Abstract of "Accelerating Interactive Data Exploration" by Alexander Galakatos, Ph.D., Brown University, May 2019.

The widespread popularity of visual data exploration tools has empowered domain experts in a broad range of fields to make data-driven decisions. However, a key requirement of these tools is the ability to provide query results at "human speed," even over large datasets. To meet these strict requirements, we propose a complete redesign of the interactive data exploration stack. Instead of simply replacing or extending existing systems, however, we present an *Interactive Data Exploration Accelerator* (IDEA) that connects to existing data management infrastructures in order to speed up query processing for visual data exploration tools.

In this redesigned interactive data exploration stack, where an IDEA sits between a visual data exploration tool and a backend data source, several interesting opportunities emerge to improve interactivity for end users. First, we introduce a novel approximate query processing formulation that better models the conversational interaction paradigm promoted by visual data exploration tools. Since the time dimension is often a critical component of many datasets, we then present a detailed survey of existing backend data sources specifically designed for time-dependent data in the context of interactive data exploration. Finally, based on the results from our study, we propose a new approximate index structure for an interactive data exploration stack that leverages the trends that exist in the underlying data to improve interactivity while significantly reducing the storage footprint.

Accelerating Interactive Data Exploration

by
Alexander Galakatos
B.S., Lehigh University 2012
Sc. M., Brown University, 2015

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2019

This dissertation by Alexander Galakatos is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ________________ ____

Tim Kraska, Director

Recommended to the Graduate Council

Date ________________ ____

Carsten Binnig, Reader
(Technical University of Darmstadt)

Date ________________ ____

Rodrigo Fonseca, Reader
(Brown University)

Approved by the Graduate Council

Date ________________ ____

Andrew G. Campbell
Dean of the Graduate School

To my family.

Acknowledgements

First and foremost, I would like to thank my advisor, Tim Kraska, for his endless support during the entire process. A few months after I started my Master's degree at Brown, Tim convinced me to continue my research and pursue a Ph.D.. Doing so was one of the best decisions that I have ever made. Tim always pushed me towards interesting and important research questions that solved real-world problems.

This thesis would also not have been possible without Andrew Crotty. Over the years we have grown together to become better researchers and great friends. Together we have spent countless hours writing code, revising introductions, and debating everything from low-level caching effects and sorting algorithms to the peer review process and road traffic patterns. I attribute much of my success to him.

I would also like to thank my committee members, Carsten Binnig and Rodrigo Fonseca as well as Ugur Cetintemel and Stan Zdonik, who have all offered invaluable guidance and advice. Emanuel Zgraggen also helped shape a large piece of this work and I'm grateful for our collaboration.

Additionally, I would like to thank the Brown Database Group and the friends that I have made in the department, including Kayhan, John, Erfan, Ani, Phillip, Emanuel, Eric, Yeounoh, and Sam.

I would also like to thank Henry Korth who sparked my interest in database research during my time as an undergraduate at Lehigh University.

Finally, and most importantly, I would like to thank my family, Elizabeth, and all of my friends for encouraging and supporting me throughout my entire life.

Contents

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The growing prevalence of big data across all industries and sciences is causing a profound shift in the nature and scope of analytics. Now, rather than relying on mere intuition or historical precedent, data scientists can help domain experts make data-driven decisions based on insights derived from data. Moreover, with the growing amount of disparate and diverse datasets available, data scientists are now able to integrate data that was previously not considered during the decision making process. This has resulted in an entirely new breed of frameworks that help users explore and analyze data, including distributed analytics frameworks like Spark [150], Flink [77], and Tupleware [32]. The domain experts that are interested in such analyses, however, usually lack the sophistication that more complex data analysis tools demand, whether it be the need to craft complex SQL queries or use nuanced technical languages.

On the other hand, visual analytics enables domain experts with little technical expertise to gain insights out of a dataset through visualizations, which are one of the most important tools for exploring, understanding, and conveying facts about data. This process, referred to as *Interactive Data Exploration* (IDE), is an interactive and iterative process where users need to switch frequently among a range of distinct but interrelated tasks. For example, users of such tools often create simple visualizations that show the distribution of a attribute, drill down into individual subpopulations, run statistical significance tests, and even build complex machine learning models.

However, as dataset sizes continue to grow and analysis becomes more complex, a wide variety of data management problems begin to surface. Unfortunately, most existing components of an IDE stack are created in isolation which leads a fragmented stack that cannot easily support new and large datasets. For example, a company's existing data warehouse is not designed to serve as the backend for a visual data exploration tool. Furthermore, existing visual data exploration tools are not built to work interactively over both large and small datasets and across data stored in variety of different types of systems. Therefore, since existing tools are unable to effectively handle this increasingly popular use case, we require a fundamental redesign of our approach to designing IDE frameworks. Such an undertaking requires us to consider the entire IDE stack as a whole, spanning from visual interfaces all the way to data warehouses.

