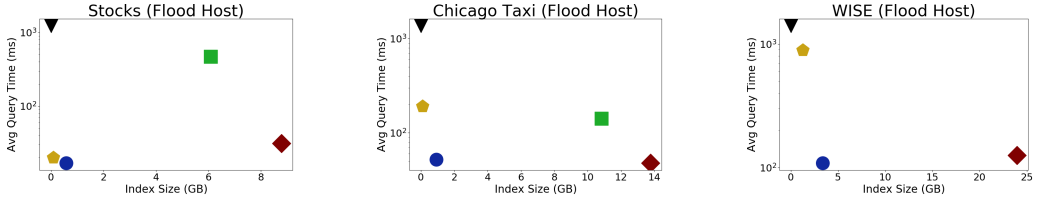


Figure 3-8: Performance of Cortex and baselines on three datasets, using an Flood host index (0.1% selectivity). Note the log scale.

1. *Clustered 1-D* is a one-dimensional primary index sorted by a single column that performs a binary search to look up a range. Cortex splits the host column into up to 100,000 host buckets. The clustered key is the column value, plus a unique ID in case of duplicates.

2. *Octree* is the extension of a quadtree to more than two dimensions, with a maximum leaf page size of 10,000. The clustered key for a record is its page ID with a counter to uniquely identify it within the page. We choose an Octree, since results from [66] suggest it is the most performant of traditional multi-dimensional index structures.

3. *Flood* is a learned multi-dimensional index that outperforms traditional data structures by adapting to the dataset and query workload [66]. Flood uses a grid layout for its index and automatically tunes the number of partitions based on a sample workload. We train Flood on a workload consisting of uniformly distributed queries over all host columns, with an average selectivity of 0.1%. The clustered key for a record is its Flood cell ID with a counter to uniquely identify it within the page.

In all cases, the clustered key index is a cache-optimized B-Tree. Note that each experiment includes randomly shuffled queries over only the *correlated* (i.e. target) dimensions, but *not* over the host columns. Therefore, all queries utilize the correlation index, which focuses the evaluation on the performance of the baselines alone.

3.6.2 Datasets

We evaluate Cortex on several real large-scale datasets with a variety of correlation types, summarized in Table 3.2. Note that host columns vary based on host index and dataset.

Stocks is a dataset with 165M rows comprising daily statistics for over 70k equities listed on various international exchanges, the earliest of which has data from 1962 [**?**]. The columns are date, stock ticker, daily high, daily low, open, close, and volume. The high, low, open, and close attributes exhibit a tight soft-functional linear correlation, since the price does not usually fluctuate substantially (more than a few percentage points) over a single day.

Chicago Taxi is a dataset with data on 194M taxi trips in Chicago [**?**]. Columns include start time, end time, duration, distance, metered fare, tips, tolls, and total fare. This dataset covers a wide range of correlations (Figure 3-4): the metered and total fares are well correlated with the combination of (duration, distance), but have a very weak relationship with each individually. This relationship is not simply linear: for example, longer trips tend to use a flat fare. Tips have a non-functional correlation with total fare (since passengers tend to choose one of several fixed percentages), while total fare and metered fare are moderately correlated.

WISE samples 198M astronomical objects from the NASA Wide-Field Infrared Survey Explorer [**?**]. Among its 15 attributes are coordinates for each object in 3 different coordinate systems: Right Ascension (RA) and Declension (Dec), galactic longtitude and latitude (GLon/GLat), and ecliptic longtitude and latitude (ELon/ELat). The two coordinates in any one coordinate system together determine each coordinate in every other system, making this a strong multi-way correlation. In addition to coordinates, the dataset contains two photometric measurements, $W1$ and $W2$, along with 2 different error measurements for each. Figure 3-5 shows examples of the non-functional, weak functional, and multi-way correlations in WISE. Unlike Chicago Taxi, where many complex correlations still had strong linear components, the correlations in WISE are more diffuse. NULL values are assigned a large negative value, which skew Hermit's linear model by producing a large outlier band; therefore,

we automatically assign them as outliers in Hermit and disregard them when building the piecewise linear model.

3.6.3 Performance Comparison

Figures 3-6 – 3-8 show the tradeoff between memory footprint and query performance for Cortex and the other baselines, on a query workload with 0.1% selectivity. Note that the correlated target columns being indexed (and therefore the query workload) are different for single and multi-dimensional host indexes (see Table 3.2); as a result, the size of an index will differ between the two settings. All datasets demonstrate that, on queries with 0.1% selectivity, Cortex strikes the best size-speed tradeoff, achieving the lowest query time with nearly the smallest index size. In particular, across the three datasets, Cortex can *match* or *outperform* a B-Tree's performance while using $17.5 - 71\times$ and $5.5 - 7\times$ less space on single and multi-dimensional host indexes, respectively. At the same time, Cortex outperforms the next best correlation index by between $2 - 18\times$, with a wider margin ($6.4 - 18\times$) on the weaker correlations in the Chicago Taxi and WISE datasets. Note that we don't include Hermit in the evaluation of the WISE dataset with an Octree host, since it cannot fit the multi-way correlations. For example, GLon is nearly independent to any other single attribute; when Hermit tries to fit this correlation, it hits its max depth, occupies more memory than the secondary index, and takes an inordinate amount of time.

On the Stocks dataset, Cortex can outperform Hermit and CM on a single-dimensional host index, despite their specializing in these types of soft-functional correlations. Its performance improvement is less substantial on the multidimensional hosts, which we investigate further below. However, the gap between Cortex and these previous solutions grows wider on the Chicago Taxi and WISE datasets, which have weaker correlations. CM suffers by not being able to account for the outliers on

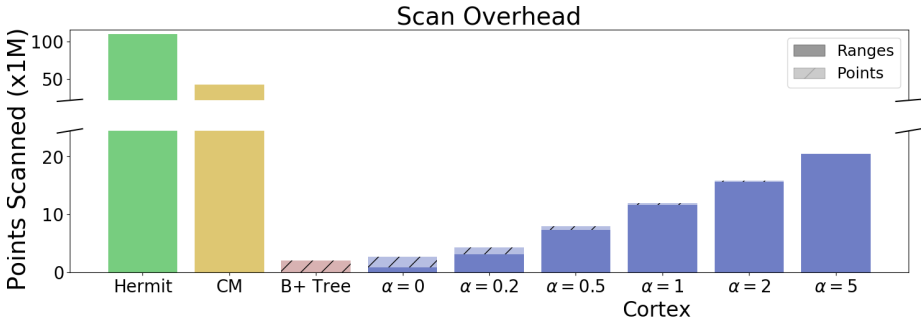


Figure 3-9: Cortex scans a smaller range than CM or Hermit, with an octree host index on Chicago Taxi (1% selectivity).

the fringes of these weaker correlations, which results in its scanning a large range (Figure 3-9). On the other hand, Hermit's outlier threshold filters out almost the entire dataset, which we investigate further below. Hermit's tight bound on outliers works well on strong correlations but quickly degrades in the presence of weaker ones.

We now take a deeper look at how Cortex's performance varies based on multiple parameters: query selectivity, the parameter α, the choice of host index, and the strength of the correlation.

Baseline	**Stocks**				**Chicago Taxi**				**WISE**			
	0.01%	**0.1%**	**1%**	**5%**	**0.01%**	**0.1%**	**1%**	**5%**	**0.01%**	**0.1%**	**1%**	**5%**
Full Scan	0.002	0.022	0.173	0.878	0.004	0.028	0.220	0.974	0.005	0.045	0.193	0.683
Correlation Map	0.482	0.635	1.843	3.495	0.098	0.179	0.495	1.427	0.015	0.093	0.348	1.229
Hermit	1.842	2.463	2.377	3.402	0.005	0.049	0.350	1.069	0.006	0.051	0.250	0.648
Cortex	2.934	6.149	5.351	6.347	1.336	1.654	2.449	3.012	0.956	1.722	1.221	2.484

Table 3.3: Multiplicative speedup of Cortex and other baselines compared to an optimized secondary B-Tree index, across a range of query selectivities. All experiments use a single-dimensional clustered host index.

Selectivity. Table 3.3 and Table 3.4 show how the relative performance between Cortex and baselines changes as the query selectivity varies from 0.01% to 5%. Note that for this and all further experiments, we take $\alpha = 1$, which we find strikes a reasonable balance between good performance at less than 20% the size of a B-Tree.

Baseline	Stocks				Chicago Taxi				WISE			
	0.01%	0.1%	1%	5%	0.01%	0.1%	1%	5%	0.01%	0.1%	1%	5%
Full Scan	0.006	0.025	0.193	0.953	0.034	0.058	0.256	1.073	0.054	0.089	0.306	1.020
Correlation Map	0.589	1.560	3.544	5.555	0.552	0.541	1.196	2.764	0.093	0.141	0.407	1.156
Hermit	0.023	0.066	0.950	1.230	0.154	0.733	1.132	1.034				
Cortex	0.940	1.854	4.031	6.214	0.946	1.052	1.892	3.413	0.788	1.156	2.467	4.274

Table 3.4: Multiplicative speedup of Cortex and other baselines compared to an optimized secondary B-Tree index, across a range of query selectivities. All experiments use a Flood host index.

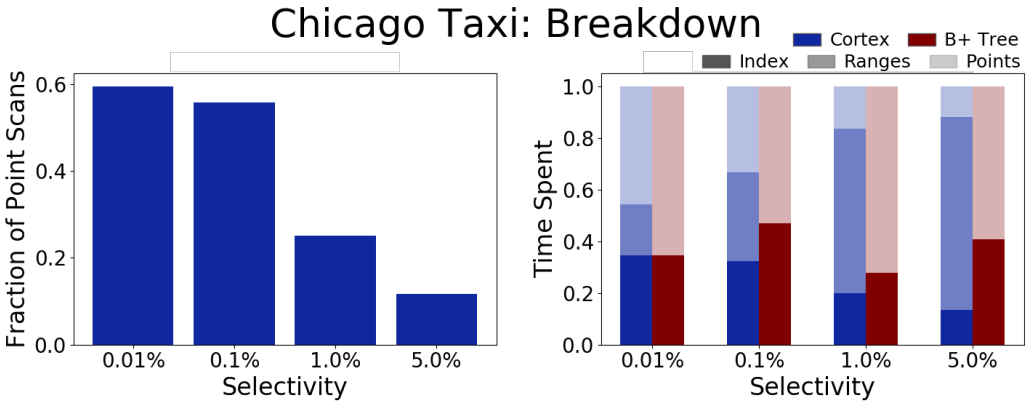


Figure 3-10: As the selectivity increases, Cortex ($\alpha = 0.2$) performs fewer point accesses using the outlier index, as a fraction of result size (left). Cortex switches from point scans to range scans as the query selectivity increases (right). "Index" time refers to all other sources of latency, including deduplication.

Cortex's outlier assignment remains the same for each query workload; in other words, it is not retrained or reconfigured differently for each set of queries. We present the results as speedups *relative* to the secondary B-Tree index, since it is de facto standard in databases today. Note that the sizes of the indexes are the same as in Figures 3-6 – 3-8; just the query workload has changed. There are two main takeaways.

First, compared to the other baselines, the B-Tree shines in its handling of queries with low selectivity (at 0.1% and lower). It outperforms Correlation Maps and Hermit by up to several *orders of magnitude* because it quickly identifies matching records

without scanning false positives. However, unlike other approaches, Cortex is able to achieve query speeds that are generally competitive with the B-Tree on selective queries, and sometimes even outperform it. Figure 3-10 shows that the explanation lies in Cortex's hybrid approach: at low selectivities, Cortex gets most of its results from the outlier index, and can therefore avoid scanning unnecessarily large ranges. This is because selective queries cover very few target buckets, and each target bucket only has a small number of host buckets it considers inliers. Given that a B-Tree's performance on these low selectivity queries is what often justifies its high storage overhead, Cortex is an attractive alternative, offering similar performance at a fraction of the storage cost.

Second, Cortex consistently outperforms the B-Tree and other alternatives at higher selectivities. Note that it is relatively easy to outperform the B-Tree at selectivites of 5% or higher; at this selectivity, even a full scan is on par with a B-Tree. The poor performance of secondary indexes at high selectivities usually results in RDMSes falling back to a full scan for larger query ranges [**?**]. However, Cortex is able to maintain a $1.5 - 6\times$ improvement over the B-Tree while also outperforming other indexes. This again follows from Figure 3-10, since Cortex's design lets it transition to efficient range scans and away from costly point scans as the query selectivity grows. This transition is completely organic: even though Cortex selects more outliers to scan than on low selectivity queries, a larger fraction of them are subsumed within the inlier ranges being scanned (which are also larger) and are therefore deduplicated. Although deduplication is extra overhead in Cortex (as captured by "Index" time in Figure 3-10), at larger selectivities, this cost is more than made up for by avoiding expensive point lookups.

Overall, Cortex performs favorably compared to all baselines, on both low and high selectivity workloads.

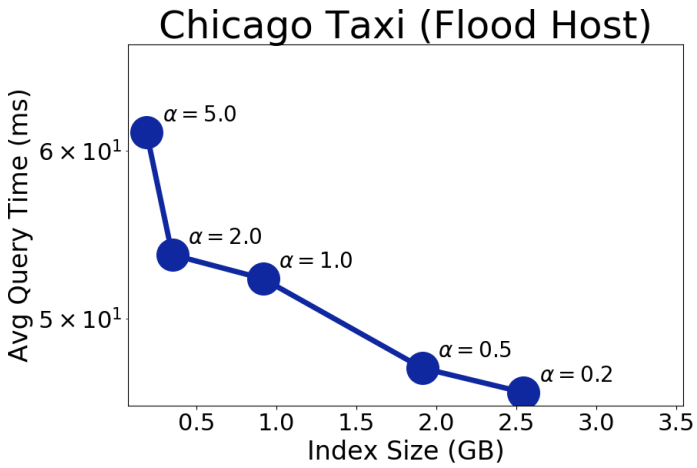


Figure 3-11: Performance effects of varying α on a query workload with 0.1% selectivity on a Flood host index. Other host indexes show similar trends.

Varying α. Figure 3-11 shows that, as α decreases, Cortex's performance generally improves and its space overhead increases. This is expected, since a lower α means we are willing to give up more storage for even marginally faster performance. As a result, Cortex prunes out large ranges of points by assigning as outliers exactly those points that would have otherwise caused it to scan extra host buckets. Figure 3-9 confirms that as α decreases, the number of point scans (as a fraction of the result size) increases, but is more than compensated by the decrease in the size of the ranges scanned. Indeed, Figure 3-9 shows that even a small number of points in the outlier buffer can dramatically reduce the range scans Cortex performs compared to CM and Hermit.

Multi-dimensional Hosts. The choice of host index plays an important role in the performance of Cortex and other baselines. Figure 3-7 shows that on some datasets, like Stocks, Cortex shows only modest performance improvements over CM; on others, like Chicago Taxi and WISE, CM performs well but is still about 4× slower than Cortex. This is due to the geometry of the host pages. For example, in the Stocks

dataset, since the Octree indexes two dimensions, (a) each bucket is wider along the *Open Price* column than on a single-dimensional index and (b) a single value of *Open Price* may map to multiple host buckets. The effect of (a) is to decrease the number of outliers, since nearby values are more likely to be mapped to the same bucket. This is particularly true in the Stocks dataset, where all points lie close to the best fit line. With fewer outliers, CM's performance improves. The effect of (b) is to increase the size of the bucket mapping maintained by the inlier index. The number of host buckets per target bucket increases by $11.1\times, 6.1\times$, and $12\times$ on the Stocks, Chicago Taxi, and WISE datasets, respectively, when changing from a single to multi-dimensional host index.

Hermit's performance deteriorates substantially with a multi-dimensional hosts because its piecewise linear mapping assumes a one-dimensional host column and does not understand the division of points imposed by a multi-dimensional index. The outlier tolerance around the piecewise model ends up intersecting a large number of buckets, even at a low selectivity, and even if those buckets do not contain any relevant points. This incurs a larger unnecessary scan overhead than CM, since CM will only scan a host bucket if it is known to contain at least one point in a target bucket touched by the query.

Given Cortex's performance with single and multi-dimensional host indexes, what is the advantage of a multi-dimensional host index? Their benefit is in allowing the database to efficiently answer queries along more attributes than if using a single-dimensional host index. With more indexed columns, there's a better chance that a correlation exists that Cortex can take advantage of. Table 3.2 shows that with an Octree host index, we are able to correlate more columns to the set of host columns.

Multi-dimensional indexes also have the option of extending their reach simply by indexing more columns. A natural question is then: how does Cortex, with an Octree

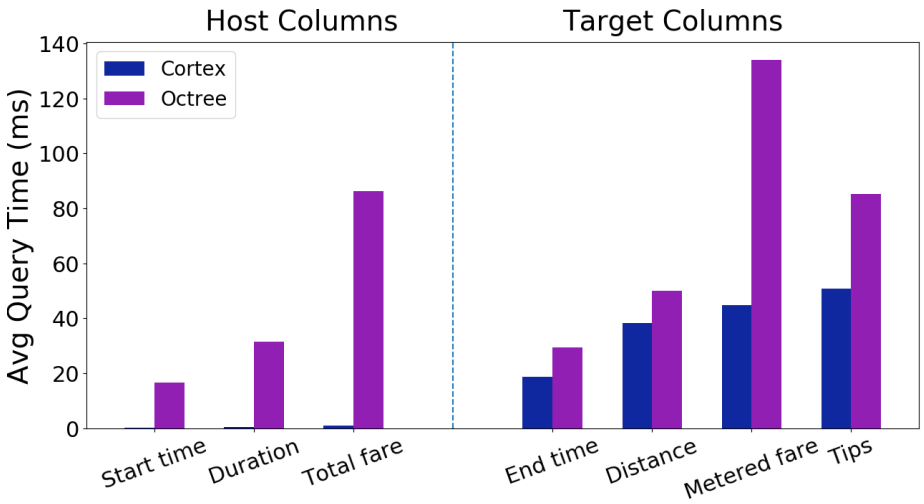


Figure 3-12: Compared to an Octree on both host and target columns on Chicago Taxi, Cortex with an Octree host boosts query times on *all* columns (0.1% selectivity). Host columns see a 76× boost in query performance on average. For reference, a full scan takes 1.3s.

index over a small number of host columns, compare to using a "full" Octree that indexes all columns together (both host and correlated target columns). Figure 3-12 breaks down the performance of these two options on the Chicago Taxi dataset, using queries with 0.1% selectivity. Though Cortex achieves faster query times on the target columns, this is not where its main advantage lies. On the host columns, Cortex achieves a 76× speedup over a full Octree. This is a consequence of the curse of dimensionality: by indexing the extra four target columns, the Octree loses granularity on the original host columns, making queries on those host columns slower. This is particularly undesirable if the host columns happen to contain frequently queried attributes. Figure 3-12 thus shows that Cortex can extend the reach of host indexes, especially multi-dimensional indexes, to more attributes.

Correlation Types. Table 3.3 and Table 3.4 suggest that the type of correlation impacts Cortex's performance improvement relative to alternatives. Cortex (and

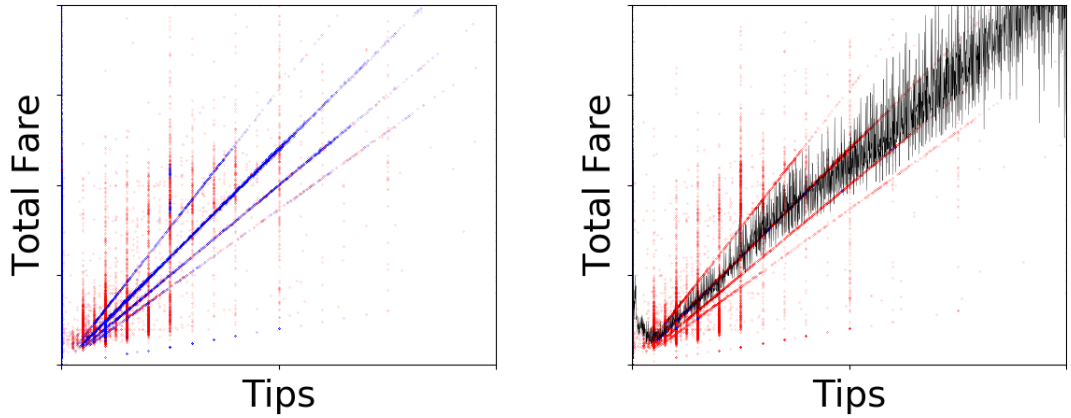


Figure 3-13: Outliers in Cortex (left, $\alpha = 0$) and Hermit (right, with the piecewise model), on a correlation in Chicago Taxi.

Hermit) enjoy strong performance on the Stocks dataset (Figure 3-6), even on low-selectivity queries, since the strong correlations mean that they can zero in on a narrow range on the host column with few false positives. The Chicago Taxi and WISE datasets have progressively weaker correlations, so Cortex's speedup over alternative indexes widens, as those approaches are less effective at indexing weak correlations. Notably, Cortex still remains competitive with the B-tree.

In particular, CM deteriorates rapidly in the presence of outliers (consistent with findings in [**?**]), while Hermit's outlier thresholds are hard-coded for a situation where each target column value maps to only a handful of host column values. This is particularly noticeable on non-functional correlations, like in the Chicago Taxi dataset shown in Figure 3-13. Hermit struggles to fit its piecewise linear model to the data. It jumps across large ranges of values, which causes it to scan large ranges of data even on low selectivity queries (confirmed by Figure 3-9). On the other hand, Cortex is able to better isolate the major trend lines. Section 3.6.4 takes a closer look at the effect of correlation strength on Cortex's performance.

In addition to non-functional correlations, Cortex is also able to handle correlations between more than two attributes, or *multi-way* correlations. The WISE dataset offers a straightforward instance of a multi-way correlations between its three coordinate systems: RA / Dec., Galactic Lat/Lon, and Ecliptic Lat/Lon. Indexing the two coordinates in any one system determines the coordinates in the other two. However, given only one coordinate, the distribution of any other coordinate appears nearly uniformly random. Compared to a host index that only indexes a single coordinate, using a host index with a pair of coordinates results in Cortex classifying 4.4× fewer outliers and reducing the number of host buckets per target bucket by 8×, when indexing any of the remaining coordinates. In other words, when both coordinates are host columns, Cortex can scan much smaller ranges while taking considerably less space. This is how Cortex takes advantage of multi-way correlations.

3.6.4 Cortex's Outlier Assignment

		Outlier Detection Algorithm			
		Cortex (Flood)	Cortex (Octree)	Isolation Forest	DBSCAN
Host	Flood	**45.8**	118	117	117
	Octree	154	**36.8**	156	155

Table 3.5: Performance (in ms) of multi-dimensional hosts on various outlier detection algorithms. Cortex (Flood) uses Cortex's outlier assignment with Flood's host buckets (likewise for Octree). Results are on the Chicago Taxi dataset at 0.1% selectivity.

Outliers are central to Cortex: the choice of outliers almost single-handedly determines the storage-performance tradeoff. In this section, we examine which points Cortex prioritizes when making its assignment. Figure 3-14 helps answer this question by looking at the outliers from a weak correlation in the WISE dataset, between W_1 *SNR* and W_1 Magnitude, for two values of α. At $\alpha = 1$, Cortex eliminates points

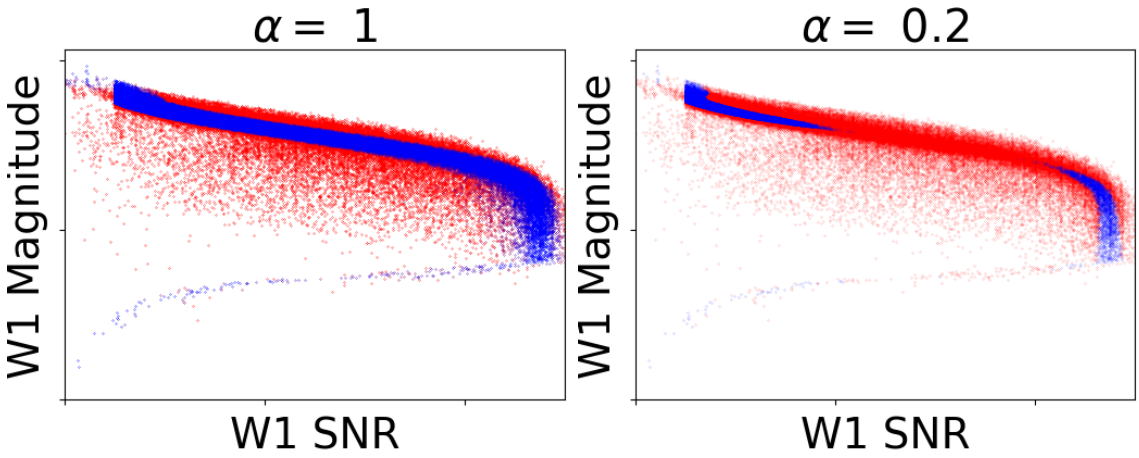


Figure 3-14: Cortex's outlier assignment on a correlation from the WISE dataset. Outliers are shown in red.

that do not lie in the main trend of the data, therefore isolating the dense areas. It seems to treat all parts of the trend shape equally: the trend is not substantially wider in some areas than others. However, when decreasing α to 0.2, Cortex assigns many more outliers to the center of the trend than to other parts. The reason is that the trend has a shallower slope in the middle. A host bucket (of W_1 *magnitude*) is intersected by more target buckets in the center of the trend than at the endpoints. Cortex realizes that these host buckets add a large number of false positives and increase scan overhead. In response, Cortex tends to stash points in these host buckets into the outlier index. Importantly, this shows that Cortex therefore can prioritize different areas of the correlation, and assign proportionally more or fewer outliers as necessary. This distinguishes Cortex from prior work: the ability to flexibly assign outliers is only possible because Cortex doesn't determine outliers based on a fixed functional model.

Section 3.4.3 argues that the right outlier detection scheme for Cortex must cater to the underlying host index. Table 3.5 demonstrates this by evaluating Cortex

on multi-dimensional host indexes with outlier assignments that do *not* cater to the respective hosts. We demonstrate Cortex's performance on outlier assignments determined by the DBSCAN [**?**] and Isolation Forests [**?**] algorithms implemented in Python's `scikit-learn` [**?**]. To improve the speed of these implementations, we randomly partition the dataset into 100 chunks and run each algorithm on all chunks. Generic outlier detection schemes perform more than 3× worse than Cortex's outlier assignment algorithm. While these generic algorithms capture the "obvious" outliers and thus outperform Correlation Maps, they do a poor job of choosing outliers near host bucket boundaries.

3.6.5 Scalability

In this section, we probe Cortex's performance when extended to many columns and a large amount of noise in the correlation. We use a synthetic dataset with 100M records with values in the range $[0, 10^6]$. Correlations consist of an exact linear map ($i.e., y = x$) with some *noise fraction* f (which varies by experiment). For a given f, we add additive noise to f fraction of the points, drawn from a Laplace distribution with $\sigma = 200000$ (one-fifth of the value range).

Number of Columns. Cortex scales linearly, both in space usage and performance, with the number of target columns, since each column is indexed independently of the others, much like a secondary index. Figure 3-15 confirms this expectation using our synthetic dataset. We generate increasingly larger datasets with between 10 and 45 columns, using noise fraction $f = 0.2$. For each dataset, we create Cortex ($\alpha = 1$) and B-Tree indexes on all except one column (the host column). Since Cortex occupies a fraction of the space of a B-Tree, we found that the B-Tree ran out of memory after indexing 25 target columns. Meanwhile, Cortex scales to 45 target columns.

Robustness to Noise. How weak of a correlation can Cortex index and still maintain

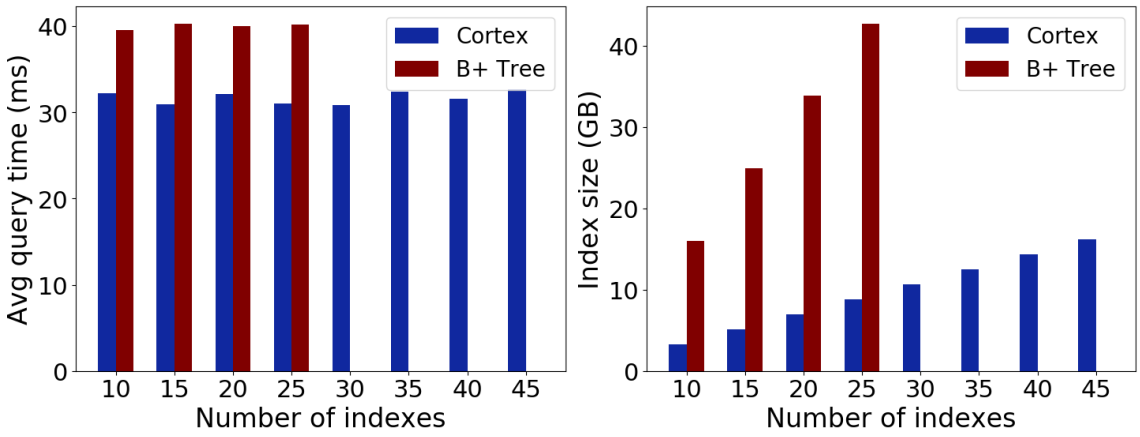


Figure 3-15: Cortex scales to more columns (without sacrificing query speed) than a traditional secondary index, which runs out of memory after indexing 25 columns.

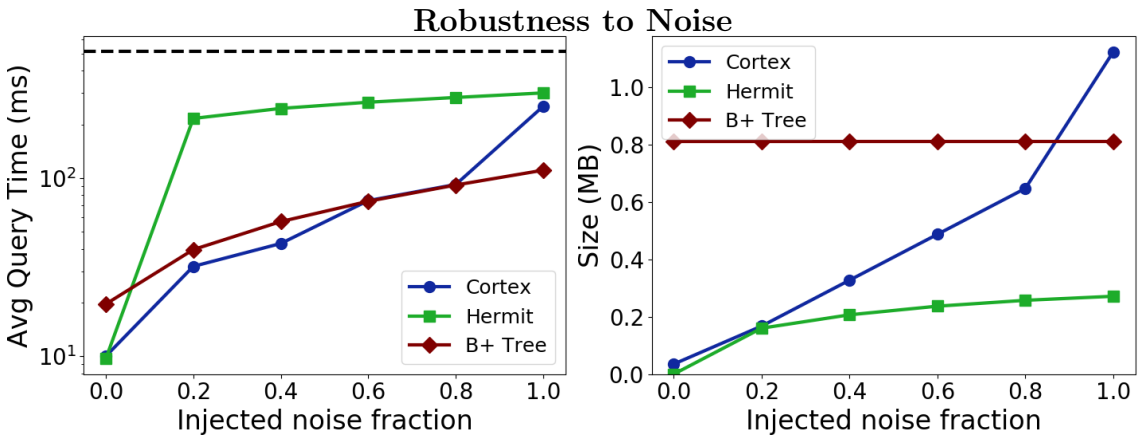


Figure 3-16: Cortex's query performance and speed relative to baselines, as a function of the noise fraction. The dotted line indicates a full scan. Note the log scale on the left.

a favorable size-speed tradeoff when compared with secondary indexes? To quantify this, we run Cortex and a secondary B-Tree index on our synthetic dataset with 2 columns and increasingly more noise, measured by the noise fraction f defined above. Figure 3-16 shows that Cortex is able to maintain performance on par with a B-Tree up until the data is 80% noise, all while occupying a fraction of the B-Tree's memory

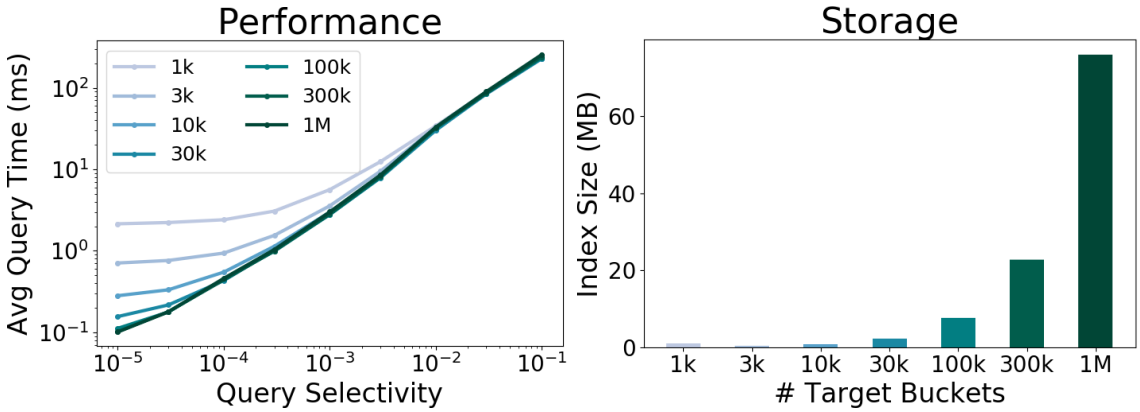


Figure 3-17: Increasing the number of target buckets lets Cortex handle more selective queries but uses more space.

footprint. As long as a strong correlation exists under the noise, Cortex finds it by stashing all other points as outliers; its space usage grows linearly with the noise fraction, reflecting its ability to grow its outlier index without tuning. On the other hand, Hermit's performance deteriorates since it does not locate as many outliers. This is a consequence of Hermit's requirement of a hard-coded limit on outliers, which doesn't let its outlier stash adapt to the correlation like Cortex's. Cortex's deterioration in performance at 100% noise is due to two factors. Cortex only stashes 47% of the records, which means its outlier assignment cannot capture all the noise. Even though the number of outliers dropped, Cortex's size still increases: with fewer outliers stashed, each target bucket maps to more host buckets, increasing the size of the cell mapping maintained by the inlier index.

Bucketing Strategy. Here, we examine the reasoning behind Cortex's strategy to optimize the target bucket size (Section 3.5.2). We use the same synthetic dataset and observe the query performance over a range of selectivities and target buckets in Figure 3-17. Note that outliers are distributed uniformly over the target value range, consistent with the assumptions made in Section 3.5.2. We compute the measured

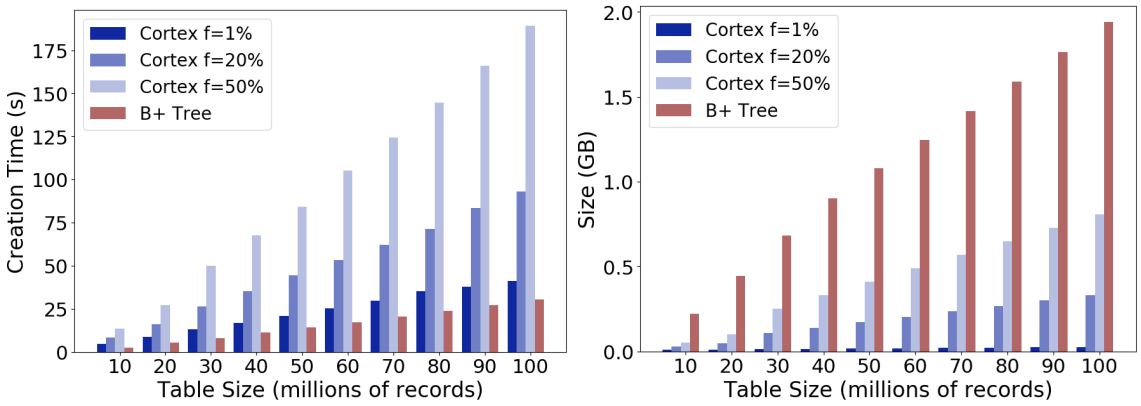


Figure 3-18: Creation time (left) and size of Cortex (right) over a range of table sizes and correlation strengths.

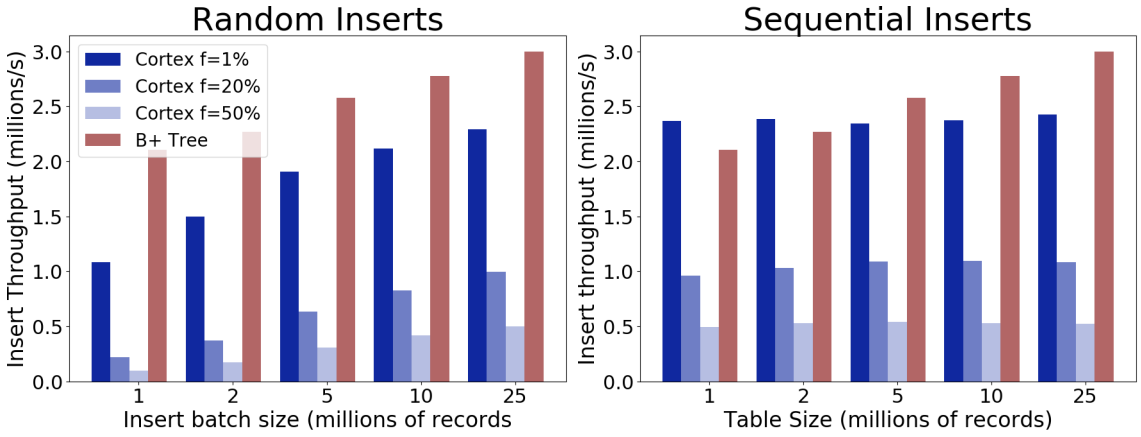


Figure 3-19: Throughput on random (left) and sequential (right) inserts for a variety of correlation strengths.

scan overhead (MSO) as the average query time divided by the query selectivity; this quantity should be roughly proportional to the expression from Equation 3.2. Indeed, fitting $\text{MSO} = A\left(1 + \frac{1}{ts_q}\right)$ for A yields a strong correlation, with $R^2 = 0.993$, suggesting that Equation 3.2 is a suitable approximation for query time in Cortex's optimization.

3.6.6 Insertions

Cortex's creation consists of initializing its correlation tracker with all the cells that contain points. The latency of this operation depends heavily on the strength of the correlation being indexed. Figure 3-18 shows that strong correlations ($f = 1\%$ noise) can be initialized more than 4× faster than weak correlations ($f = 50\%$ noise). Note that Cortex achieves more than 1M inserts per second on strong correlations, but is slower than inserting into a traditional B-Tree, since it must first aggregate records by cell.

Cortex allows incremental updates without the need to reassign all records in the table. Since it is often the case that records are inserted in order of primary key (e.g., a timestamp), we evaluate inserts on both a random load, and a load where records are sequentially added in increasing key order. Figure 3-19 shows that, in general, sequential inserts are faster than random inserts; insertion time scales with the number of affected *cells*, and sequential inserts affect fewer host buckets in each batch. This also explains why random inserts are more efficient in larger batches: as more points are added, many of them (particularly inliers) fall in the same bucket.

3.7 Conclusion

This work presented Cortex to harness the correlations between attributes in a dataset. As an alternative to a secondary index, which incurs high storage overhead, Cortex adapts itself to the primary index on the table, whether single or multi-dimensional, to encode a correlated column in terms of a set of host columns. On queries with low selectivities, it is able to match the B-Tree's performance using 10× less space, while outperforming prior solutions by close to an order of magnitude. On high selectivity queries, it outperforms both B-Trees and other baselines. Key to Cortex's

performance gains is its ability to judiciously choose which records to consider as outliers. It makes this determination without the use of any predefined model, instead using a cost model of performance to guide its outlier assignment. Cortex offers speedups on both single and multi-dimensional host indexes and is a compelling alternative to secondary indexes in the case of correlated columns.

Chapter 4

Minerva: End-to-End Transport for Video QoE Fairness

4.1 Background

In most on-demand video streaming services, such as Netflix and YouTube, users stream video from a centralized video server using the Dynamic Adaptive Streaming over HTTP(S) (DASH). Since these connections are made over the wide area, traffic is variable and results in fluctuating network bandwidth. To adapt to these conditions, video servers using the DASH protocol do the following:

1. Chunking (server-side): instead of downloading the entire video in one shot, clients download chunks of video, typically around 4 seconds long, when they need it. This prevents wasted bandwidth in case the user pauses or exits the video before finishing it.