To better understand the requirements for interactive data exploration tools, we first discuss some of the key challenges associated with designing an IDE stack. Then, we discuss common characteristics of interactive data exploration workloads, including the types of data that users frequently examine as well as the common actions that users perform. Next, present a running example that we trace through each of the three layers of our stack, explaining some of the key research insights and contributions we discovered along the way. Finally, we summarize the key contributions of this work.

1.1 Challenges of Interactive Data Exploration

IDE has a very unique set of requirements, many of which are pushing the boundaries of what is feasible today. In the following, we outline the key challenges that need to be addressed in the development of an IDE stack. The work presented in this thesis aims to directly address each of these challenges, including a new IDE stack that enables immediate exploration of new datasets (Chapter 2) as well as new approximate query processing (Chapter 3) and indexing (Chapter 5) techniques to improve interactivity.

Conversational Querying

Users typically begin exploring a new dataset by first creating a simple visualization, perhaps of a single attribute, and then proceed to build up a complex visual query through several intermediate steps. In contrast to one-shot DBMS sessions, data exploration is an iterative, session-driven process where a user repeatedly modifies a workflow after examining the results until finally arriving at some desired insight. Visual data exploration tools have encouraged this more conversational interaction paradigm, whereby users incrementally compose and iteratively refine queries throughout the data exploration process.

Connect and Explore

Ideally, the user should be able to connect to a dataset and immediately start exploring it without the need for expensive data preparation or long data loading times (e.g., for index construction). For example, many existing visual data exploration tools require a full copy of the data and suffer from long loading times, which contradicts our goal of being able to start exploring a new dataset immediately without expensive data preparation (e.g., indexing, compression).

Interactive Performance

By far, the most important challenge in supporting interactive data exploration is to display a result to the user interactively. As [93] showed, even small delays of more than 500ms can significantly impact the data exploration process and the number of insights a user makes. Therefore, an IDE stack needs to maintain certain response time guarantees in order to provide a fluid user experience.

1.2 Interactive Data Exploration Workloads

Due to the exploratory nature of interactive data exploration, it it not surprising the workflows that users typically create vary dramatically. For example, an analyst at a retailer may drill down into different subpopulations to learn about spending habits (e.g., billionaires in the United States, residents of Massachusetts, male teenagers) while a medical researcher might examine the heart rate of a critically ill patients over time to determine if a patient has a rare heart condition. This wide variety stems from the fact that users of visual data exploration tools are often in search of new insights. Therefore, they either examine new datasets that may be relevant or ask new questions from existing datasets that have already been examined. However, even though benchmarks for interactive data exploration have been proposed [41], there is no single use case that represents interactive data exploration.

In the following, we first discuss the different types of interactive data exploration workloads that we have observed after examining various real-world use cases. Based on our analysis, we categorize interactive data exploration into two distinct classes (1) nominal data exploration, and (2) time-dependent data exploration. Although these are the most common patterns that we noticed after studying interactive data exploration workloads, other classes certainly exist. For example, users may wish to examine spatial data (e.g., map data), explore differences in DNA samples between patients, and analyze image data to find overlapping regions. However, such applications are usually highly specialized and ad-hoc, resulting in a broad range of task-specific tools.

Based on these observations, in Chapter 3, we present techniques that specifically target nominal data, while Chapter 4 and Chapter 5 focus on interactive data exploration of time-dependent data. After discussing the different classes of data exploration, we then present results from a user study we performed that aims to explore how users interact with visual data exploration tools. Based on these results, we present various interesting optimizations in Chapter 3 that leverage user behavior to improve the interactivity of visual data exploration tools.

1.2.1 Data Types

After speaking with real world users and examining the use cases and datasets that are commonly used in interactive data exploration workloads, we present two common yet distinct classes of interactive data exploration. Although similar, each of these two classes requires substantially different techniques to optimize in the context of interactive data exploration. In the following we discuss the unique characteristics and common operations for (1) nominal data, and (2) time-dependent data.

Nominal Data

Common in survey and marketing use cases, nominal data refers to an unordered dataset that does not naturally incorporate the dimension of time. For example, the Census dataset [89] contains attributes such as age, gender, marital status and income level for individuals in the United States. Similarly, the Young People Survey [75] contains survey responses that include the music, movie, health, and spending habits of young individuals in the United Kingdom. Finally, the Wine [89] dataset contains the chemical properties (e.g., pH level, amount of sulfuric acid) along with a quality rating for various wines.