2. Multi-resolution encoding (server-side): each chunk is encoded at multiple resolutions. A user with a poor connection can request a low-resolution chunk, while those with higher bandwidth can benefit from a higher quality viewing

experience.

When chunks are fully downloaded before they are viewed, they are added to a buffer, from which the video is played. If the buffer is depleted, no video frames are available, and the video player stalls or *rebuffers*, an undesirable impact on the end user.

In order to take advantage of both chunking and multi-resolution encoding, video streaming players implement an *adaptive-bitrate (ABR) algorithm* to decide, in real time, the resolution at which to download the next chunk. These algorithms try to measure the available capacity in the network and use those measurements to optimize some notion of *quality of experience (QoE)* for the viewer, which captures both the perceived quality of the video (which is heavily tied to the resolution), jarring switches between resolutions, and rebuffering events. Well known examples of ABR algorithms include:

- Buffer-based: the client defines a minimum buffer level, or reservoir, that it would like to maintain. The original implementation used by Netflix uses a reservoir of 90 seconds. Based on the throughput measured while downloading the last chunk, the video player chooses the highest bitrate which ensures that the buffer after downloading will stay above the reservoir.

- Model-Predictive-Control (MPC): the client models the QoE as:

$$\text{QoE}(c_k, R_k, c_{k-1}) = P(c_k) - \beta R_k - \gamma \left| P(c_k) - P(c_{k-1}) \right| \tag{4.1}$$

  Here, c_k is the kth chunk, R_k is the time spent rebuffering before playing the kth chunk, β is a parameter that dictates how bad a rebuffering event is, and γ

governs how much the client penalizes switching between bitrates. The function $P(\cdot)$ is the perceptual quality of the chunk.

MPC makes a conservative estimate of the network bandwidth, and uses this estimate to simulate the next H chunks, where H is a user-defined horizon parameter (the original implementation used $H = 5$). It then chooses the bitrate that results in the highest QoE after H chunks.

- Pensieve: The variability of the network connection is what makes ABR a challenging problem; if the network were to provide a constant bandwidth, computing the optimal bitrate for the client to fetch would be trivial. Yet, the above solutions make only rough predictions about network throughput. Pensieve is an ABR algorithm that uses deep reinforcement learning to translate measurements of network quality and client state into a decision for the next chunk to fetch from the video server. Pensieve uses an actor-critic approach to train both a value and policy network that, respectively, estimates the QoE advantage of choosing a particular chunk and decides which chunk to choose.

4.1.1 Our Contribution: Minerva

ABR algorithms optimize the QoE for a single client. However, they cannot mediate bandwidth between multiple clients that are sharing the same link. Such mediation could be advantageous to the video provider by preventing situations where one client's experience suffers at the expense of others'. For example, consider the following three situations:

1. Client A has a large amount of video buffered, and Client B is about to stall, due to a sudden drop in network capacity.

2. Client A is watching on a mobile device with a small screen, on which an increase in resolution will have little to no benefit. Meanwhile, Client B is watching on a large TV where such bitrate changes are more noticeable.

3. Client A is watching a video that lends itself to efficient compression, while Client B is watching a video that is harder to encode, e.g. an action film.

In all of the above cases, the network could "reassign" bandwidth from Client A to Client B, substantially improving Client B's quality of experience without a significant negative impact on Client A's experience. Minerva is an end-to-end transport protocol that, unlike ABR algorithms, can adjust the bandwidth allocation between clients to achieve these types of collective QoE benefits. Most notably, it does so without either (a) adding any extra hardware to the middle of the network or (b) being unfair to other non-video traffic sharing the bottleneck link.

Minerva implements the components of the aforementioned instance-optimality framework. The workload it considers is the distribution of videos being streamed in any particular time window. The visual qualities of these videos, in addition to the client's state, inform how much bandwidth Minerva needs to allocate to each video to achieve the same "performance", in this case measured by Equation 4.1. In order to take advantage of this information, Minerva operates at, and exposes new knobs at, the transport layer, by modifying the congestion control algorithm of each client independently. Clients occupying less bandwidth than their fair share request a larger share of the bandwidth, while those with higher bandwidth than required reduce their aggressiveness. This process, governed by a custom rate update algorithm, eventually converges to an optimal rate allocation that maximizes fairness.

Minerva uses information about the player state and video characteristics to adjust its congestion control behavior to optimize for *QoE fairness*. Minerva clients receive

no explicit information about other video clients, yet when multiple of them share a bottleneck link, their rates converge to a bandwidth allocation that maximizes QoE fairness. At the same time, Minerva videos occupy only their fair share of the bottleneck link bandwidth, competing fairly with existing TCP traffic. We implement Minerva on an industry standard video player and server and show that, compared to Cubic and BBR, 15-32% of the videos using Minerva experience an improvement in viewing experience equivalent to a jump in resolution from 720p to 1080p. Additionally, in a scenario with dynamic video arrivals and departures, Minerva reduces rebuffering time by an average of 47%.

4.2 Introduction

HTTP-based video streaming traffic has grown rapidly over the past decade. Video traffic accounted for 75% of all Internet traffic in 2017, and is expected to rise to 82% by 2022 [14]. With the prevalence of video streaming, a significant body of research over the past decade has developed robust adaptive bitrate (ABR) algorithms and transport protocols to optimize video quality of experience (QoE) [3, 56, 94, 34, 85, 10, 24]. The majority of this research focuses on QoE for a *single* user in isolation. However, due to the fast-paced growth of video traffic, it is increasingly likely that multiple video streaming clients will share a bottleneck link. For example, a home or campus WiFi network may serve laptops, TVs, phones, and tablets, all streaming video simultaneously. In particular, the median household in the U.S. contains five streaming-capable devices, while one-fifth of households contain at least ten [76].

Video content providers today are beholden to the bandwidth decisions made by existing congestion control algorithms: all widely-used protocols today (e.g. Reno [4] and Cubic [30]) aim to achieve *connection-level fairness*, giving competing flows an equal share of the link's capacity on average. Therefore, content providers running

these protocols can only optimize for user viewing experience in isolation, e.g. by deploying ABR algorithms. They miss the bigger picture: allocating bandwidth carefully between video clients can optimize the overall utility of the system. The opportunity to optimize viewing experience collectively is particularly relevant for large content providers. Netflix, for example, occupies 35% of total Internet traffic at peak times and may therefore control a significant fraction of the traffic on any given bottleneck link [78].

Specifically, there are two problems with standard transport protocols that split bandwidth evenly between video streams. First, they are blind to user experience. Users with the same bandwidth may be watching a variety of video genres in a range of viewing conditions (e.g. screen size), thereby experiencing significant differences in viewing quality. Today's transport protocols are unaware of these differences, and cannot allocate bandwidth in a manner that optimizes QoE.

Second, existing congestion protocols are blind to the dynamic state of the video client, such as the playback buffer size, that influences the viewer's experience. For example, knowing that a client's video buffer is about to run out would allow the transport to temporarily send at a higher rate to build up the buffer, lowering the likelihood of rebuffering. Protocols like Cubic, however, ignore player state and prevent clients from trading bandwidth with each other.

We design and implement Minerva, an *end-to-end transport protocol for multi-user video streaming*. Minerva clients dynamically and independently adjust their rates to optimize for *QoE fairness*, a measure of how similar the viewing experience is for different users. Minerva clients require no explicit information about other competing video clients, yet when multiple of them share a bottleneck link, their rates converge to a bandwidth allocation that maximizes QoE fairness. Crucially, throughout this process, Minerva clients together occupy only their fair share of the link bandwidth,

which ensures fairness when competing with non-Minerva flows (including other video streams). Since clients operate independently, Minerva is easy to deploy, requiring changes to only the client and server endpoints but not to the network. A content provider can deploy Minerva today to optimize QoE fairness for its users, without buy in from other stakeholders.

Central to Minerva are three ideas. First is a technique for deriving *utility functions* that capture the relationship between network bandwidth and quality of experience. Defining these functions is challenging because standard video streaming QoE metrics are expressed in terms of video bitrates, rebuffering time, smoothness, and other application-level metrics, but not network link rates. We develop an approach that reconciles the two and exposes the relationship between them.

Second is a new distributed algorithm for achieving fairness between clients using these utility functions. Our solution supports general notions of fairness, such as max-min fairness and proportional fairness [45]. Each client computes a dynamic *weight* for its video. The transport layer then achieves a bandwidth allocation for each video that is proportional to its weight. A client's weight changes throughout the course of the video, based on network conditions, the video's utility function, and client state, but independently of other clients. Collectively, the rate allocation determined by these weights converges to the optimal allocation for QoE-fairness.

Third is a weight normalization technique used by Minerva to compete fairly with standard TCP. This step allows clients to converge to a set of rates that simultaneously achieves QoE fairness while also ensuring fairness with non-Minerva flows on average.

We implement Minerva on top of QUIC [50] and adapt an industry-standard video player to make its application state available to the transport. For deployability, our implementation uses Cubic [30] as its underlying congestion control algorithm for achieving weighted bandwidth allocation. We run Minerva on a diverse set of network

conditions and a large corpus of videos and report the following:

1. Compared to existing video streaming systems running Cubic and BBR, Minerva improves the viewing quality of between 15-32% of the videos in the corpus by an amount equivalent to a bump in resolution from 720p to 1080p.

2. By allocating bandwidth to videos at risk of rebuffering, Minerva is able to reduce total rebuffering time by 47% on average in a scenario with dynamic video arrivals and departures.

3. We find that Minerva competes fairly with videos and emulated wide area traffic running Cubic, occupying within 4% of its true fair share of the bandwidth.

4. Minerva scales well to many clients, different video quality metrics, and different notions of fairness.

5. We run Minerva over a real, residential network and find that its benefits translate well into the wild.

4.3 Motivation

Central to Minerva is the realization that connection-level fairness, where competing flows get an equal share of link bandwidth, is ill-suited to the goals of video providers like Netflix and YouTube. In particular, connection-level fairness has two undesirable effects from a video provider's point of view.

First, it is oblivious to the bandwidth requirements of different users. Different videos require different amounts of bandwidth to achieve the same viewing quality. For example, a viewer will have a better experience streaming a given video on a smartphone at 1 Mbit/s than a large 4K TV at 1 Mbit/s [55]. Further, discrepancies in viewing experience extend beyond differences in screen size. User studies show that

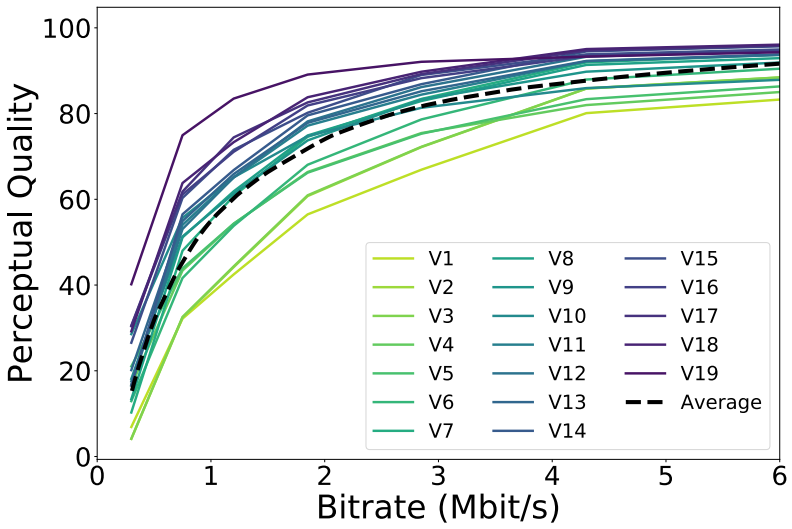
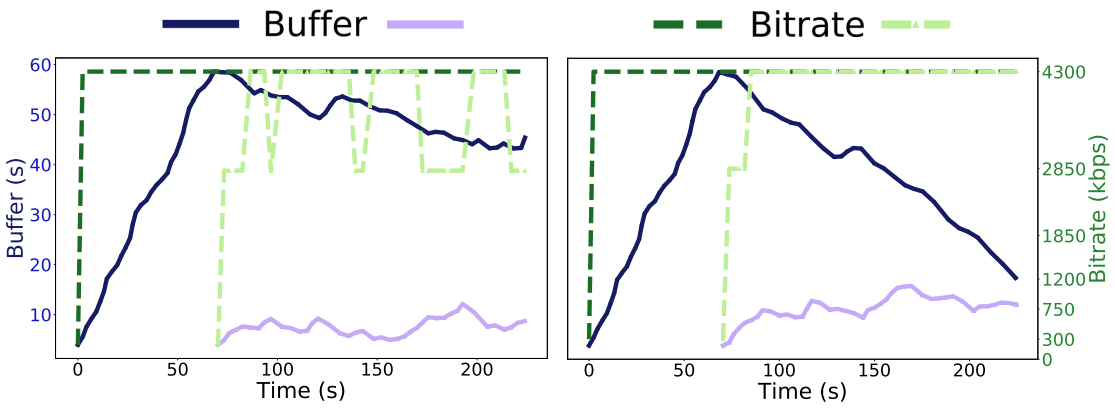


Figure 4-1: Perceptual quality for a diverse set of videos based on a Netflix user study [52]. Also shown is the "average" perceptual quality (dotted), which is Minerva's normalization function (Section 4.6.4). For context, the average qualities at 720p and 1080p are 82.25 and 89.8.

the perceived quality of a video is influenced by its content as well, e.g., its genre or degree of motion. Figure 4-1 plots the average video quality, as rated by viewers, at several bitrates for a diverse set[1] of videos [52]. A client watching the video "V3", for example, would require a higher bandwidth than a client watching "V19" to sustain the same viewing quality. If both received the same bandwidth, "V3" would likely appear blurrier and less visually satisfying. However, current transport protocols that provide connection-level fairness are unaware of these differences in videos, and thus they relegate some viewers to a worse viewing experience.

Second, protocols that split bandwidth evenly between connections are blind to the state of the video client, such as the playback buffer size, so they cannot react to application-level warning signs. In particular, a video client with a low buffer, e.g. at video startup, has a higher likelihood of rebuffering, which prevents the ABR

[1]For a description of each video, see Table 4.1.



(a) Cubic (b) Minerva

Figure 4-2: To demonstrate buffer pooling, we start one video client, allow it to build up a large buffer, and then introduce a second video after 70 seconds. (a) With Cubic, the first video maintains a large buffer, causing the second video to sacrifice its quality to avoid rebuffering. (b) Minerva trades the large buffer of the first video to fetch higher quality chunks for the second.

algorithm from requesting high-quality chunks. Consider such a video sharing the network with other clients that have large buffers. Dynamically shifting bandwidth to the resource-constrained video in question would allow it to quickly build up its buffer, mitigating the adverse effects of rebuffering without sacrificing its own viewing experience or that of the other clients. This dynamic reallocation effectively creates a shared *buffer pool*, letting clients with small video buffers tap into the large buffers of other video clients, as illustrated in Figure 4-2. Equal bandwidth sharing, on the other hand, isolates clients and prevents them from pooling their resources.

While most Internet traffic expects bandwidth to be shared equally between competing flows, connection-level fairness *between videos* is not ideal: providers should be able to allocate bandwidth between their videos in a way that optimizes their viewers' experience, provided they play fairly with other traffic. In fact, providers have economic incentives to consider video quality when delivering content: to boost

user engagement, improving viewing quality at lower bitrates is often more important than at higher bitrates.[2] Connection-level fairness, being blind to video quality, is therefore ill-suited to a video provider's needs. Instead, having the ability to shift bandwidth based on video quality considerations would allow them to directly optimize relevant business objectives.

Many video providers have already abandoned connection-level fairness. Netflix and YouTube typically use three parallel TCP connections to download video chunks, effectively giving their videos larger bandwidth shares and preventing poor viewing experiences [58, 86, 65]. Additionally, Netflix uses a larger number of connections at video startup, when buffer is low, to avoid potential rebuffering events early in the video [86]. These remedies are ad-hoc, coarse (adding or removing only an integral number of connections), heavyweight (incurring startup time for a new connection), and make no effort to be fair to competing web traffic.

Minerva offers video providers a more principled solution. It dynamically allocates bandwidth between videos in a manner that (a) allows fine-grained control over a video's rate share, (b) responds quickly to low buffers and (c) competes fairly with non-video traffic. Importantly, each provider has full control over their use of Minerva. They may deploy Minerva on their client and server endpoints, independently of other providers, and without any change to the network.

Minerva is able to ameliorate the drawbacks of connection-level fairness by dynamically modifying a video's rate allocation to optimize a *QoE fairness* metric. Minerva uses a standard definition of QoE fairness (max-min QoE fairness) that aims to improve the video quality for clients with the worst QoE (Section 4.5.1). However, Minerva is flexible and can optimize for a variety of QoE fairness definitions (Section 4.9.6). It is beyond the scope of this work to determine the best fairness metric

[2]Corroborated in private communication with a large content provider.

for video providers.

4.4 Related Work

Minerva is informed by a large body of work focused on improving user experience while streaming video.

Single-user streaming. In single-user video streaming, each video optimizes only within the bandwidth allocation prescribed by the underlying transport. The underlying bandwidth share between videos is not modified in response to video-level metrics, such as perceptual quality or playback buffer. Improvements to single-user streaming include ABR algorithms, which use bandwidth measurements to choose encodings that will improve perceptual quality and minimize rebuffering. State of the art algorithms are typically also aware of client state and may optimize for QoE either explicitly [94, 3] or implicitly, e.g. via a neural network [56].

Further single-user streaming improvements include techniques that correct for the shortcomings of DASH [83] to achieve fairness across multiple users, by improving bandwidth estimation [53] or avoiding idle periods during chunk downloads [96]. Other schemes manage the frequency at which chunks are requested [43] or model the choices of other clients using a game-theoretic framework [9]. As a result, these methods improve utilization, perceptual smoothness, and fairness among competing videos. However, they improve only connection-level fairness and ignore the perceptual quality differences between videos, so they cannot optimize QoE fairness. Furthermore, they are still ultimately bound by the equal-bandwidth allocation prescribed by the underlying transport.

Transport Protocols. Alternate transport protocols may also improve a viewer's experience. PCC has been shown to make better use of available link bandwidth and thus improve a user's QoE [20]. However, PCC is a general purpose transport

and is not aware of video quality metrics. Salsify [24] designs real-time video conferencing applications that are network aware; the video encoder uses estimates of available bandwidth to choose its target bitrate. However, Salsify targets the real-time conferencing use case, while Minerva targets DASH-based video-on-demand.

Centralized multi-user streaming. Existing systems that optimize QoE fairness over multiple users only consider *centralized* solutions [93, 13]. They require a controller on the bottleneck link with access to all incident video flows. This controller computes the bandwidth allocation that optimizes QoE fairness and enforces it at the link; clients are then only responsible for making bitrate decisions via traditional ABR algorithms. However, this requires a network controller to run at every bottleneck link and thus presents a high barrier to deployment.

Video quality metrics. Another line of work has focused on defining metrics to better capture user preferences. Content-agnostic schemes [55] use screen size and resolution as predictors of viewing quality. Other efforts [91, 61], including Netflix's VMAF metric [52], use content-specific features to compute scores that better align with actual user preferences. Minerva can support any metric in its definition of QoE.

Decentralized schemes. Minerva clients use a distributed rate update algorithm to converge to QoE fairness. A popular framework for decentralizing transport protocols that optimize a fairness objective is Network Utility Maximization (NUM) [45]. NUM uses link "prices" that reflect congestion to solve the utility maximization problem by computing rates based on prices at each sender; repeated iterations of the price updates and rate computations converges to an allocation that optimizes the fairness objective. In practice, NUM-based rate control schemes can be difficult to stabilize. Another approach solves NUM problems by dynamically deciding a *weight* for each sender, and then using a rate control scheme to achieve rates proportional to those weights. This avoids over-and under-utilization of links [64]. Minerva implements this

second approach and is therefore able to simultaneously achieve full link utilization and QoE fairness.