More specifically, a single row in a nominal dataset represents a single entity (e.g., person, customer, wine variety) and each entity only appears once in the entire dataset since entities do not incorporate the time dimension. Moreover, in most datasets, the number of entities is generally relatively small and limited by the physical world. For example, there are only approximately 300 million people that live in the United States [22] and the there are only around $10,000$ different varieties of grapes used to produce wine [107]. Because the number of entities is generally relatively small, the size of nominal datasets is generally smaller than time-dependent datasets. Note that this does not hold true for certain nominal datasets (e.g., scientific datasets that record data about subatomic particles, highly dimensional data).

Since nominal data contains only a single entry for each observed entity, interactive data exploration sessions over nominal data generally aim to uncover similarities or differences between groups of entities. Common interactive data exploration patterns over nominal data often drill down and compare individual subpopulations (e.g., comparing the salary distribution of males and females, determining which country produces the highest quality of wines), allowing the user to discover new insights. Additionally, users often issue simple aggregate queries (e.g., number of residents per state, total amount of wine produced in a single country).

To better support interactive data exploration workloads over nominal data, we present a new approximate query processing formulation in Chapter 3 that better models the interaction paradigm that visual data exploration tools promote. More specifically, our new approximate query processing formulation maximizes result reuse between subsequent queries in order to improve the user experience. Additionally, we present a novel low-overhead index structure in Chapter 5 that is designed to specifically improve the interactivity of queries over rare subpopulations.

Time-dependent Data

Through our discussions with various users and practitioners, time-dependant data created as part of automatic processes emerged as a very important aspect of data exploration. For example, the Weblogs dataset [98] records every hit to the webserver of the Brown University computer science department over the past 14 years. Additionally, the Performance Monitor dataset [98] contains records numerous metrics, including CPU utilization, amount of free memory, and network bandwidth for hundreds of machines in the computer science department at Brown University. Finally, the IoT dataset [98] monitors building activity (e.g., door opened, motion detected) throughout the computer science building at Brown University.

Unlike nominal data, time-dependent data generally tracks single entities over time and often contains subtle yet important trends. Moreover, time-dependant data often contains many entries for a single physical entity since entities are usually tracked over time. For example, even though a building may only have 100 sensors, a single entry exists in the dataset every time a physical action occurs (e.g., temperature changes, a door is opened), resulting in datasets that can often be much larger than nominal datasets.

Throughout our analysis, we also found that queries over time-dependant also differ compared to queries over nominal data. In addition to drilling down into individual subpopulations (e.g., sensors on the third floor, hits to a certain resource URL), the time dimension is often key to the business question at hand. More specifically, we noticed that time series data exploration almost universally required "constraints and group-bys at multiple temporal hierarchies" [99] (e.g., timestamp extraction/manipulation, aggregation grouped by derived time values), whereas traditional OLAP benchmarks merely use the time dimension as just another filter condition.

While existing benchmarks address either streaming aspects (e.g., Linear Road [8], StreamBench [95], RIoTBench [130]) or time series data mining (e.g., CATS [87]), benchmarks for analytical queries over time series data have received almost no attention, despite the importance of these workloads in the data-driven decision making process. Therefore, as a first step, in Chapter 4, we present MgBench, a new analytical benchmark specifically for machine-generated time series data derived from interviews with real-world practitioners. MgBench is the first benchmark for machine-generated data that uses multiple large real-world datasets and realistic queries derived from in-depth interviews with actual stakeholders.

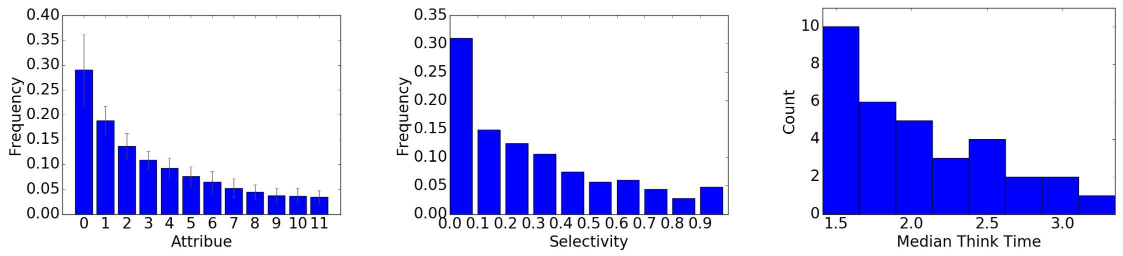
After deriving several insights that shape the future of system design for time series data, in Chapter 5, we propose A-Tree, a novel approximate index structure. A-Tree is designed to leverage the underlying trends that exist in real-world time series data in order to reduce the storage footprint of an index by orders of magnitude.

1.2.2 User Behavior

In addition to examining common data types in the context of interactive data exploration, we also observed actions that users frequently perform through an extensive user study that we performed. The goal of this study was to help us understand how people use visual data exploration tools in order guide optimization effort and improve the interactivity of visual data exploration tools.