4.5 Problem Statement

4.5.1 QoE Fairness

Minerva optimizes for a standard definition of QoE found in the video streaming literature [94]. QoE is defined for the kth chunk c_k based on the previous chunk and the time R_k spent rebuffering prior to watching the chunk:

$$\text{QoE}(c_k, R_k, c_{k-1}) = P(c_k) - \beta R_k - \gamma \|P(c_k) - P(c_{k-1})\|. \tag{4.2}$$

$P(e)$ denotes the quality gained from watching a chunk at bitrate e, which we term Perceptual Quality (PQ), β is a penalty per second of rebuffering, and γ penalizes changes in bitrate between adjacent chunks (*smoothness*). In general, PQ may vary between videos and clients, based on parameters such as the client's screen size, screen resolution, and viewing distance, as well as the video content and genre. It also typically varies by chunk over the course of a single video: chunks with the same bitrate may have different PQ levels depending on how the content in those chunks are encoded. Minerva can use any definition of PQ that meets the loose requirements in Appendix B, e.g., it is sufficient that $P(e)$ be increasing and concave.[3]

Suppose N clients share a bottleneck for a time period T, during which each client i watches n_i chunks and experiences a total (summed over all chunks) QoE of QoE_i. Our primary goal is *max-min fairness* of the per-chunk average QoE between clients, i.e., to maximize $\min_i \dfrac{QoE_i}{n_i}$.

[3]This property, standard for utility functions, captures the notion that clients experience diminishing marginal utility at successively higher encodings.

Max-min QoE fairness, a standard notion of fairness, captures the idea that providers may reasonably seek to improve the experience of their worst-performing clients. However, achieving max-min QoE fairness may require very different rate allocations for two videos with substantially different perceptual qualities. Providers who are unhappy with such a rate allocation have two alternatives. First, Minerva allows optimizing for max-min QoE fairness *subject to* the constraint that no video achieves a bandwidth that differs from its fair share by more than a given factor μ. Second, it also supports different definitions of fairness, such as proportional fairness. Both approaches are discussed further in Section 4.6.5.

4.5.2 Goals

Minerva's overarching motivation is to provide a practical mechanism to achieve QoE fairness among competing video flows. In particular, we desire that Minerva

1. be an *end-to-end* scheme. Deploying Minerva should only require modifications to endpoints, without an external controller or changes to the network.

2. improve *QoE fairness*. N video flows using Minerva should converge to a bandwidth allocation that maximizes QoE fairness between those N videos. The clients do not know N or any information about other video flows.

3. ensure *fairness with non-Minerva flows*. The total throughput of N Minerva videos should equal that of N Cubic flows to not adversely impact competing traffic. We design Minerva to compete fairly against Cubic, since it is a widely deployed scheme, but our approach extends to other protocols as well.

4. be *ABR agnostic*. The ABR algorithm should be abstracted away, so that Minerva can work with any ABR algorithm. Minerva may have access to the ABR algorithm, but can only use it as a black box.

These properties offer benefits to large video providers, like Netflix, to use Minerva. Netflix videos constitute 35% of total Internet traffic, making them likely to share bottleneck links [78]. This large bandwidth footprint motivates using Minerva to improve collective experience for Netflix viewers. Additionally, since Minerva video streams operate independently, it can be deployed without knowing which videos share bottlenecks or where those bottlenecks occur. Further, even a single provider can benefit from Minerva, independently of whether other providers deploy Minerva as well, since Minerva videos achieve a fair total throughput share with other competing traffic.

4.6 Design

4.6.1 Approach

Minerva repeatedly updates clients' download rates in a way that increases $\min_i \frac{QoE_i}{n_i}$. However, making rate control decisions in the context of dynamic video streaming is a tricky task. The definition of QoE does not directly depend on the client's download rate, so it is not immediately apparent how to optimize QoE fairness by simply changing the bandwidth allocation. In fact, the effects of rate changes on QoE may not manifest themselves immediately; for example, increasing the download rate will not change the bitrate of the chunk currently being fetched. However, it may have an indirect impact: if a client's bandwidth improves, its ABR algorithm may measure a higher throughput and choose higher qualities for future chunks.

To solve the above optimization problem with a rate control algorithm, Minerva must recast it in a form that depends only on network link rates. This is made difficult by the fact that the QoE is also a function of other parameters, such as buffer level and bitrate. Therefore, Minerva first **formulates** a *bandwidth utility* function

for each client that decomposes the optimization problem into a function of only its download rate.

Following formulation, the competing videos must **solve** the QoE fairness maximization problem in a decentralized manner. Since Minerva cannot change the available capacity, it can control only the relative allocation of bandwidth to each video. It determines this allocation by assigning each video a weight based on the solution of the bandwidth utility optimization, using a custom decentralized algorithm. Then in the **send** step, Minerva utilizes existing congestion control algorithms to achieve a bandwidth allocation for each video proportional to its weight, while also fully utilizing the link capacity. Figure 4-3 illustrates Minerva's high level control flow.

Each video runs Minerva's three-step formulate-solve-send process once every T milliseconds, where T is tunable parameter. Section 4.6.2 details the basic operation of these three steps, including the form of the bandwidth utility functions, the method by which weights are determined, and how existing congestion control algorithms are adapted to achieve those weights. Section 4.6.3 discusses a key optimization to improve performance, Section 4.6.4 explains how Minerva achieves fairness with TCP, and Section 4.6.6 outlines how Minerva can be used on top of a variety of existing congestion control algorithms.

4.6.2 Basic Minerva

Formulating the bandwidth utility. Given the definition of QoE (Equation 4.2), we aim to construct a bandwidth utility function $U(r)$ that is a function of only the client's download rate. $U(r)$ should capture the QoE the client expects to achieve given a current bandwidth of r. In Minerva's basic design, $U(r)$ assumes that the client is able to maintain a bandwidth of r for the rest of the video.

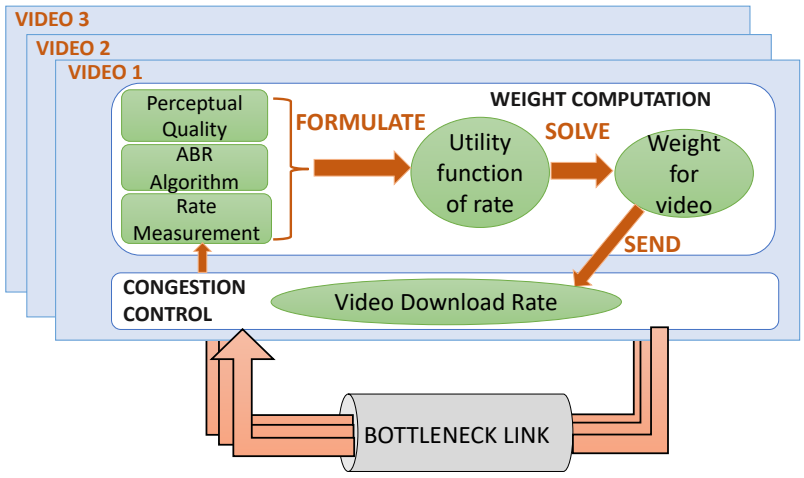


Figure 4-3: Minerva's high-level control flow. Clients run Minerva's formulate-solve-send process independently and receive feedback only through the rate they measure on the next iteration.

Buffer dynamics provide insight into what this QoE value will be. After downloading a single chunk of duration D and size C, the client loses $\frac{C}{r}$ seconds of download time but gains D seconds from the chunk's addition. In the client's steady state, the average change in the buffer level is close to 0 over long time periods (several chunks). During this time, the bitrate averages to approximately r; if r lies between two available bitrates e_i and e_{i+1}, the client switches between those bitrates such that its average bitrate per chunk is r. Therefore, the expected per-chunk QoE is a linear interpolation of the PQ function between $P(e_i)$ and $P(e_{i+1})$ at r. This observation yields a formalization of $U(r)$:

$$U(r) = \begin{cases} P(e_i) & \text{if } r = e_i \\ \texttt{Interpolate}(P, r) & \text{if } e_i < r < e_{i+1} \end{cases} \tag{4.3}$$

assuming the video is available to stream at discrete bitrates $\{e_i\}$. r is the client's

average download rate, measured over the past T milliseconds.

This definition of the bandwidth utility only considers the PQ component of the QoE; it does not take into account client state, such as buffer level, nor does it factor in penalties for rebuffering or smoothness. We present a more sophisticated bandwidth utility function in Section 4.6.3 that does both.

Solving $U(r)$. Given a bandwidth utility function $U_i(r_i)$, which is an estimate of a client's expected QoE, the QoE fairness optimization problem now becomes:

$$\text{maximize } \min_i U_i(r_i)$$

Each client must find a *weight* w_i, such that the set of all client weights determines a relative bandwidth allocation that optimizes QoE fairness. Assuming the U_i are continuous, the solution to the optimization problem occurs when the U_i are all equal and none can be made larger. Minerva relies on the Send step to ensure that clients achieve full link utilization. Therefore, this step focuses only on distributing the link bandwidth to achieve equality between the bandwidth utility functions.

Reaching equality is complicated by the fact that Minerva is completely decentralized: each client is not aware of the utility functions of the others. However, Minerva uses a decentralized algorithm to achieve max-min utility fairness on a bottleneck link, where each video makes decisions based only on its own state. At every timestep, the weight w_i for the next interval is

$$w_i = \frac{r_i}{U_i(r_i)} \tag{4.4}$$

This iterative update rule has two key properties. First, provided a fixed capacity link, the optimal rates $\{r_i^*\}$ are a fixed point. To see why, notice that at the optimal

rates, $\{U_i(r_i^*)\}$ are equal for all clients. Therefore, if the clients are downloading at the optimal rates, after one weight update, the ratio of the new rates will be

$$\frac{r_i'}{r_j'} = \frac{r_i^*/U_i(r_i^*)}{r_j^*/U_j(r_j^*)} = \frac{r_i^*}{r_j^*}$$

Therefore, the rate ratios remain identical. Minerva assumes the link capacity stays constant in the short term, so identical rate ratios will result in identical rates.

Second, each iteration of (Equation 4.4) moves the rates closer towards their optimal values. Consider two clients with ratio $\rho = \frac{r_1}{r_2}$. If $r_1 < r_1^*$ and $r_2 > r_2^*$, then $U_1(r_1) < U_2(r_2)$. Correspondingly, the ratio in the next iteration is $\rho' = \rho \cdot \frac{U_2(r_2)}{U_1(r_1)} > \rho$. Client 1's share of bandwidth will then increase, and Client 2's will decrease, moving the clients closer to $U_1(r_1) = U_2(r_2)$. See Appendix B for a full proof of convergence.

Setting rates. After the Solve step, clients have weights w_i that they must use to achieve rates proportional to w_i, while still fully utilizing the link capacity. A straightforward method to realize these rates is to emulate w_i connections using a congestion control algorithm that achieves per-flow fairness. This makes Minerva capable of building on top of any transport protocol that accepts such a weight. Section 4.6.6 discusses examples of such protocols.

4.6.3 A Client-Aware Utility Function

The basic bandwidth utility function takes into account only the current download rate and the PQ function. However, while this model of QoE is approximately accurate over long time scales, it is overly simplistic for several reasons. First, it is blind to client state. The client holds valuable information, such as buffer level, that influences its future QoE. For example, having a larger buffer may allow the client to receive less bandwidth in the short term without reducing its encoding level. A more

sophisticated utility function should be able to capture the positive value of buffer on a client's expected QoE.

Second, it accounts for only the PQ term in the QoE and ignores the rebuffering and smoothness terms. The basic utility function does not understand that a client with a lower bandwidth or low buffer has a higher likelihood of rebuffering. Additionally, though it expects the client to fetch encodings that average to r, it does not factor in the smoothness penalty between these encodings.

Third, it only looks at future QoE, while ignoring the past. A client that rebuffers early on will afterwards be treated identically to a client that never rebuffered. In order to achieve max-min QoE fairness, the rebuffering client should be compensated with a higher bitrate. Only a utility function that is aware of the QoE of previous chunks can hope to have this capability.

Recognizing the limitations of the basic PQ-aware utility function, we construct a *client-aware* utility function that addresses all three limitations. This new utility function directly estimates the per-chunk QoE using information from past chunks, the current chunk being fetched, and predicted future chunks:

$$U(r) = \frac{\varphi_1(\text{Past QoE}) + \varphi_2(\text{QoE from current chunk}) + V_h(r, b, c_i)}{1 + \varphi_1 + \varphi_2}$$

where φ_1, φ_2 are positive weights that determines relative importance of the three terms. The QoE of the current chunk is estimated by using the current rate r to determine if the video stream will rebuffer. Suppose that a client with buffer level b is downloading a chunk c_i and has c bytes left to download, Minerva computes the expected rebuffering time $R = [c/r - b]_+$ and estimates the QoE of the current chunk as $QoE(c_i, R, c_{i-1})$, where QoE is defined as in Equation 4.2.

One of Minerva's key ideas is V_h, a *value function* computing the expected per-

chunk QoE over the next h chunks, where h is a horizon that can be set as desired. It captures the notion that the QoE a client will achieve depends heavily on the ABR algorithm that decides the encodings. In order to accurately estimate future QoE, clients simulate the ABR algorithm over the next h chunks as follows:

```
r ← Measured download rate
S ← client state
do h times:
  e = ABR(S)
  S, rebuf ← client state and
       rebuffer time after chunk
       e is downloaded at rate r
```

The value function uses the chunk encodings and rebuffer times at every iteration to evaluate the expected QoE. Clients need not modify the ABR implementation; they can simply use it as a black box when computing the value function.

Figure 4-4 shows the value function for MPC [94] with $h = 5$. Note the dependence on rate and buffer. In particular, a large buffer has diminishing marginal returns. The jump in value due to increasing the buffer from 0s to 4s approximately equals the bump due to increasing the buffer from 4s to 20s. This aligns with intuition: clients with a low buffer are at a higher risk of rebuffing and therefore place a high value on each second of buffer.

In general, the value function depends heavily on the underlying ABR algorithm used to compute it, which has two implications. First, it may be expensive to compute. Since the value function for a video does not change across sessions, Minerva precomputes the value function before streaming begins (Section 4.7). Second, the value function may not exactly satisfy the convergence conditions of Minerva's rate

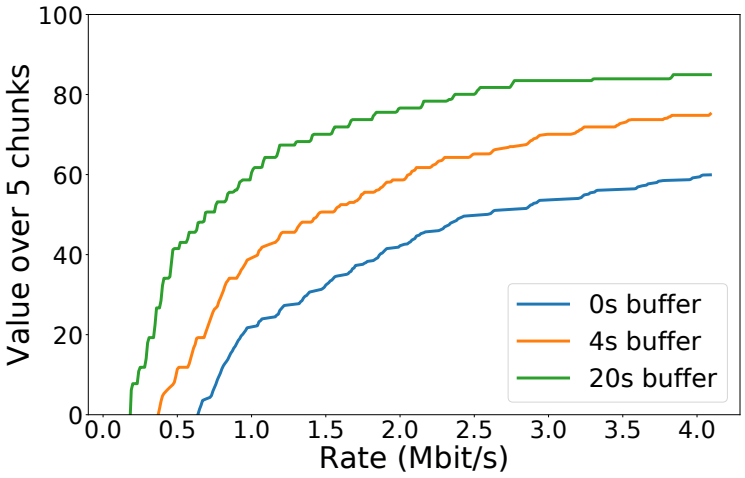


Figure 4-4: The value function simulated with MPC for a range of buffers and rates. Higher rates and buffers yield a higher value, and larger buffers have diminishing marginal utility.

update rule. Appendix C discusses methods to overcome this.

4.6.4 Normalization: Fairness with TCP

Section 4.6.2 describes how Minerva flows competing with each other converge to the correct bandwidth allocation. However, in the wild, they will have to also compete with, and be fair to, other TCP flows (e.g., Cubic), including Minerva flows from different video providers. In this section, we describe how Minerva flows achieve fairness with TCP while optimizing the relative bandwidth allocation between themselves.

As discussed previously, flows updating their weights using Equation 4.4 converge to a steady state where the $U_i(r_i)$ are equal. This convergence is not specific to the exact functional form of U_i; in fact, we can replace $U_i(r_i)$ with $f(U_i(r_i))$ if the function f is the same across all Minerva clients. As long as f is monotonically increasing and $f(U_i(r_i))$ satisfy the loose requirements outlined in Appendix B, all clients will still

reach a steady state such that the $f(U_i(r_i))$ are equal; the monotonicity of f then implies that the $U_i(r_i)$ are also equal.

We take advantage of this flexibility by choosing an f that allows Minerva flows, in the steady state, to collectively occupy their fair share of the link bandwidth. In particular, for each Minerva flow to occupy the equivalent of α TCP flows on average, we *normalize* their weights, so they average to α:

$$\frac{1}{N}\sum_i \frac{r_i}{f(U_i(r_i))} = \alpha$$

where N is the number of flows on that link, and we typically choose $\alpha = 1$. Since all $U_i(r_i)$ converge to the steady-state value u, the resulting form of f is:

$$f(u) = \frac{1}{\alpha N}\sum_i U_i^{-1}(u)$$

In practical deployments of Minerva, *clients are not aware of N or the utility functions used by competing flows.* Therefore, we compute $f(u)$ using a popularity *distribution*, which video providers can measure. If video i has popularity p_i, where $\sum_i p_i = 1$, then:

$$f(u) = \frac{1}{\alpha}\sum_i p_i U_i^{-1}(u)$$

We call f^{-1} the normalization function instead of f because f^{-1} has the same signature as the U_i and can be compared to them. Figure 4-1 shows a diverse set of PQ curves along with their normalization function.

If this popularity distribution over videos is accurate, then on average, N randomly sampled Minerva flows will converge to the same bandwidth as N TCP flows, giving Minerva the property of fairness to non-Minerva flows. The normalization function is

also how Minerva isolates videos from different providers: if two providers A and B have videos sharing a bottleneck link, and each provider uses a suitable normalization function based on the popularity distribution of its own videos, then A's videos will occupy their fair share on average, as will B's. Therefore, videos from one provider do not affect the other.

It's important to note that Minerva achieves fairness with competing traffic *in expectation* over its video distribution. At any particular bottleneck, the aggregate bandwidth consumed by Minerva is determined largely by the PQ curves of the videos in question. Consider client i playing a video whose PQ curve lies above f, i.e. $U_i(r_i) > f(r_i)$. This client will compute $f^{-1}(U_i(r_i)) > r_i$, so $w_i < 1$, and it will occupy less than its fair share of bandwidth. The converse holds for videos with PQ curves under f. However, over several samples from the video distribution, Minerva's aggregate bandwidth footprint is fair to competing traffic, by the law of large numbers.

The function f maps utilities back to rates, so the weight computed by each Minerva flow is unitless. As a result, weights from all clients are comparable to each other, *even if* those clients are using different QoE definitions that cannot be compared (e.g., PSNR and SSIM). This property is crucial to the design of Minerva: different video providers using Minerva can properly compete with each other without sharing information about their utility functions.

4.6.5 Generalizing Max-Min Fairness

Practically, providers may not want any of their videos to consume *too* much more (or less) bandwidth than their equal share. Minerva allows them to optimize max-min fairness *subject to* constraints on the video's bandwidth. This is accomplished by setting upper and lower bounds on the weight w_i of each client. The weight has a

straightforward interpretation: it is the ratio of that client's rate to a standard Cubic flow, on average. Keeping it from straying past a bound μ ensures that its rate is never μ times more than its equal link share.

Alternatively, Minerva is able to optimize for different notions of fairness, such as proportional fairness. We relegate further discussion of proportional fairness to Appendix D.