In our user study, each of the 33 participants were first given a brief overview of Vizdom [33], a visual data exploration tool that allows users to build up complex data exploration pipelines. Then, they were given 15 minutes to explore each dataset and find non-obvious relationships. All user actions (e.g., user created a visualization plotting age vs. weight, user linked two visualizations and performed a selection) were logged and all insights were derived from directly analyzing the logs.

In the following, we discuss the three key insights (summarized in Figure 1.1) that we gained from our user study: (1) incremental query construction provides many opportunities for result reuse, (2) rare subpopulations are particularly interesting to users, and (3) users spend several seconds thinking about query results before issuing a follow up query. These results form the basis of many of the optimizations and techniques that we present in Chapter 3.



(a) Attributes **(b) Selections** **(c) Think Time**

Figure 1.1: User Study Results

Result Reuse:

Our first insight from the user study was that an interface for interactive data exploration naturally provides opportunities for reuse. Figure 1.1a shows the average frequency (i.e., the percentage of drag actions to the canvas) for different attributes across all three datasets for all users in the user study. For example, on average, the most frequently queried attribute accounted for almost 30% of all drag actions, and 50% of all queries involved three of the most frequently examined attributes. Given this result, it is clear that interactive data exploration sessions provide myriad opportunities for past result reuse and this result motivates our reuse techniques for approximate query processing presented in Chapter 3.

Rare Items:

Although rare items are infrequent, our user study revealed that they are often interesting to users. Intuitively, this makes sense, since rare subpopulations often reveal relationships that are not obvious. Figure 1.1b shows the frequency of selectivities for all selection operations across all three datasets that the user study participants analyzed. For example, more than 30% of all selection operations were on subpopulations which have a selectivity of less than 10%. This insight motivates the presented partial index structure presented in Chapter 3 that stores only rare subpopulations in order to save space while also efficiently supporting queries over rare subpopulations.

Think Time:

Finally, our user study led us to the observation that users often take a substantial amount of time to think about the current visualization and determine what action to take next. Figure 1.1c shows the median think time for each user. As shown, users often spend between one to three seconds thinking about what to explore next. However, we found that the absolute think times vary from a few hundred milliseconds to over 80 seconds. Therefore, the system can often leverage this time to begin indexing the rare subpopulations in a dataset to reduce the latency of future queries. This observation forms the basis of our tail indexing techniques described in Chapter 3.

1.3 A Running Example

As an illustrative example of the IDE process, consider the sample exploration session shown in Figure 1.2, which is drawn from a past user study [34]. In the exploration session, the user is analyzing data from the 1994 US census [89] using Vizdom [33] in order to understand which attributes affect an individual's annual salary, starting with a high-level query and then gradually refining the analysis to focus on specific subpopulations of interest.

First, the user examines the *sex* attribute (Step A) to view the distribution of 'Male' and 'Female' values. The user then links *sex* to *salary* and filters the *salary* distribution to only 'Female' (Step B). A second *salary* bar chart with a negated link (dotted line) compares the salaries for 'Male' and 'Female' subpopulations (Step C). By linking *education* to each of the *salary* visualizations and selecting only individuals with a 'PhD', the user can compare the respective salaries of highly educated males and females (Step D). Afterwards, the user can go on to build a classifier to predict the income level of a new individual or perform a k-means clustering analysis to identify similar groups within the selected subpopulations (not shown).

We use this example as a concrete reference point for our discussion of the different components of our IDE stack throughout the remainder of this work.

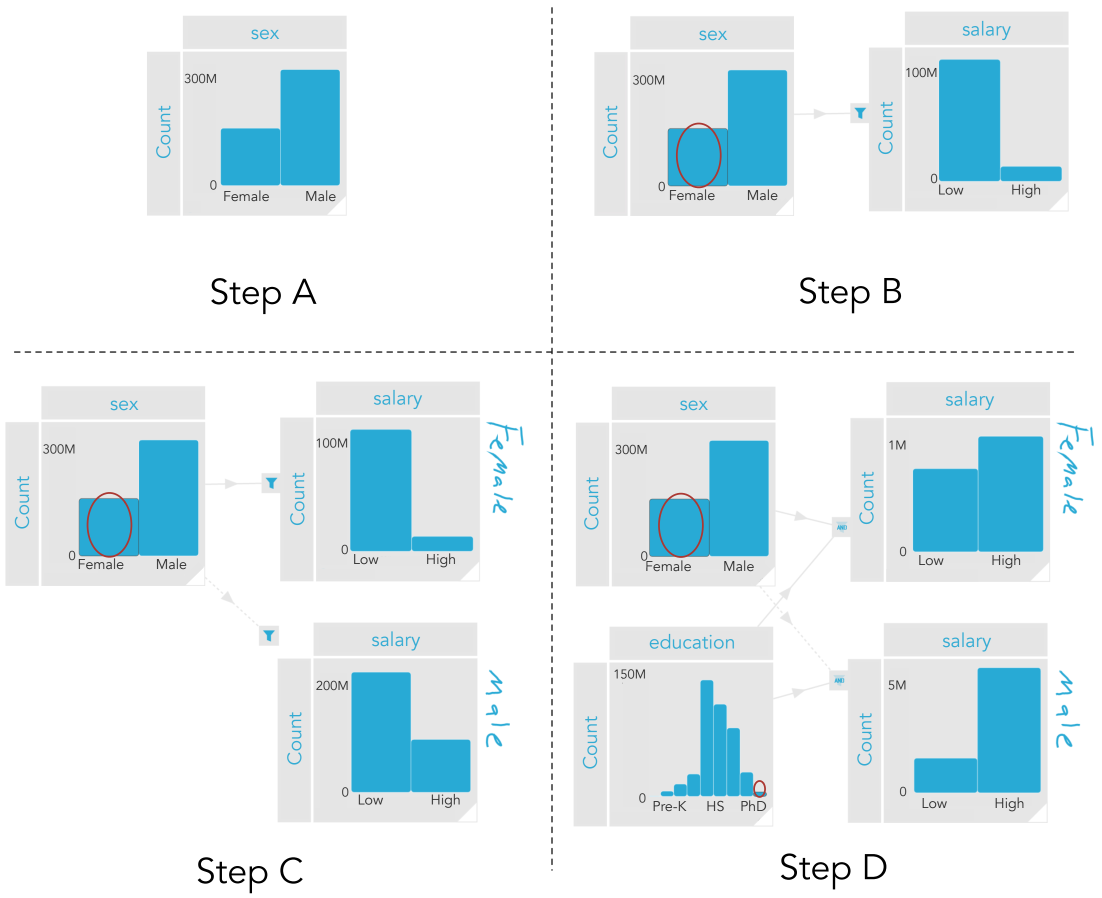


Figure 1.2: An example exploration session where the user is trying to understand which attributes affect an individual's *salary*.

1.4 Contributions and Summary

The goal of this work is to develop new techniques and systems that improve the interactivity of visual data exploration tools for common data exploration tasks.

Therefore, as part of an interactive data exploration stack, we propose an *Interactive Data Exploration Accelerator* (IDEA) in Chapter 2, a middleware that sits between a visual data exploration tool and a backend data source (e.g., legacy DBMS, flat file, analytics framework). As we demonstrate, IDEAs, along with the accompanying techniques, allow us to accelerate interactive data exploration workloads while retaining the complex data infrastructures that existing institutions have deployed. Chapter 2 provides a high-level description of each component in our redesigned interactive data exploration stack.

To better support the conversational paradigm that visual data exploration tools promote and to enable exploration for larger datasets, we present a new approximate query processing formulation specifically for IDEAs in Chapter 3. More specifically, our new approximate query processing formulation is designed to maximize result reuse and improve interactivity for nominal data types in the context of visual data exploration tools. Chapter 3 provides an in-depth explanation of our techniques and outlines the new opportunities that our formulation enables.

Then, given that fact that time is often a critical component of many interesting datasets, Chapter 4 presents an in-depth study of backend systems for use in our IDE stack specifically in the context of time series data. The results of our study provide valuable guidance to future system designers and help end users of an interactive data exploration stack navigate the vast tradeoffs that exist in time series data management.

Finally, based on the results from our findings in our study of existing time series data management solutions, we present A-Tree, a new type of index structure. Our proposed index structure leverages properties of the underlying data distribution to reduce the storage footprint of an index, further improving the interactivity of visual data exploration tools. Chapter 5 discusses, in detail, the design of A-Tree and quantifies the benefits of our new indexing techniques.

Chapter 2

Interactive Data Exploration Stack

To build an IDE stack that successfully allows users to visually explore large datasets in an efficient way, we needed to address the core challenges of interactive data exploration at all levels of the stack from the visual interface to the backend data management system.

Our interactive data exploration stack, shown in Figure 2.1, consists of three basic elements: (1) a visual frontend (Vizdom), (2) a middleware layer (IDEA), and (3) a backend data source. The Vizdom frontend is a visual data exploration environment specifically designed for pen-and-touch interfaces, such as the recently announced Microsoft Surface Hub. In between the visual interface and the backend sits an *Interactive Data Exploration Accelerator* (IDEA), which acts as a middleware that enables users to begin exploring a new dataset (e.g., data warehouse, text file, analytics framework) without any expensive preprocessing. When a user starts a new exploration session, an IDEA begins pulling tuples from a backend data source and populates an in-memory cache. An IDEA then evaluates queries that are issued from the frontend against the data in the IDEA's in-memory cache, using approximate query processing techniques in order to provide query results that refine over time. The approximate results are then sent to the visual frontend in a progressive manner, ensuing a fluid user experience.