4.6.6 Using Existing Transport Protocols

Section 4.6.2 describes how the first two steps (formulate and solve) result in a weight w_i for each flow that Minerva must translate to a rate proportional to w_i, while ensuring that the flows achieve full link utilization. To accomplish this, Minerva can use any existing transport protocol that accepts such a weight and can achieve a bandwidth share proportional to that weight. Abstracting out the congestion control layer is valuable, since there are many protocols that have this property and can be plugged into Minerva. We discuss two such protocols here: Cubic [30], which is loss-based and allows Minerva to compete with wide area traffic, and FAST [44], a delay-based congestion control scheme.

For the sake of deployability, we choose Cubic [30], as our primary underlying congestion control mechanism. Minerva emulates multiple Cubic connections using a technique similar to MulTCP [15]: a client emulating w connections counts every ACK towards all w flows but counts a loss against only a single flow. Note that w need not be an integer for this to work. The average throughput of a Cubic flow as a function of loss probability p is [41]:

$$T = C \left(\frac{3+\beta}{4(1-\beta)} \right)^{1/4} \left(\frac{RTT}{p} \right)^{3/4},$$

where β is Cubic's multiplicative decrease factor, i.e., the factor by which Cubic cuts its congestion window after a packet loss. Substituting β with

$$\beta' = \frac{\beta(w+1) + w - 1}{\beta(w-1) + w + 1}$$

multiplies the throughput by the desired factor of w.

We also describe how to implement Minerva on top of the delay-based FAST, which has been shown to achieve faster convergence times than Cubic [44]. Faster convergence times of the underlying congestion control protocol mean that the measurement interval T can be shortened, resulting in faster convergence of Minerva's decentralized algorithm.

A client with weight w_i sets its congestion window to

$$\texttt{cnwd}' = \left(\frac{\text{RTT}}{\text{RTT}_{min}} \cdot \texttt{cwnd} + \kappa w_i \right) \tag{4.5}$$

This attempts to keep κw_i packets in the bottleneck queue, for some positive constant $\kappa > 1$, and can be shown to achieve an allocation proportional to $\{w_i\}$.

4.7 Implementation

Minerva is implemented using a DASH Video client [2] and a HTTPS video server running over QUIC [50]. Figure 4-5 shows the overall flow of data in Minerva. The QUIC server runs Minerva's weight update algorithm and uses that weight to adjust the download rate (equivalently, the server's sending rate) via the underlying congestion control protocol. In order to send the necessary state from clients to the server, e.g. playback buffer and past QoE, the client piggybacks its state onto the HTTP chunk requests as URL parameters.

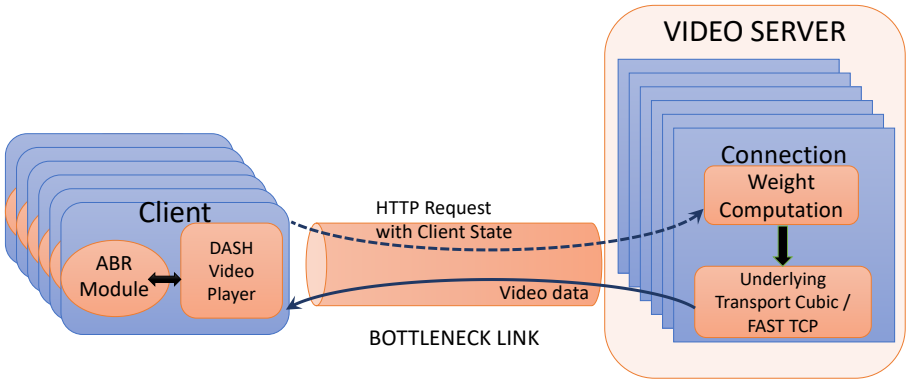


Figure 4-5: System architecture for Minerva. Clients run MPC and convey their state to the QUIC video server via chunk requests. The server is responsible for setting the download rate.

4.7.1 Client

In Minerva, video clients are headless Google Chrome [27] browsers (Version 62) watching their video using a DASH video player. Chrome was configured to send HTTP requests to the server using QUIC. We use a modified version of dash.js (Version 2.4) [2] with a ABR module running MPC [94]. The player requests video chunks via HTTPS, and is responsible for tracking the playback buffer, rebuffer time, QoE from previous chunks, chunk download times (for bandwidth estimation), and other client parameters, such as screen size and device. This information is sent to the ABR module, which responds with the next quality for the client to fetch. Note that the client's bandwidth measurement is only used to make the ABR decision and is not sent to the video server; the server makes its own higher resolution rate measurements. However, all other data listed above are appended on video chunk requests.

4.7.2 Video Server

Choosing a QUIC-based server made it easy to test rate control changes. In addition to updating congestion control specific parameters, like the congestion window, the video server was also responsible for tracking client state and performing the weight update computation from Section 4.6.2. QUIC uses a persistent connection for each client, preserving state across requests made on that connection. Therefore, when video clients request a new chunk, congestion control information, such as congestion window, is carried over from the previous chunk request, preventing clients from having to re-enter slow start. Persistence grants Minerva enough time (several chunks) to converge to the QoE-fair rate allocation.

Rate Measurements. In order to predict the expected QoE from the current chunk and future chunks, the video server requires rate measurements at finer granularity than once per chunk. Section 4.6 notes that weight updates are made every T milliseconds, independently for each flow. Following a weight update, the server waits $\frac{T}{2}$ milliseconds to allow the flow to converge to its new rate, and then measures the number of bytes acked over the next $\frac{T}{2}$ milliseconds to estimate the current instantaneous rate r_i. We use $T = 25 * \text{RTT}_{\text{min}}$, to allow time for convergence. In practice, we found that this instantaneous rate measurement can fluctuate and be noisy. Rate measurements that exceed the true rate cause the client to overestimate its utility and drop its weight significantly; this increases the chance of rebuffering. To mitigate this problem, we compute a *conservative* rate estimate $r_{i,cons} = \max(0.8r_i, r_i - 0.5\sigma)$, where σ is the standard deviation of the last 4 rate measurements. Each client uses their conservative rate estimate to calculate their utility.

Specifically, Minerva computes the weight update based on Section 4.6.2 with two modifications. First, $w_i = \frac{\tilde{r}_{i,cons}}{f(U_i(\tilde{r}_{i,cons}))}$, where $\tilde{r}_{i,cons}$ is an exponential weighted

moving average (EWMA) over the conservative instantaneous rate:

$$\tilde{r}_{i,cons} = 0.1 r_{i,cons} + 0.9 \tilde{r}_{i,cons}$$

Second, we apply another EWMA, $\tilde{w}_i$, to the computed weight:

$$\tilde{w}_i = 0.1 w_i + 0.9 \tilde{w}_i$$

Minerva passes $\tilde{w}_i$ as the weight to the send step. Minerva clamps this weight to the interval $[0.5, 20]$, although this is only as a precaution; we do not observe the weight passing these bounds in practice.

Control Flow. Clients only begin updating weights after downloading the first chunk, which is always fetched at the lowest quality. This allows the client's bandwidth share to stabilize before starting Minerva's rate update algorithm. Starting on the second chunk, when a new rate measurement is available after T milliseconds, Minerva allows clients to update their weights. When computing the utility function, we use $\varphi_1 = \frac{1}{N}$, where N is the number of chunks already played, and $\varphi_2 = 1$. This equally weights the contribution from past QoE, current chunk, and value function.

Function Evaluation. Minerva's design involves evaluating both a value function, to estimate the QoE of future chunks, and a normalization function. To save computation time when running the 'Solve' step, these functions are pre-computed before running the video server. Recall that the value function $V(r, b, e_i)$ is a function of the download rate, buffer level, and the bitrate currently being fetched. We precompute the value function for every chunk, for each combination of rates, from 0 to 8 Mbit/s in intervals of 100 kbit/s; buffers, from 0 to 40 seconds in intervals of 0.05 seconds; and previous bitrates. This amounts to 38.4 million values for a single video, which is

approximately 153MB. We losslessly compress the value functions by storing them as a series of contiguous line segments, reducing the space overhead to between 6MB and 16MB, depending on the video. We discuss ways to reduce this overhead further in Section 4.8.

The normalization function is precomputed from the videos being watched before the videos start, using the PQ curve for each video. Although normalization ensures that Minerva converges to its fair share of bandwidth, this may not be true during the convergence process. To compensate for the higher bandwidth share occupied during convergence, we use $\alpha = 1.65$ (QUIC's Cubic implementation emulates 2 flows). The normalization function is stored as a table of values and is loaded when the video server launches.

Tracking client state. The QUIC server keeps an estimate of the client's buffer, which it uses to predict the QoE expected from downloading the current chunk. This estimate is updated after every chunk request with the true buffer size, which the client sends as a URL parameter on the HTTP request. We find that DASH incurs an overhead of 0.6 seconds for each chunk, so Minerva decrements the estimate by 0.6 seconds before using it in its weight computation. In addition, the client sends the total QoE it has experienced for all chunks watched so far, which the server incorporates into the utility function.

Overhead. The overhead added by Minerva's computation on the video server is negligible: we observed no quantifiable increase in CPU usage when running Minerva as compared to QUIC Cubic.

4.8 Discussion

Global vs Local Fairness. Minerva guarantees that videos are fair to Cubic in aggregate over a set of links, provided that the normalization function is crafted

from the popularity distribution of videos over those links. However, popularity distributions may vary both geographically and over time. In particular, the global distribution over all videos may differ substantially from the videos being watched in a particular geographic area, and may additionally vary from morning to evening. Therefore, if the provider bases their normalization off a single global popularity distribution, videos will be fair to Cubic on average *globally* but may not achieve *local* fairness. Achieving global fairness at the expense of local fairness may not be desirable, since videos playing in the same region may then achieve significantly more or less than their fair share of bandwidth. Providers have two remedies to retain control over the weights of their videos and prevent them from straying too far from their fair-share allocation.

First, providers can achieve local fairness by using a different normalization function in each geographic region, which accurately captures the popularity of videos in that particular area. The more granular the popularity distribution, the more likely it is that videos in the corresponding area closely match the distribution, and the less Minerva videos stray from their fair-share allocation. Taken to the extreme, if providers knew precisely which sets of videos shared common bottleneck links, they could guarantee that their videos achieve their fair share allocation on every link. In practice, obtaining link-level statistics is difficult; however, it may be more feasible for providers to approach local fairness by specializing their normalization functions to less granular regions.

Second, Minerva exposes the client's weight w_i, so providers may cap it at a particular value to limit how far a video's bandwidth strays from its equal share. After normalization, w_i has a straightforward interpretation: it will occupy w_i times what a standard Cubic flow would occupy. Keeping w_i between 0.5 and 2, for example, ensures that no video grabs less than half or more than twice its fair share, respectively.

However, a more stringent cap limits the range in which Minerva can operate, so QoE fairness may suffer. We evaluate this approach in Section 4.9.6.

Deployment Considerations. Minerva requires two types of modifications to video servers. First, Minerva requires servers that keep track of application state for each video flow they serve and have the ability to adjust their sending rates according to that application state. This may mean that conventional CDNs, which are stateless and implement a traditional TCP stack, are ill-suited for serving Minerva videos. However, some providers, such as Netflix, already deploy video servers with custom software [67].

Second, Minerva's current implementation precomputes the value functions and stores them on the video server to avoid the overhead of real-time computation. This incurs a memory overhead of 16MB per video, and may be prohibitive when a single server serves a large number of videos simultaneously. There are multiple approaches to reducing this overhead. First, we can compute the value function at a coarser granularity, e.g., for buffers at every 0.5 seconds instead of 0.05 seconds. Second, the video clients can compute the value function on demand and send them to the video server when requesting a chunk. Recall that for any given chunk, the value function $V(r, b, e_i)$ depends on the rate, buffer, and previous bitrate. Since the client knows both its current buffer level B and previously played bitrate E, it sends $V(r, B, E)$, now only a function of r, for a handful of buffer values close to B. We estimate the size of this representation to be around 2kB per chunk.

4.9 Evaluation

Minerva has two design goals: (1) maximizing QoE fairness while (2) occupying its fair share of the link capacity. We start by isolating the first goal and asking how well Minerva is able to improve QoE max-min fairness in the absence of competing

traffic (Section 4.9.2), including dynamic environments with videos arriving and leaving at different times (Section 4.9.3). To test the second goal, we add cross-traffic, and show that Minerva shares bandwidth fairly with other videos, scales to many clients (Section 4.9.4), and adapts to link conditions that vary with time (Section 4.9.4, Section 4.9.4). Third, we show that Minerva works in the wild, over a real residential network (Section 4.9.5). Fourth, we explore how Minerva can optimize for alternatives to max-min fairness (Section 4.9.6). Fifth, we consider the case of two providers competing with Minerva (Section 4.9.7).

4.9.1 Setup

System Details

All clients use dash.js (version 2.4) running on Google Chrome (version 62), with QUIC support enabled. Each client runs in a headless Chrome browser, on an Amazon AWS r5a.4xlarge EC2 instance. The server is based off the HTTP server distributed with Chromium, modified only to implement Minerva's rate control scheme.

Clients choose the bitrate of the next chunk by contacting an ABR server that implements MPC. The ABR server is colocated on the same machine as the clients and is shared by all of them. The overhead of a client's request to the ABR server, including latency to/from the server and computation time, is 70ms seconds, which is small relative to the length of a video chunk download (4 seconds). Unless otherwise noted, clients begin watching their videos at the same time, and fairness is computed over a 200 second interval from the start of the video.

The bulk of our evaluation considers Minerva videos sharing a single bottleneck link, emulated with a Mahimahi [68] shell. The link has a minimum RTT of 20ms and a buffer with a capacity of 1.5 bandwidth delay products (BDPs). The link runs a PIE [74] AQM scheme for both Minerva and baseline flows, with a target delay of

15ms.[4] However, we also evaluate over droptail buffers. Since all video traffic travels from server to clients, our evaluation only alters the Mahimahi downlink (outside to inside) capacity; the uplink is fixed at 10 Mbit/s and is never saturated.

Videos

Our corpus consists of 19 4K videos between 4 and 5 minutes long, which we label V1 to V19. They span a diversity of genres, including news, action, and animation. Table 4.1 is the list of videos in our corpus with their genres. The labels correspond to those in Figure 4-1.

Name	Description
V1	Aerial Footage
V2	Nature
V3	Gaming Livestream
V4	Cooking / Nature
V5	Advertisement (GoPro)
V6	Soccer Match
V7	Action Movie Trailer
V8	Animated Music Video
V9	Animated Short
V10	Tornado Footage
V11	Cat Video
V12	Animated Short
V13	News (Video Blog)
V14	Lecture
V15	Action Movie Clip
V16	Video Game Trailers
V17	Music Video
V18	Advertisement (Apple)
V19	News (Documentary)

Table 4.1: The 19 videos used in our evaluation corpus.

[4]The PIE AQM scheme is a part of the DOCSIS3.1 standard and widely deployed in residential areas: Comcast has already deployed DOCSIS3.1 compliant routers to 75% of their customers [29, 63].

Each video has a corresponding *VMAF score*, a perceptual quality metric designed to predict user perceived quality [52]. We use the VMAF score (version 0.3.1) as the PQ function for each chunk.

For a single video, both the chunk sizes and VMAF scores differ from chunk to chunk, even at the same bitrate. Figure 4-6 shows the variation in VMAF scores in video V9. Note that while adjacent chunks may have very different qualities, there is also a temporal trend over several chunks. This property makes it important for Minerva to dynamically adjust rates throughout the video.

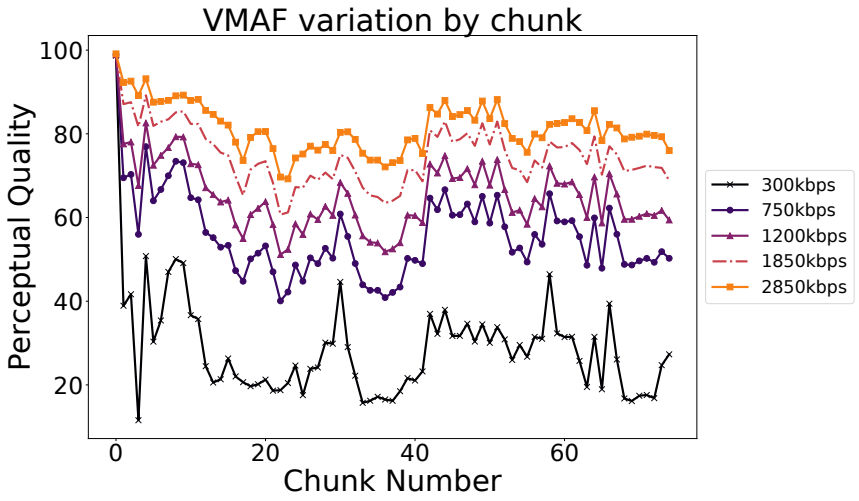


Figure 4-6: VMAF scores for all chunks in video V9, shown for the 5 lowest bitrates.

Since the videos span several genres, the VMAF scores also vary significantly between videos. Figure 4-1 visualizes the score of the 19 videos, averaged over all chunks, at the eight available bitrates. Our results are presented in terms of VMAF scores. To contextualize these numbers, Figure 4-7 maps popular resolutions to their corresponding scores on our corpus. For example, a bump from 720p to 1080p equates to a gain of 7.65 points. We use this as a benchmark for our evaluation of Minerva, since VMAF scores have a close-to-linear correlation with user-perceived quality [52].

That is, a delta in VMAF accurately predicts a delta in viewer experience, regardless of bitrate or video genre. Consequently, any delta of 7.65 points is comparable to the jump in quality a user would perceive between 720p and 1080p on a 4K TV, uniformly along the bitrate spectrum.

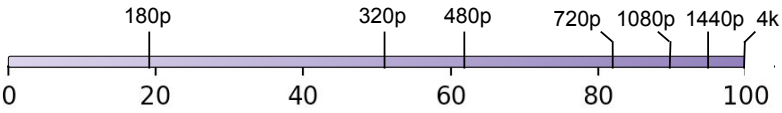


Figure 4-7: VMAF scores for well known resolutions, averaged across our corpus. A gain of 7.65 corresponds to a bump from 720p to 1080p on a 4k TV at standard viewing distance.

Metrics

We use the definition of Quality of Experience defined in Equation 4.2. The VMAF score $P(c_k)$ ranges from 0 to 100, while our QoE metric uses a rebuffering penalty of $\beta = 25$ and a smoothness penalty of $\gamma = 2.5$. We evaluate Minerva on max-min QoE fairness.

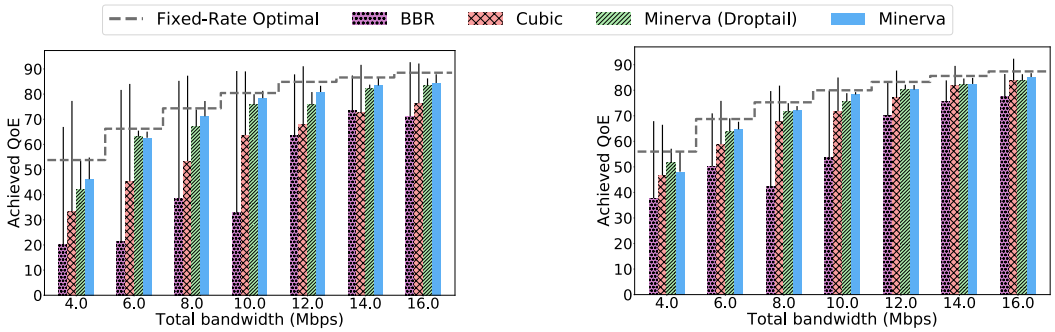
Our baseline is a client running unmodified dash.js over Cubic. Since Minerva is implemented over QUIC, we use the Cubic implementation provided by QUIC, which simulates 2 connections by default. We call this system "Cubic".

4.9.2 Benchmarking QoE Fairness

We first evaluate Minerva's ability to improve QoE fairness, by playing videos over a fixed emulated PIE-enabled link with capacities ranging from 4 Mbit/s to 16 Mbit/s. We conduct a total of 44 runs, in each one selecting 4 distinct videos uniformly at random from our corpus. We play the same videos over four additional benchmarks:

1. QUIC Cubic running over a PIE-enabled link.

2. Minerva over a link with a droptail buffer of 1.5 BDPs.



(a) Videos V2, V7, V18, V19

(b) Videos V1, V2, V6, V10

Figure 4-8: Max-min QoE fairness for protocols over a constant link. The black whiskers extend from the minimum to maximum QoEs.