In the following, we present a high-level overview of the principal components of our interactive data exploration stack, including the Vizdom frontend, IDEA middleware, and various possible backend data sources.

2.1 Vizdom

As previously mentioned, Vizdom is a visual data exploration tool developed in collaboration with the Graphics, Visualization, & Interaction Group at Brown University. Since properties about the user interface are crucial for guiding the design of core components (e.g., the approximation engine), we now provide a brief overview of Vizdom and discuss its core features.

Vizdom is a interactive visual analytics tool entirely based on progressive computation [152]. It

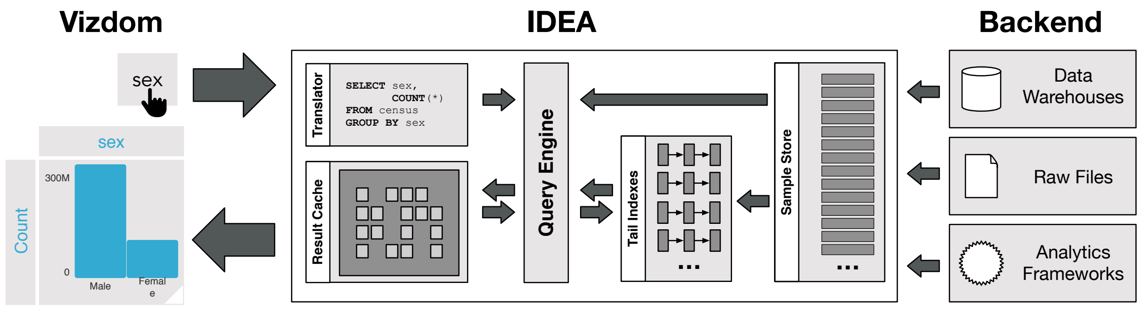


Figure 2.1: IDEA Architecture

supports a wide range of reoccurring visual analysis tasks, including coordinating multiple visualizations to create custom exploratory interfaces and derive models through machine learning operators. Vizdom features a pen-and-touch-based user interface that is designed with a "touch-first" mindset in order to promote a fluid interaction style.

Figure 2.2 shows the primary VIZDOM interface, which is a blank canvas onto which users can drag either individual attributes or predefined operators. Pre-loaded datasets are listed along the bottom, while the attributes for the selected dataset and available operators are displayed along the left-hand side. Due to Vizdom's unbounded 2D canvas, organizing visualizations or operators is as simple as arranging them on the screen through direct touch manipulation.

2.1.1 Visualizations

Visualizations can be created with simple drag and drop operations, and directly manipulated by users with actions such as tapping on an axis which cycles through different aggregation functions (e.g., count, average, etc.). Bins within visualizations can be selected through tapping- or pen-based lasso-gestures.

By default, dragging an attribute onto the canvas creates a histogram showing the distribution of the data. Users can also modify these visualizations to plot different attributes against each other (e.g., height vs. weight), producing a two-dimensional histogram.

Arranging visualizations in VIZDOM is as simple as manipulating them through direct touch manipulation. In VIZDOM, we support two distinct yet interrelated type of links: persistent and transient links. Persistent links are created using a pen-gesture while transient links are inferred by the proximity of two elements. In the following, we map both of these types of links to filtering and brushing operations, described below.

Filter

Using a persistent link, a user can create detailed workflows where each visualization acts as a filter to downstream visualizations. For example, Figure 2.3 shows different workflows that a user can create when exploring data about automobiles. A set of linking operations turns the three visualizations

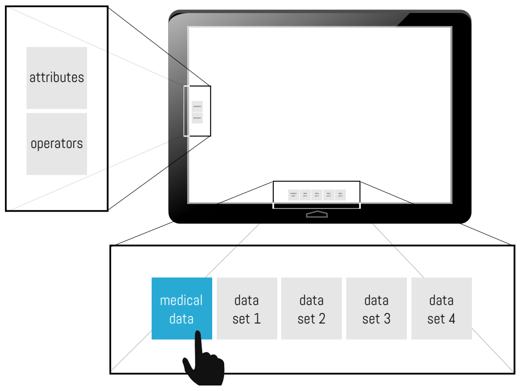


Figure 2.2: Vizdom Frontend

in Figure 2.3 (a) into a custom workflow (b) that can be used to filter the data down to cars with a high miles per gallon (mpg) value and low horsepower (hp) along with cars that originate from “Country 1.” Any change in an upstream visualization triggers downstream visualizations to be recomputed automatically.

Brush

When the user drags two visualizations close together, Vizdom implicitly creates a transient link between them. This action maps to a linking and brushing operation, where selected data points are highlighted in another visualization. For example, as shown in Figure 2.3 (c), the horsepower (hp) vs. miles per gallon (mpg) visualization is dragged close to the displacement (dis) visualization. From this, the user can see that cars with high miles per gallon values and low horsepower are all within the lower end of the displacement range. Moving the origin visualization close to the displacement visualization (d) reveals that there is little overlap between cars with a high mpg/low horsepower and cars from “Country 1” as well as the fact that cars with a high displacement value come all from “Country 1”.

Vizdom assigns a unique color to each brushed visualization and data points that are associated to two or more brushes are colored black. The height of each bar as well as the area of each area in a 2D scatterplot is scaled based on the number of data points that fall within a specific brush. As with filtering operations, any change in brush-selections automatically trigger the system to recompute any downstream visualizations.

2.1.2 Operators

When analyzing a dataset, analysts often create or derive new variables from existing values. Examples range from simple transformations (e.g., normalizations, log transforms or a user defined calculation) to computing complex statistical models over the dataset. Vizdom supports these kinds of derived values through operators.

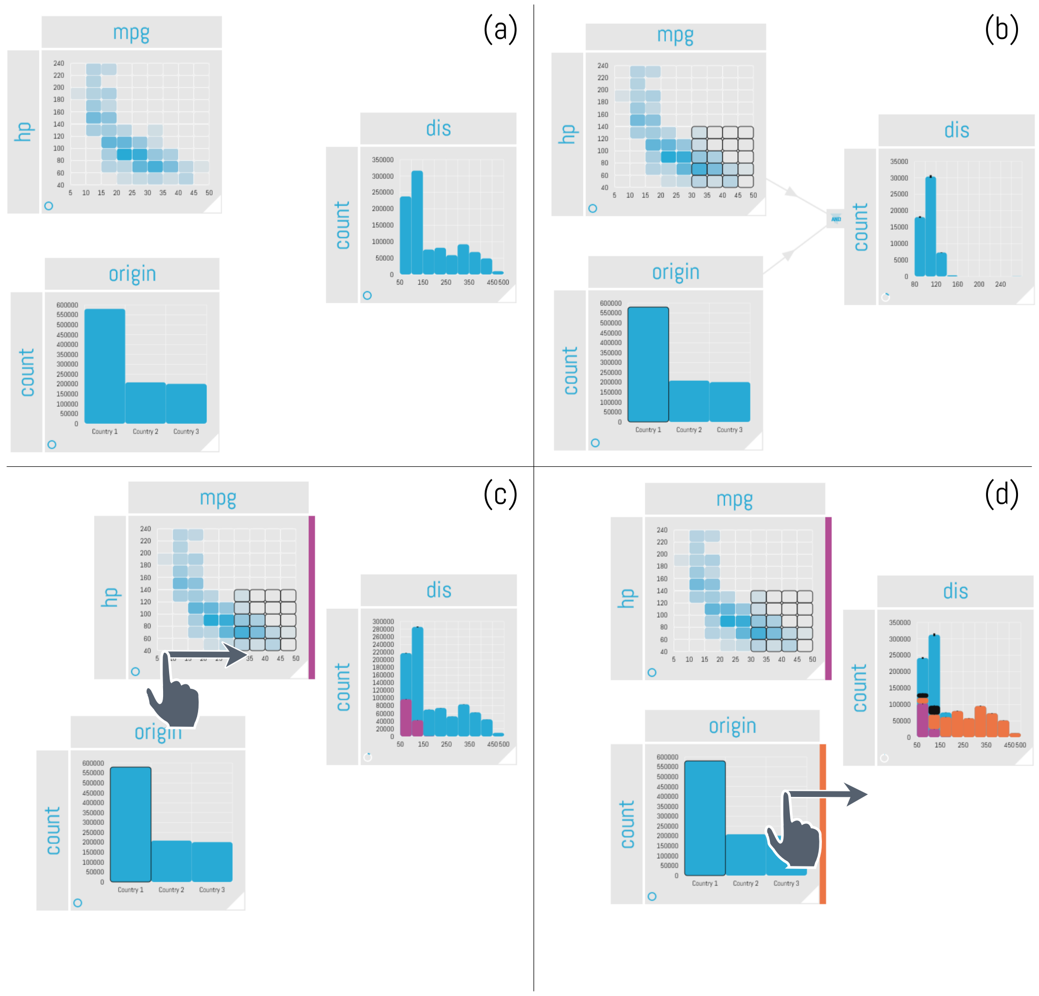


Figure 2.3: Examples of filtering and brushing. (a) Three uncoordinated visualizations. (b) Visualizations coordinated through persistent links where upstream selections serve as filter predicates for downstream visualizations. (c) Example of a brushing interaction through a transient link that is based on proximity. (d) Multiple brushes on the same visualization.