3. BBR, running over a droptail link sized to 1.5 BDPs.

4. The "Fixed-Rate Optimal", computed offline, which is the max-min QoE fairness assuming that each video receives a constant rate over the course of the video. This optimal changes for each video combination, since it depends heavily on the PQ curves of the videos involved.

Figure 4-8 shows two representative combinations of videos with these points of comparison. There are four main takeaways.

First, the magnitude of improvement of Minerva over Cubic depends on the link bandwidth and the particular videos used. Videos whose PQ values are far from each other at a particular link rate require a larger bandwidth difference to achieve the same QoE, creating more room for Minerva to surpass Cubic. For example, Figure 4-8a and Figure 4-8b show average improvements of 12.5 and 3 points, respectively. At high link rates, e.g. 16 Mbit/s, the difference between videos' PQ curves is typically smaller, so Cubic's allocation is closer to optimal and there is less room for Minerva to improve. Conversely, at low link rates, e.g. 4 Mbit/s, or 1 Mbit/s per video on average, the gap between utilities is large. However, VMAF curves are steep at those

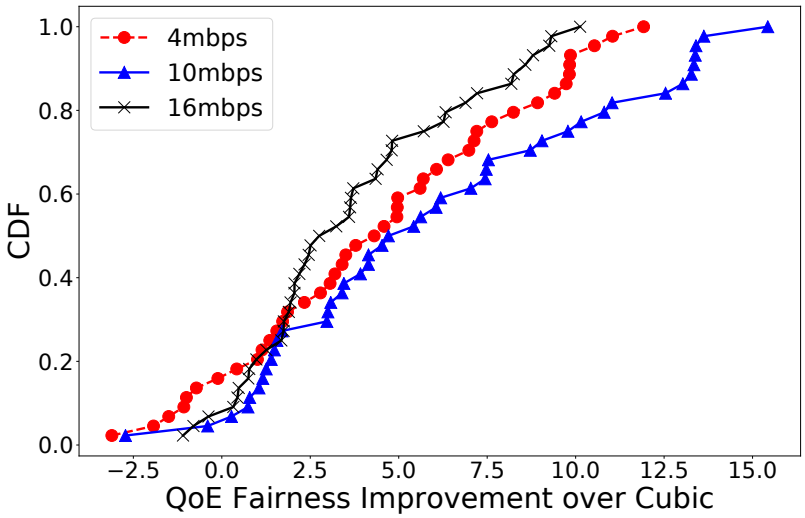


Figure 4-9: Minerva's improvement in QoE fairness over Cubic over 44 runs, each using 4 distinct videos sampled from our 18-video corpus.

low rates; in order to improve the worse-off video, QoEs of other videos must drop sharply. As a result, the QoE fairness gain is small. We observe the largest gains on a 10 Mbit/s link, where videos' utilities have a sufficiently large difference with gentler slopes.

Figure 4-9 shows the gains Minerva is able to achieve over Cubic on a variety of different video combinations and links. The videos are chosen uniformly at random without replacement from our corpus, and are played over a 4 Mbit/s, 10 Mbit/s, and 16 Mbit/s link. To put the magnitude of Minerva's improvement over Cubic into perspective, consider that the average difference between 720p and 1080p is 7.65 VMAF points, on a scale of 100. For 24% of cases across our corpus (24%, 32%, and 16% on 4 Mbit/s, 10 Mbit/s, and 16 Mbit/s, respectively), the worst-performing video client sees a boost in viewing experience equivalent to or better than a jump in resolution from 720p to 1080p.

The second takeaway is that Minerva substantially closes the gap between Cubic and the optimal allocation. For videos like Figure 4-8a, Cubic's gap to optimal is 17.5 points on average, which Minerva reduces to 4, an improvement of 77%. For videos with more similar PQ curves (Figure 4-8b), Cubic's bandwidth split is closer to optimal (6.5 points), but Minerva is still able to bring this down by 53% (3 points).

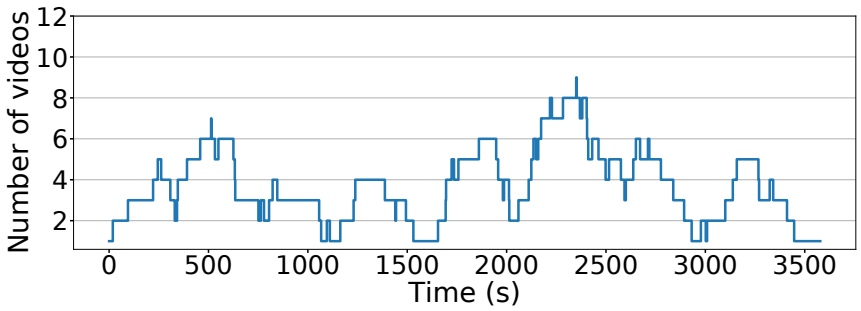
Third, Minerva's improvements over Cubic do not hinge on using PIE; it performs nearly identically over droptail, with only a marginal reduction in fairness. This reduction is 2.5 points and 0.5 points on average, respectively, over the scenarios in Figure 4-8. Cubic relies on independent drops in order to give weighted Cubic flows their expected bandwidth share. Links with droptail queues result in drops that are bursty and correlated, a possible barrier for videos to achieve their desired rate ratios.

Fourth, BBR may achieve better link utilization [19, 10] but it performs poorly from a fairness perspective. Some BBR flows, determined seemingly arbitrarily, grab a larger bandwidth share, starving the others. This observation matches previous findings that BBR does not achieve fairness with itself [33].

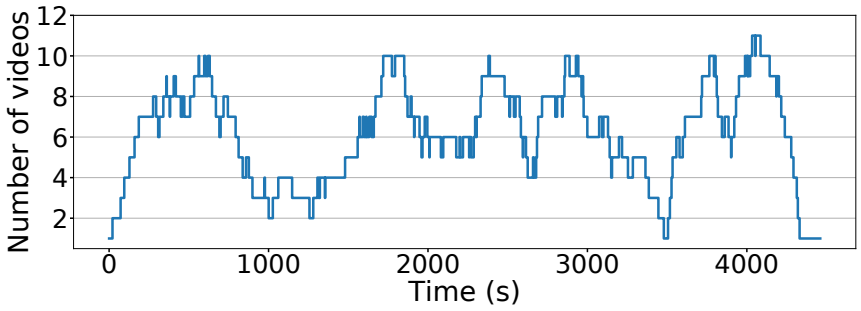
4.9.3 Minerva in a Dynamic Environment

We now evaluate Minerva in a setting where videos sharing the same link start and stop over the course of more than an hour. In contrast to Section 4.9.2, in which all videos started simultaneously and played throughout the entire experiment, each video starts at a randomly determined start time and then plays to completion.

We sample 48 videos uniformly at random from our corpus to play over a link with a constant total capacity of 8 Mbit/s. Video start times are determined by a Poisson process such that the average number of videos playing simultaneously matched a given number, either 4 or 8, depending on the experiment. Videos have similar runtimes, between 270 and 300 seconds. Figure 4-10a shows the number of



(a) 4 videos at a time on average, sampled uniformly from the corpus.

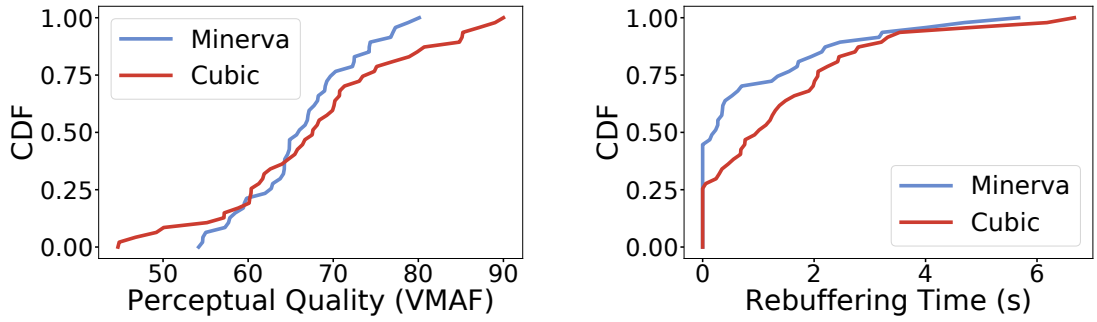


(b) 8 videos at a time on average, all identical (V11).

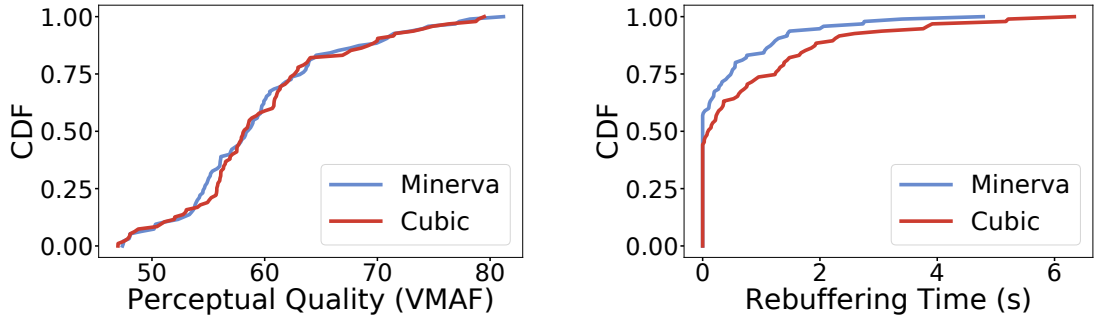
Figure 4-10: The number of videos sharing the link over time in the dynamic environment. All videos are between 4 and 5 minutes long.

videos sharing the link at each point in time during the experiemnt, sampling videos randomly from the corpus, such that there were 4 playing simultaneously, on average. Figure 4-10b shows a process targeting 8 videos sharing the link on average. We run the same configuration of videos using both Minerva and Cubic as the underlying transports.

Since Minerva siphons bandwidth away from videos with a high perceptual quality, one might expect those videos to be at higher risk for rebuffering. This additional rebuffering may not be reflected by max-min QoE fairness, which considers only the



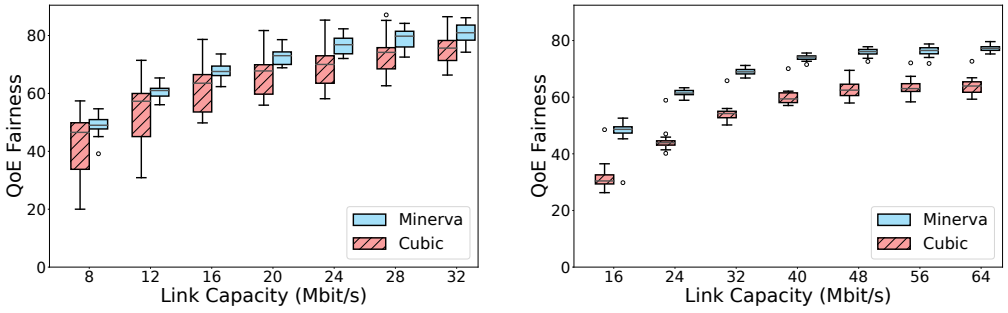
(a) Dynamic environment with randomly sampled videos.



(b) Dynamic environment with identical videos.

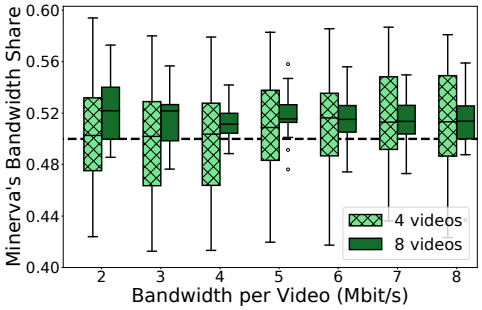
Figure 4-11: Minerva's effect on visual quality and rebuffering time in a dynamic setting, with videos joining and leaving.

video with the worst quality. Therefore, Figure 4-11a examines the effect of Minerva on *both* the perceptual quality and rebuffering time, considering the distribution over all videos. In our setup, every video fetches the first chunk at the lowest bitrate and must stall until it finishes downloading. Additionally, Minerva's rate control does not kick in until the second chunk. We therefore focus on *non-startup rebuffering time*, i.e. the amount of time the video spends stalling when downloading any chunk after the first. Unlike startup delay, non-startup rebuffering time materially interrupts the viewing experience and is visually jarring for users.



(a) 4 videos each.

(b) 8 videos each.



(c) Minerva's aggregate bandwidth share

Figure 4-12: Minerva improves QoE fairness when competing with Cubic, each using 4 (a) and 8 (c) videos. Minerva's bandwidth share, as a fraction of total traffic, for these videos is close to equal (b), indicating that it is fair to cross-traffic.

Minerva increases the minimum QoE by 9.3 VMAF points, while simultaneously reducing rebuffering time across the board: average total rebuffering time (including startup delay) decreases by 17%, while the average non-startup rebuffering time falls by 38%. The drop in rebuffering time is due to Minerva's use of a buffer-aware utility function that captures the negative effects of rebuffering. By understanding when a client is at risk for rebuffering, Minerva can increase that client's weight, improving its bandwidth share and growing its buffer. This allows videos to tap into the *global buffer pool* formed by the other clients sharing the bottleneck link.

Buffer Pooling. The results in Figure 4-11a combine improvements due to both

static attributes, such as the differences in PQ values between videos, and *dynamic* factors like buffer level, which vary across sessions for the same video. To isolate the potential of the global buffer pool, we repeat the same long-running experiment, but with all clients watching the same video. This eliminates the potential for QoE improvements arising from static differences, and focuses on the ability of Minerva to take advantage of dynamic differences in buffer size among clients sharing the same link. Additionally, to place clients in challenging network conditions where rebuffering is more likely, the Poisson process that governs video start time is adjusted so that 8 videos share the link at any given time, on average.

Figure 4-11b shows that, by harnessing the buffers of other clients, Minerva is able to significantly reduce time spent rebuffering, but does not hurt the video's perceptual quality in the process. Minerva reduces average visual quality by only 0.4% compared to Cubic, but is able to cut average non-startup rebuffering time by 47% and the number of rebuffering events by 45%. This shows that Minerva can achieve notable gains in viewing experience that do not just stem from perceptual quality differences. In particular, even if information about the videos' perceptual qualities is absent or videos are perceptually similar, Minerva can still provide substantial gains by tapping into the global buffer pool.

4.9.4 Fairness with Cross Traffic

Minerva should improve QoE fairness without violating its second design goal: achieving an equal bandwidth split with competing traffic, on average. That is, a collection of N Minerva flows should occupy the same bandwidth as N Cubic flows. To evaluate Minerva on connection-level fairness, we play multiple Minerva videos, again chosen uniformly at random, simultaneously with the same N videos running over Cubic. All videos start at the same time and play for the duration of the experiment, as in

Section 4.9.2. The reasons for choosing video cross-traffic are twofold. First, even videos running over Cubic are not able to achieve fairness with long-lived Cubic flows, due to the idle periods between chunk requests [96]. Pitting Minerva against other video flows evens the playing field. Second, the QoE fairness achieved by the Cubic videos serves as a convenient point of comparison for Minerva.

Figure 4-12a shows that Minerva videos achieve an improvement in median QoE fairness of 5 points over Cubic. 42% of cases result in a 7.65 improvement, corresponding to the same perceptual jump between 720p and 1080p. Minerva's gains in this setting should be viewed in the context of its bandwidth share, i.e. its fraction of the total traffic throughput (Figure 4-12c). Minerva maintains a 75th percentile bandwidth share that is within 5% of a perfectly even split with Cubic across the board. However, we note that slightly uneven bandwidth shares may be partly responsible for Minerva's QoE fairness gains exceeding those in the setting without cross-traffic.

Although Minerva's median QoE fairness is higher, Cubic still attains a maximum fairness that exceeds Minerva's. These are due to runs in which all videos have PQ curves that lie above the normalization function. In these particular cases, Minerva occupies less than its fair share of bandwidth, which reduces QoE of all videos. The converse is true when all the videos lie under the normalization curve. These situations, while they exist, constitute only 12% of all cases.

The gap between Minerva and Cubic only improves when more clients are added (Figure 4-12b). The median improvement is 14.9 points when Minerva and Cubic run 8 videos each; for perspective, a change from 360p to 480p is an improvement of only 11 points. A full 95% of cases see an improvement larger than the 7.65 point gap between 720p and 1080p. The large improvement is due to this larger set of videos having a higher chance of including two videos with significantly different PQ

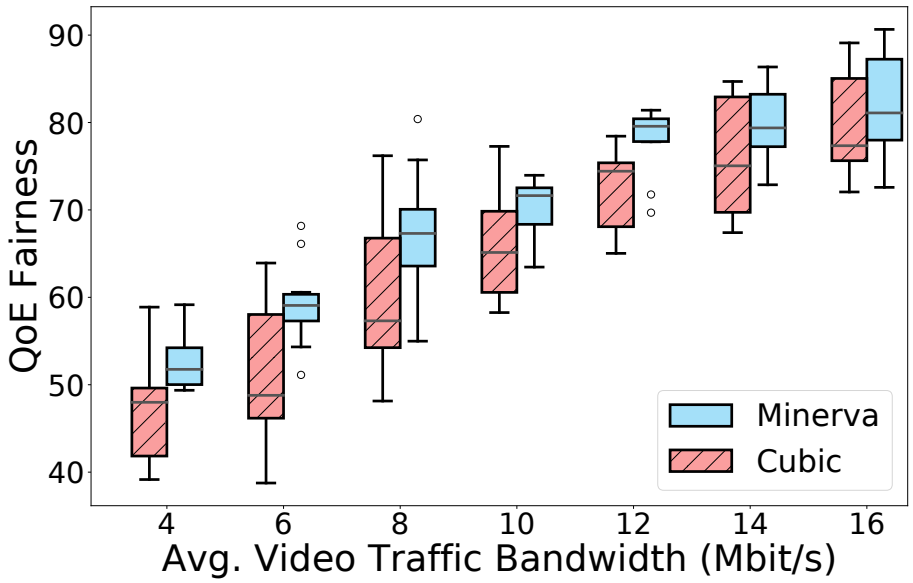


Figure 4-13: QoE fairness achieved by Minerva and Cubic, each playing 4 videos, over 10 runs on a 20 Mbit/s link with various loads of emulated web cross traffic.

curves; this creates a large room for improvement over Cubic. A larger number of videos also means that the average PQ curves of the videos more closely matches the normalization function, so Minerva bandwidth share is closer to 50% with lower variance. (Figure 4-12c).

Variable Cross Traffic

We now shift from video cross-traffic to web cross traffic, still over an emulated link. This experiment pits Minerva against cross-traffic it is likely to encounter in practice, and tests its performance over link conditions that vary with time. Time varying links stress Minerva's future QoE estimation, which assumes that the client will continue to see the rate they most recently measured for the next several chunks.

We emulate web traffic from 200 servers, drawing flow sizes from an empirical distribution derived from CAIDA data [1]. The cross traffic's average offered load

varies between 4 and 16 Mbit/s on a 20 Mbit/s emulated link. We perform 10 runs of Minerva with each cross-traffic load, using 4 distinct videos each time. For comparison, we separately run Cubic over identical loads and videos. Minerva and Cubic achieve similar aggregate bandwidths, so Minerva's gains do not come at the expense of competing traffic. To demonstrate this, we show an example of Minerva behavior when competing with a variable wide-area workload. For reference, we show Cubic competing with the same workload. Figure 4-14 highlights the extent of variability in our workload and demonstrates that Minerva tracks Cubic's behavior closely in terms of aggregate bandwidth.

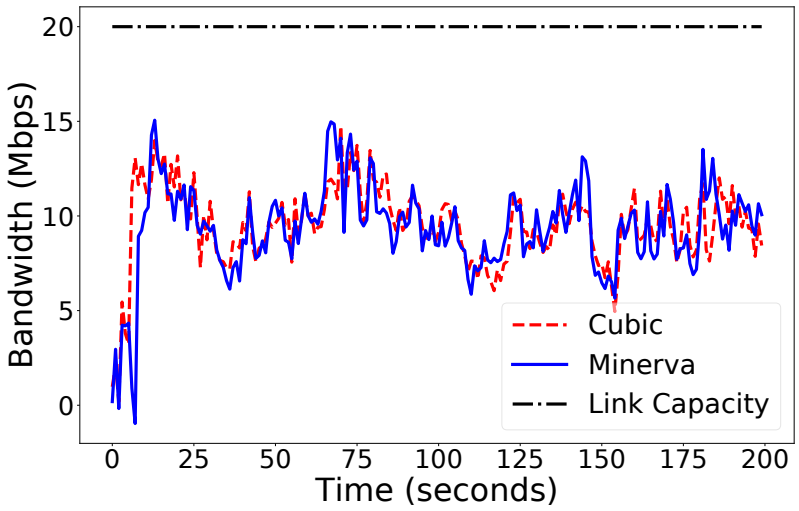


Figure 4-14: Achieved aggregate bandwidth across all video flows when 4 Minerva or Cubic flows are running on a 20Mbps link along with web cross traffic that consumes 10Mbps on average.

We observe that Minerva's gains do not suffer when running over a time-variable link. The distribution of Minerva's QoE fairness is more diffuse, due to the workload variability. However, across all tested loads, 37% of video combinations hit our

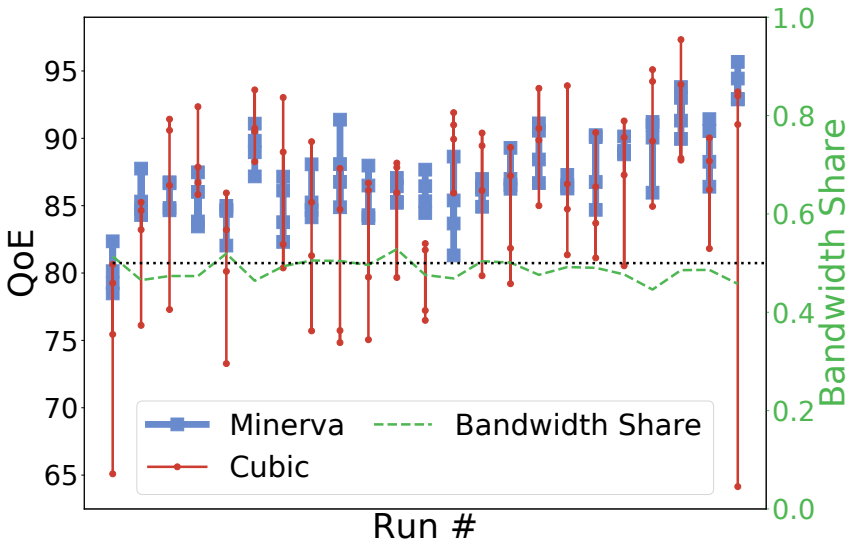


Figure 4-15: QoEs of Minerva and Cubic videos for 23 runs over a real residential link, each with randomly selected videos. Runs are ordered by increasing total bandwidth.

benchmark of 7.65 points. We conclude that Minerva is still able to maintain its QoE fairness improvements and equal-bandwidth guarantees even in variable link conditions.

4.9.5 A Real Residential Network

The experiments thus far have tested Minerva in a controlled environment using emulated links. To test Minerva in the wild, we run it over an actual residential WiFi link during both peak evening and non-peak hours. The WiFi router supports speeds of 343 Mbit/s [7]. The ISP advertises a rate of 25 Mbit/s, although we see a larger rate in our measurements.

We run two video servers on an Amazon EC2 r5a.4xlarge instance; one is responsible for serving Minerva videos, while the other serves videos over Cubic. We simultaneously launch 8 clients (4 Minerva and 4 Cubic) as separate Google Chrome instances for 200 seconds. Figure 4-15 compares the QoEs achieved by Minerva

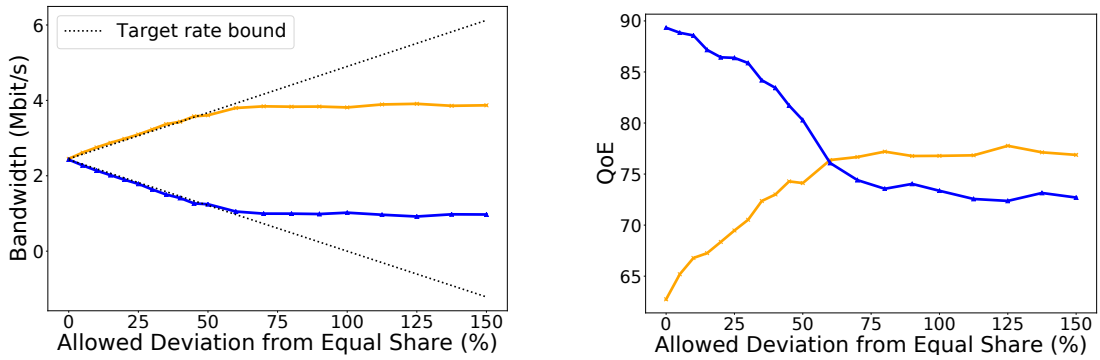


Figure 4-16: QoE and bandwidth for two Minerva videos, as a function of the rate bounding policy they implement.

flows to those achieved by Cubic over 23 independent runs. We observe that the bandwidth of the link fluctuates over time, so results from different runs are not comparable. Averaged over all runs, Minerva stays close to its equal bandwidth share, while improving the minimum QoE from 79 to 85 points; however, the improvement varies substantially between runs, with 3 of 25 runs (12%) not seeing any improvement over Cubic. Minerva also pushes QoEs of all videos closer together: while the range between Cubic video QoEs is 11 VMAF points, Minerva reduces it to 3.75.

4.9.6 Generalizing Max-Min Fairness

Max-min QoE fairness may not be suitable for all video providers, since it may require a large deviation from each client's equal-bandwidth share, particularly when videos have substantially different PQ curves; however, Minerva is not tied to this metric. Here, we consider two ways that Minerva generalizes past max-min fairness.

First, Minerva allows policies that optimize max-min fairness *subject to the constraint* that the bandwidth shares of the videos don't exceed a value set by the provider. In particular, providers can implement policies to limit the amount of bandwidth a video grabs above its equal-share allocation. Of course, restricting the

client limits the space of rate allocations available to Minerva; Figure 4-16 shows the tradeoff as a function of the amount by which the provider chooses to constrain the flows. Minerva videos stay within the target rate bounds, improving QoE fairness as the bounds are relaxed. Ideally, as the bounding policy is relaxed, the QoEs would converge to be equal; however, Minerva's allocation isn't perfectly optimal and still leaves a small gap.

Second, Minerva's weight update rule allows it to optimize for other fairness metrics by choosing the proper utility function for each client. Appendix D demonstrates how Minerva can optimize for proportional fairness, another popular fairness metric.

4.9.7 Multiple Providers using Minerva

We demonstrate here that multiple providers can use Minerva independently on the same bottleneck link, without exchanging information or even being aware of other Minerva providers sharing the link.

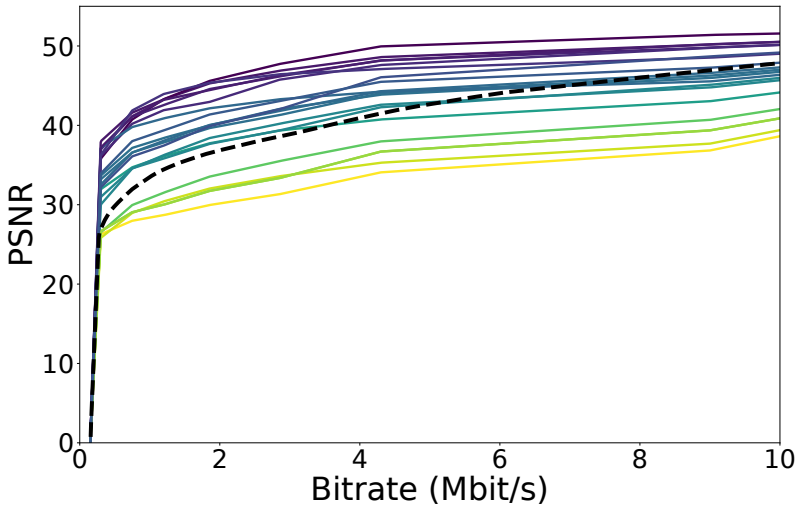


Figure 4-17: The average PSNR scores for the videos in our corpus, along with the normalization function (dotted).

We run two instances of Minerva against each other, emulating two providers using different measures of video quality: VMAF and PSNR [77]. VMAF and PSNR use different scales and cannot directly be compared to each other. The PSNR scores for the videos in our corpus are shown in Figure 4-17. Note that they are noticeably different from the VMAF curves in Figure 4-1: the PSNR curves are on a scale of 53, instead of 100, and are generally flatter for bitrates above 1Mbit/s. This impacts the magnitude of gains Minerva achieves, since moving to a higher bitrate does not result in a large improvement in PSNR value. In particular, the difference between 720p and 1080p is only 2.26 on the PSNR scale, averaged over our entire corpus; by contrast, the difference in quality between the same bitrates measured with VMAF is 7.65. Even when accounting for the difference in scale between the two metrics, the marginal improvement in PSNR between adjacent bitrates is much less than the corresponding improvement in VMAF.

Each provider serves 4 videos and uses a normalization computed over the entire corpus with their respective metric. Figure 4-18 shows that both providers split bandwidth about equally on average, over 20 randomly sampled video combinations and a variety of link rates.

We now consider the QoE fairness of clients using Minerva to those using Cubic, in the presence of two providers. As before, both providers run four videos each, sampled randomly from our corpus. Both providers run Minerva, with one using VMAF as its quality metric, while the other uses PSNR. As a separate baseline, we run the same videos over Cubic with an ABR module that uses either VMAF or PSNR. We run 20 different video combinations over a range of link rates. Figure 4-19 shows that, regardless of which metric is used, both clients see improvements in QoE fairness: the median improvement is 6.1 VMAF points and 2.41 PSNR points for the respective providers. Note that the magnitude of improvements between the two

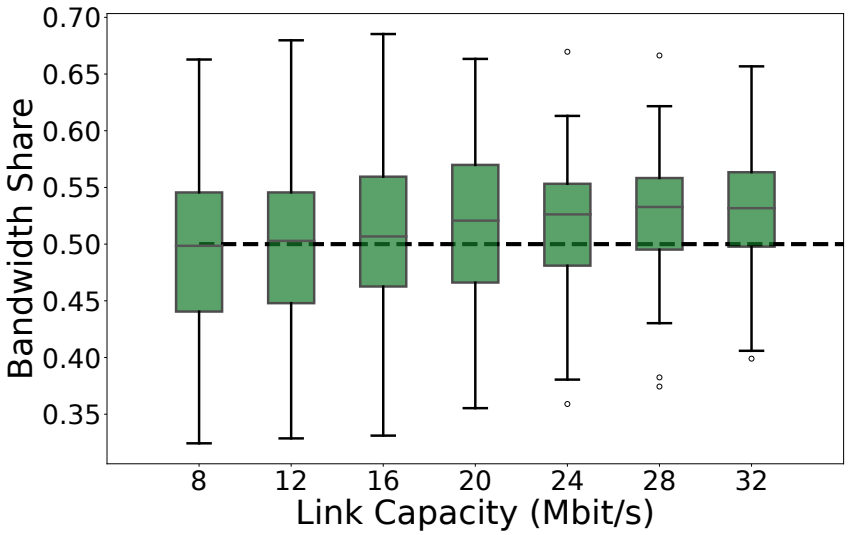
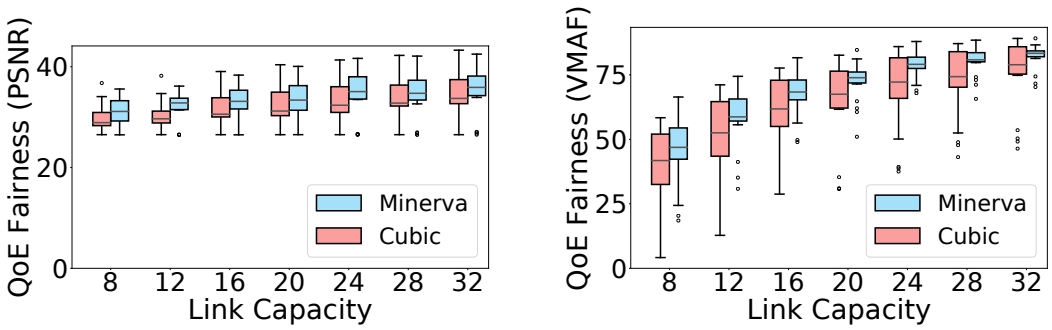


Figure 4-18: Bandwidth share between two providers using different metrics (PSNR and VMAF) over 20 video combinations. On average, the providers achieve close to a 50% bandwidth split.

clients are not comparable, since each metric has a different range and a different overall shape. However, the median PSNR improvement is larger than the 2.26-point PSNR difference between 720p and 1080p, suggesting that the improvements in PSNR are visually significant.

This result has two implications. First, the normalization step is critical to the performance of Minerva. Normalization places all providers' weights on the same scale, without them sharing any information. Without it, providers would have to collaborate to decide on a single QoE metric. Second, providers using Minerva compete fairly with *each other*, not just Cubic. This property allows any number of providers to share the same bottleneck link without worrying about the others' presence.



(a) PSNR

(b) VMAF

Figure 4-19: Providers both running Minerva on the same link achieve improvements in QoE fairness over Cubic regardless of which QoE metric they use.

4.10 Conclusion

Despite the growth of video streaming traffic, there has been relatively little work on deployable solutions to improve QoE fairness between multiple users. We propose Minerva, the first system that achieves QoE fairness in a distributed manner, requiring no changes to underlying network infrastructure. Minerva's formulate-solve-send control flow updates rates for each client independently, such that when they share a bottleneck link, their rates converge to a bandwidth allocation that maximizes QoE fairness, while competing fairly with other traffic on average. We implement Minerva over QUIC and show that, 24% of the time, it can improve the worst viewing experience among a set of videos, by an amount roughly equivalent to a jump from 720p to 1080p. Additionally, in a dynamic environment, Minerva can take effectively pool the buffers of the competing clients to achieve reduction in rebuffering time of up to 47%. Minerva generalizes well to multiple clients, different link speeds, and real residential links, suggesting that it is a deployable solution for video providers seeking to optimize QoE fairness.

Chapter 5

Conclusion and Future Work

This thesis described three different systems that we designed to be instance aware. Although the exact approach to instance optimize each system differs between systems and across domains, we isolated several common attributes that capture the essence of our design paradigm: the ability to expose sufficiently many knobs, a two-phased optimization approach that consists of both system measurement and performance modeling components, and an optimization technique that pairs with the performance model. These components provide a blueprint for creating more instance aware systems that can achieve significantly better performance than existing systems, while also being fast to train and optimize, making them practical to deploy.

However, work on instance awareness is far from complete. We examine some key remaining questions to serve as a guide for future work.

More Complex Models. The systems we implement opt for simple performance models. For example, Flood uses a linear combination of statistics, and Cortex uses a linear equation that can be solved analytically. We choose simple models because they are easy to reason about and interpret, and fast to evaluate. However, this leaves the question: *can we do better by using more complex models?* While we believe that

simple models can capture a substantial share of the performance improvement made possible by instance awareness, more elaborate models might be able to capture the intricate interactions between system components that are ignored by simple, linear ones.

A natural direction for future work is then to answer: *to what extent can more intricate models further improve performance? Is the extra performance boost is worth the added complexity?* It is important to note that the accuracy of the model itself is not the end goal; it is only important insofar as it changes the final parameter values decided on by the optimization. While added complexity may make the performance model more accurate, it may not yield better performance.

Choosing Knobs and Models. Knobs come in all shapes and sizes. Flood knobs were the parameters of the grid; there were sufficiently many to be able to tune performance, but not too many to make optimization infeasible. Cortex's knobs were the outlier classification of each point; although there are 2^D possible combinations, where D is the size of the dataset, we simplify the scope of the optimization by bucketing points, so there are only 2^B possibilities, where B is the number of cells created by the host and target buckets. Minerva's knobs were the rates of each client, which changed constantly over the course of a video. Knowing the knobs in one system yields very little information about the best knobs in another system.

Since the model must be a function of the system's tunable parameters, our choice of model depends on our choice of knobs. However, choosing the best set of knobs for an instance aware system is not an easy task, since it depends on the system's design and architecture. One key question for future work is to ask: *how do we choose which knobs to expose in a given system? And how do we know those are the "best" knobs to achieve the performance goal in question?*

Instance-based Change Detection. All three systems use information about the

instance or environment. However, the instance is subject to change over time. For example, in Flood and Cortex, the query distribution may change after a period of time, or points may be inserted and deleted to alter the underlying dataset. In Minerva, the distribution of videos streamed by a particular provider may change, prompting a recalculation of the normalization function.

A changing instance can be costly. Although recalculating the normalization function for Minerva is fast, finding an entirely new grid layout in Flood (and reindexing the whole dataset as a result) is a time consuming operation that scales with the dataset size. A crucial step in making instance aware systems more practical and widespread is to solve the "re-indexing" problem: *how can we make instance-aware systems adapt smoothly and quickly to changing instances?*

Even more fundamental is knowing *when* to re-index a system. For example, slight changes in query workload may not affect the performance of Flood compared to other indexes. However, at some point, the workload will have changed substantially enough to make Flood considerably worse than either (a) a Flood index trained on the most recent workload distribution and (b) other traditional indexes. Since the database is not simultaneously running multiple indexes to choose the fastest, we must be able to answer: *when has the instance changed sufficiently to merit re-optimization?*

A straightforward answer to this question is to run the optimization procedure periodically, at pre-specified time intervals, and compare the performance of the resulting parameters against the current system. This is the approach that Flood currently takes. However, more sophisticated change detection schemes may be possible. For example, a system could take advantage of its simple performance model to estimate if the performance has deteriorated. If the performance has worsened by some predetermined amount, the system re-optimizes. This is cheaper and potentially faster in identifying instance changes, but relies heavily on the accuracy of the

performance model to trigger re-optimization.

The paradigm proposed in this thesis provides a blueprint for building instance awareness into systems, but it is only the first step to making robust instance-aware systems more widespread. We hope that answers to the above questions will push this field further and make instance-aware systems more accessible and easily deployable in the near future.

Appendix A

Cortex: Expected Scan Overhead

We consider the setting where the value range of a target attribute is divided into t buckets, each with an equal number of points. A query with selectivity s_q, defined as the ratio of the query result size to the size of the table, must scan points in all the buckets it intersects. We aim to find the average scan overhead, i.e. the ratio of the number of points scanned to the number of points matching the query.

We work in CDF space, wherein each point p is mapped to $\frac{i}{D}$, where D is the dataset size and i is p's location in the list of points sorted by the target attribute. In this space, all buckets have equal width $w = \frac{1}{t}$ and queries with selectivity s_q have width s_q.

The minimum number of bucket boundaries a query crosses is $m = \lceil \frac{s_q}{w} \rceil - 1$. Define the spillover to be $s' = s_q - wm$ (Figure A-1).

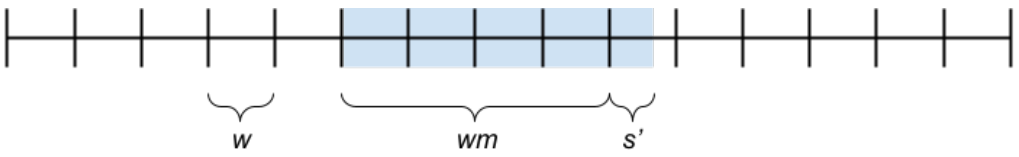


Figure A-1: Illustration of w (bucket width), m, and s'. The query is shown in blue and the target buckets are demarcated in black.

Generally, if the query's start boundary falls within the first $\frac{s'}{w}$ of a bucket, it will intersect $m+2$ buckets; otherwise, it will intersect $m+1$. (We disregard queries with $s'=0$ that fall exactly on the bucket boundary, which have volume 0 in the limit of continuous attributes.) The number of scanned points is then:

$$\frac{s'}{w}(m+2)w + \left(1 - \frac{s'}{w}\right)(m+1)w = s' + w(m+1) = w + s_q$$

The scan overhead is then $1 + \frac{w}{s_q} = 1 + \frac{1}{ts_q}$.

Note that we have ignored queries that spill past the end of the attribute's value range and therefore scan no additional buckets. However, the fraction of these queries is $O(s_q)$, so its contribution is negligible when $s_q \ll 1$.

Appendix B

Minerva: Proof of Convergence

Let $\mathbb{R}_+$ denote the non-negative real numbers.

Definition 1. *A function* $f : \mathbb{R}_+ \to \mathbb{R}_+$ *is* $(\alpha, \beta)-$subquadratic *if, for all* x, y *such that* $x \leq y$*:*

$$\left(\frac{y}{x}\right)^{\alpha} \leq \frac{f(y)}{f(x)} \leq \left(\frac{y}{x}\right)^{2-\beta}$$

Intuitively, the subquadratic condition implies that f is monotonically increasing but does not grow too fast. In particular, over any compact interval, any increasing concave function is $(\alpha, 1)$-subquadratic for some $0 < \alpha \leq 1$.

The following theorem says that, as long as the client's utility function is $(\alpha, \beta)-$subquadratic for *any* $\alpha, \beta > 0$, their rates converge to optimal in a doubly logarithmic number of iterations. This condition is fairly broad: it includes all concave functions as well as some convex ones.

Theorem 1. *Let* $r_{i,t}$ *be the rate of client* i *after* t *iterations of Minerva's decentralized weight update algorithm (Section 4.6.2), let the shared link have constant capacity* c*, and suppose that each client utility function* $U_i(r_i)$ *satisfies the following conditions:*

- $U_i(x) \geq 0$.

- *There exist $\alpha > 0, \beta > 0$ such that $U_i(x)$ is (α, β)-subquadratic on the interval $[0, c]$. We take α, β to be the maximal such values.*

- *There exists an optimal allocation of rates $\{r_i^*\}$, with $r_i^* > 0$, such that the $U_i(r_i^*)$ are equal.*

Then for all iterations t:

$$\sum_i |\log\ r_{i,t} - \log\ r_i^*| < K(1 - \min(\alpha, \beta))^t$$

where K is a constant that depends on the initial rates.

Proof. We prove convergence for two clients, with utility functions U_1 and U_2; the result easily extends to more clients. We assume the link capacity is a constant c and that the rates of the two clients, $r_{1,t}$ and $r_{2,t}$, always sum to c. The optimal rates for the two clients are r_1^* and r_2^*, which satisfy $U_1(r_1^*) = U_2(r_2^*) \equiv u^*$.

Without loss of generality, assume $r_{1,t} < r_1^*$, which implies that $r_{2,t} > r_2^*$. We aim to prove convergence of $r_{1,t} \to r_1^*$ and $r_{2,t} \to r_2^*$. Since we are bound by the constraint that $r_{1,t} + r_{2,t} = c$, it is sufficient to prove that $\frac{r_{2,t}}{r_{1,t}} \to \frac{r_2^*}{r_1^*}$ or, equivalently, $\frac{r_1^*}{r_{1,t}} \cdot \frac{r_{2,t}}{r_2^*} \to 1$. Define:

$$X_{1,t} = \frac{r_1^*}{r_{1,t}} \qquad X_{2,t} = \frac{r_{2,t}}{r_2^*}$$

so our goal is to show $X_{1,t}X_{2,t} \to 1$.

In each iteration of the weight update, the clients compute weights $w_i = \frac{r_i}{U_i(r_i)}$, and Minerva's solve step achieves new rates in proportion to these weights:

$$\frac{r_{2,t+1}}{r_{1,t+1}} = \frac{w_2}{w_1} = \frac{u_1}{u_2}\frac{r_{2,t}}{r_{1,t}} \tag{B.1}$$

Therefore, we have:

$$X_{1,t+1}X_{2,t+1} = \left(\frac{U_1(r_{1,t})}{u^*} \cdot X_{1,t}\right)\left(\frac{u^*}{U_2(r_{2,t})} \cdot X_{2,t}\right)$$

Since the U_i are $(\alpha, \beta)-$subquadratic:

$$\left(\frac{r_{1,t}}{r_1^*}\right)^{2-\beta} \leq \frac{U_1(r_{1,t})}{u^*} \leq \left(\frac{r_{1,t}}{r_1^*}\right)^{\alpha}$$

and likewise for $\frac{u^*}{U_2(r_{2,t})}$. Note that the direction of the inequality is flipped from the definition because $r_{1,t} < r_1^*$. It follows that:

$$(X_{1,t}X_{2,t})^{\beta-1} \leq X_{1,t+1}X_{2,t+1} \leq (X_{1,t}X_{2,t})^{1-\alpha}$$

It is possible that $X_{1,t+1}, X_{2,t+1} < 1$ if $\beta < 1$, which means that $r_{1,t+1} > r_1^*$ and $r_{2,t+1} < r_2^*$. In this case:

$$|\log\ X_{1,t+1}| + |\log\ X_{2,t+1}| \leq (1-\beta)\left(|\log\ X_{1,t}| + |\log\ X_{2,t}|\right)$$

In the other case, where $\beta > 1$ and $X_{1,t+1} > 1$:

$$|\log\ X_{1,t+1}| + |\log\ X_{2,t+1}| \leq (1-\alpha)\left(|\log\ X_{1,t}| + |\log\ X_{2,t}|\right)$$

We then conclude that:

$$|\log\ X_{1,t+1}| + |\log\ X_{2,t+1}| \leq (1-\min(\alpha,\beta))|\ (\log\ X_{1,t}| + |\log\ X_{2,t}|)$$

Iterating from the initial rates gives that:

$$|\log\ X_{1,t}| + |\log\ X_{2,t}| \leq (1 - \min(\alpha, \beta))^t\ (|\log\ X_{1,0}| + |\log\ X_{2,0}|)$$

completing the theorem for 2 clients.

□

Appendix C

Minerva: Convergence with a Value Function

Minerva's value function (Section 4.6.3) depends heavily on the ABR algorithm used to compute it. As a result, it is not possible to always guarantee that it satisfies the convergence conditions outlined in Appendix B. Here, we consider a sample value function, and show that while it is not exactly convex, it can still be approximated as such.

First consider the following value function $V_h(r, b, e)$, which captures the optimal QoE possible for a video over the next h chunks, given a fixed link rate r, buffer b, and current bitrate e:

$$V_h(r, b, e) = \max_{e'} \; Q\left(e, \left[\frac{4e'}{r} - b\right]_+, e'\right) + \gamma V_{h-1}\left(r, \left[b - \frac{4e'}{r}\right]_+ + 4, e'\right)$$

where Q is the QoE of a single chunk given in Equation 4.2, and $0 < \gamma \leq 1$ is a discount factor and we have assumed a chunk duration of 4 seconds. The expression $[\cdot]_+$ is equivalent to $\max(\cdot, 0)$. MPC [94] can be formulated in this way using $\gamma = 1$ and $h = 5$.

In the non-discounted case, $\gamma = 1$, it can be seen that the optimal strategy involves switching between two adjacent bitrates e_i and e_{i+1} such that $e_i \leq r \leq e_{i+1}$. The

client stays at bitrate e_i until it has enough buffer to switch to e_{i+1} for the remainder of the horizon. This incurs a smoothness penalty S only once over all h chunks. The value function is then:

$$V_h(r) = \frac{aP(e_i) + (h-a)P(e_{i+1})}{h} + \frac{S}{h}$$

for some integer $a \leq h$. As $h \to \infty$, $V_h(r)$ approaches a linear interpolation of P, the perceptual quality. V_h thus approaches a concave function, but for any finite h is not concave. Figure 4-4 shows an example undiscounted value functions over a horizon of 5 chunks at different starting buffer levels. The step-like nature of the curves in the figure prevent them from meeting the convergence conditions in Appendix B. However, there are two ways to handle this issue.

First, we can fit the value function with a function known to be meet convergence conditions. For example, the value functions in Figure 4-4 can be approximated by exponential functions of the form: $f(r) = A - Be^{-Cr}$, for fitting parameters A, B, C. Though approximating the value function in this manner guarantees convergence, the convergence point may not be the true max-min QoE fair allocation. Alternatively, since the finite-horizon value function largely resembles a concave function, except for local step-like behavior, simply using the function as is may suffice. We find that this is the case for our video corpus: we use the actual function, instead of a fit, in Minerva's implementation, and find that it yields sufficiently strong results.

Appendix D

Minerva: Optimizing for Proportional Fairness

Minerva also allows optimizing for QoE fairness with different functional forms from Max-min fairness, such as $\alpha-$fairness. Here, we consider $\alpha-$ fairness where $\alpha = 1$, known as proportional fairness.

The only change to Minerva comes in the weight-update step. The weight update step only requires each client to have some function of rate $Z_i(r)$; Minerva's decentralized algorithm modifies rates to achieve equality between all the $Z_i(r_i)$. When $Z_i = U_i$, the client's utility function, Minerva achieves max-min fairness. Using a different Z_i would optimize for a different notion of fairness. To illustrate, consider proportional fairness, which maximizes $\sum_i \log(U_i(r_i))$. The optimal rate allocation satisfies:

$$\frac{U_i'(r_i)}{U_i(r_i)} = \frac{U_j'(r_j)}{U_j(r_j)} \qquad \forall i, j$$

Setting $Z_i(r_i)$ to be some function of ${}^{U_i'(r_i)}/_{U_i(r_i)}$ pushes Minerva towards proportional fairness. However, Z_i must still be increasing and concave, so it must be chosen carefully based on the shape of the PQ curves. For example, $Z_i(r_i) = C - {}^{U_i'(r_i)}/_{U_i(r_i)}$

has the required properties for the utility curves in Figure 4-1 and can be substituted for U_i in the weight update step. In particular, the normalization function must be computed using Z_i.

For proportional fairness, $U_i'(r_i)/U_i(r_i)$ should be equal for all i. If $\overline{P}(r)$ is the PQ curve averaged over chunks, then We compute $U_i'(r_i)$ by taking the numerical derivative of the PQ curve averaged over all chunks. On each weight update, we compute the expected QoE q, the *representative rate* $r_r = \overline{P}^{-1}(q)$, and finally $Z_i = C - U_i'(r_r)/q$. The choice of C is arbitrary and does not affect convergence.

Bibliography

[1] Empirical Traffic Generator. https://github.com/datacenter/empirical-traffic-gen.

[2] Akamai. dash.js. https://github.com/Dash-Industry-Forum/dash.js/, 2016.

[3] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58. ACM, 2018.

[4] Mark Allman, Vern Paxson, Wright Stevens, et al. Tcp congestion control, 2009. IETF RFC 5681.

[5] 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-interleave 2016.

[6] Andy Kimball. How We Built a Cost-Based SQL Optimizer. https://www.cockroachlabs.com/blog/building-cost-based-sql-optimizer, 2018.

[7] Arris. TG862G/CT Xfinity Residential Gateway & Router. https://arris.secure.force.com/consumers/ConsumerProductDetail?p=a0ha000000GOZ3yAAH.

[8] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, 1999.

[9] Abdelhak Bentalab, Ali C. Bergen, Saad Harous, and Roger Zimmermann. Want to Play DASH? A Game Theoretic Approach for Adaptive Streaming over HTTP.

In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys)*. ACM, 2018.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5):50:20–50:53, October 2016.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[12] 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.

[13] Junyang Chen, Mostafa Ammar, Marwan Fayed, and Rodrigo Fonseca. Client-driven network-level qoe fairness for encrypted'dash-s'. In *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 55–60. ACM, 2016.

[14] Cisco. Cisco visual networking index: Forecast and methodology, 2017-2022. 2018.

[15] Jon Crowcroft and Philippe Oechslin. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, July 1998.

[16] 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`.

[17] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. Alex: An updatable adaptive learned index, 2019.

[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. 2021.

[19] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015.

[20] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *NSDI*, 2018.

[21] Andrew Lamb et al. The Vertica Analytic Database: C-Store 7 Years Later. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 2012.

[22] Mike Stonebraker et al. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st VLDB Conference*. VLDB Endowment, 2005.

[23] Exasol. The World's Fastest In-Memory Analytic Database. `https://www.exasol.com/en/community/resources/resource/worlds-fastest-analytic-database/`.

[24] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *NSDI*. USENIX, 2017.

[25] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30:170–231, 1998.

[26] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. A-tree: A bounded approximate index structure. *CoRR*, abs/1801.10207, 2018.

[27] Google. Google chrome web browser. `https://www.google.com/chrome/`.

[28] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, January 1997.

[29] Greg White. Active queue management in DOCSIS 3.1 networks. *IEEE Communications Magazine*, 53(3):126 – 132, March 2015.

[30] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.

[31] Marios Hadjieleftheriou. libspatialindex. https://libspatialindex.org/, 2014.

[32] Klaus Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, 25(4):569–592, December 1985.

[33] Mario Hock, Roland Bless, and MArtina Zitterbart. Experimental evaluation of BBR congestion control. In *25th International Conference on Network Protocols*. IEEE, 2017.

[34] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[35] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Twin grid files: Space optimizing access schemes. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 183–190, New York, NY, USA, 1988. ACM.

[36] IBM. The Spatial Index. https://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.spatial.doc/ids_spat_024.htm.

[37] IBM. Configuration parameters that affect query optimization. https://www.ibm.com/docs/en/db2-warehouse?topic=SSCJDQ/com.ibm.swg.im.dashdb.admin.config.doc/doc/c0005035.html, 2019.

[38] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. Conference on Innovative Data Systems Research (CIDR), 2007.

[39] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*. ACM, 2009.

[40] Stratos Idreos, Konstantinos 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 *ACM SIGMOD International Conference on Management of Data*, 2018.

[41] IETF. Cubic for fast long-distance networks. 2006.

[42] Intel Corporation. Scaling Data Capacity for SAP HANA with Fujitsu PRIMERGY/PRIMEQUEST Servers. Technical report, 2014.

[43] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *CoNEXT*, 2012.

[44] Cheng Jin, David X Wei, and Steven H Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking*, 14(6):1246–1259, 2006.

[45] Frank P. Kelly, Aman K. Maulloo, and David K. H. Tan. Rate control for communication networks: Shadow price, proportional fairness, and stability. *The Journal of the Operational Research Society*, 49(3):237–252, 1998.

[46] 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.

[47] 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.

[48] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, 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*, 2019.

[49] 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)*. ACM, 2018.

[50] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*, 2017.

[51] Iosif Lazaridis and Sharad Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 401–412, New York, NY, USA, 2001. ACM.

[52] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. Toward A Practical Perceptual Video Quality Metric. https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652, 2016.

[53] Zhi Li, Xiaoqing Zhu, Josh Gahm, Rong Pan, Hao Hu, Ali C. Begen, and Dave Oran. Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale. 2014.

[54] 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.

[55] Ahmed Mansy, Marwan Fayed, and Mostafa Ammar. Network-layer fairness for adaptive video streams. In *IFIP Networking Conference (IFIP Networking), 2015*, pages 1–9. IEEE, 2015.

[56] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[57] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, 2021.

[58] Jim Martin, Yunhui Fu, Nicholas Wourms, and Terry Shaw. Characterizing Netflix bandwidth consumption. In *10th Consumer Communications and Networking Conerence*. IEEE, 2013.

[59] Microsoft SQL Server. Spatial indexes overview. https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-indexes-overview?view=sql-server-2017, 2016.

[60] MonetDB. monetdb. https://www.monetdb.org/, 2018.

[61] Anush Krishna Moorthy, Lark Kwon Choi, Alan Conrad Bovik, and Gustavo de Veciana. Video quality assessment on mobile devices: Subjective, behavioral, and objective studies. *IEEE Journal of Selected Topics in Signal Processing*, 6(6):652 – 671, 2012.

[62] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing (pdf). Technical report, IBM, 1966.

[63] Multichannel News. Comcast Goes Wide With DOCSIS 3.1 Gigabit Gateway. https://www.multichannel.com/news/comcast-goes-wide-docsis-31-gigabit-gateway-416930, 2017.

[64] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM*, 2016.

[65] Hyunwoo Nam, Bong Ho Kim, Doru Calin, and Henning Schulzrinne. A mobile video traffic analysis: Badly designed video clients can waste network bandwidth. In *Globecom Workshop*, 2013.

[66] 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.

[67] Inc. Netflix. Open connect. https://openconnect.netflix.com/en/, 2019.

[68] Ravi Netravali, Anirudh Sivaraman, Keith Winstein, Somak Das, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*, 2015.

[69] 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.

[70] Beng Chin Ooi, Ron Sacks-davis, and Jiawei Han. Indexing in spatial databases, 2019.

[71] OpenStreetMap contributors. US Northeast dump obtained from https://download.geofabrik.de/. https://www.openstreetmap.org, 2019.

[72] Oracle Database Data Warehousing Guide. Attribute Clustering. https://docs.oracle.com/database/121/DWHSG/attcluster.htm, 2017.

[73] Oracle, Inc. Oracle Database In-Memory. https://www.oracle.com/database/technologies/in-memory.html.

[74] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili S. Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *14th International Conference on High Performance Switching and Routing*, 2013.

[75] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. QuickSel: Quick Selectivity Learning with Mixture Models. In *SIGMOD*. ACM, 2019.

[76] D.C. Pew Research Center, Washington. A third of Americans live in a household with three or more smartphones. `http://www.pewresearch.org/fact-tank/2017/05/25/a-third-of-americans-live-in-a-household-with-three-or-more-smartphones/`, May 2017.

[77] David Salomon. *Data Compression: The Complete Reference*. Springer, 2007.

[78] Sandvine Intelligent Broadband Networks. Global Internet Phenomena: Latin America and North America. `https://www.sandvine.com/hubfs/downloads/archive/2016-global-internet-phenomena-report-latin-america-and-north-america.pdf`, 2016.

[79] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *Proc. VLDB Endow.*, 7(2):97–108, October 2013.

[80] Scipy.org. scipy.optimize.basinhopping. `https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.basinhopping.html`.

[81] 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.

[82] Hari Singh and Seema Bawa. A survey of traditional and mapreducebased spatial query processing approaches. *SIGMOD Rec.*, 46(2):18–29, September 2017.

[83] Iraj Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[84] Spark SQL. Cost-Based Optimization (CBO) of Logical Query Plan. `https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-cost-based-optimization.html`, 2017.

[85] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. BOLA: near-optimal bitrate adaptation for online videos. volume abs/1601.06748, 2016.

[86] Jim Summers, Tim Brechy, Derek Eager, and Alex Gutarin. Characterizing the Workload of a Netflix Streaming Video Server. In *International Symposium on Workload Characterization*. IEEE, 2016.

[87] Markku Tamminen. The extendible cell method for closest point problems. *BIT*, 22:27–41, 01 1982.

[88] TPC. TPC-H. `http://www.tpc.org/tpch/`, 2019.

[89] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 2011.

[90] 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.

[91] Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 3(4):600–612, 2004.

[92] 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 Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 1223–1240, 2019.

[93] Xiaoqi Yin, Mihovil Bartulović, Vyas Sekar, and Bruno Sinopoli. On the efficiency and fairness of multiplayer http-based adaptive video streaming. In *American Control Conference (ACC), 2017*, pages 4236–4241. IEEE, 2017.

[94] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 325–338. ACM, 2015.

[95] 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.

[96] Chao Zhou, Chia-Wen Lin, Xinggong Zhang, and Zongming Guo. Tfdash: A fairness, stability, and efficiency aware rate control approach for multiple clients over dash. *IEEE Transactions on Circuits and Systems for Video Technology*, 2017.