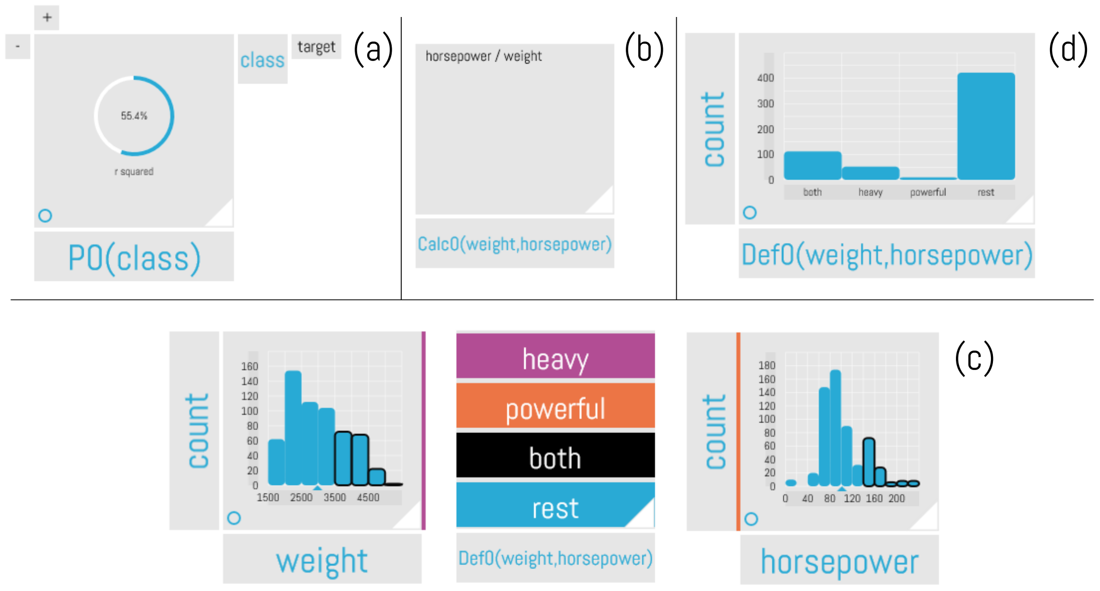


Figure 2.4: Examples of operators. (a) A prediction operator that predicts "class" (b) A user defined calculation. (c) A definition operator which defines a new derived attribute based on user specified brushes. (d) Visualization of the new attribute created in (c).

Dragging an operator onto the canvas provides a template that users can parameterize with attributes from the dataset. For example, a user can create a prediction model by dragging out the corresponding operator from the operator list. Such an operator needs two inputs before it can start computing a model: (1) the set of features to train over and (2) a label function that indicates which values it should be predicting. In Vizdom, both of these inputs are specified by dragging and dropping attributes to either the operator's feature or label slot. Figure 2.4 (a) shows such a prediction operator that was initialized to predict "class" based an all available features.

Other examples of operators in Vizdom include user-defined calculations or definitions, which allow users to perform custom logic over the data. Figure 2.4 (b) shows a user-defined calculation, where the user creates a new derived value based on a computation involving "horsepower" and "weight". Figure 2.4 (c) is an example of a user-defined definition, where the user creates a new categorical variable based on selections in visualizations. The output of all operators can be renamed and used as any other attribute. By dragging the output of the definition operator in Figure 2.4 (c) a user can visualize the distribution of values of this new attribute (d).

2.2 IDEA

A key requirement of visual data exploration tools like Vizdom is the ability to provide query results at "human speed," even over large datasets. For example, a recent study [93] demonstrated that delays longer than even 500ms negatively impact user activity, dataset coverage, and insight discovery.

As dataset sizes continue to increase, traditional DBMSs that are designed around a blocking query execution paradigm (i.e., return query results only after the system computes the full result) cannot scale to meet these interactivity requirements. To circumvent this issue, several techniques have been proposed that pre-aggregate data on top of a DBMS (e.g., materialized views, data cubes [54]). However, these techniques generally require extensive preprocessing and suffer from the curse of dimensionality. Unfortunately, attempts to overcome these problems (e.g., imMens [94], NanoCubes [90]) usually restrict the number of attributes that can be filtered at the same time, severely limiting the possible exploration paths.

Rather, in order to achieve low latency for ad hoc queries, new systems targeting interactive data exploration must instead rely upon *approximate query processing* (AQP) techniques, which provide query result estimates with bounded errors. Many AQP systems [58, 72], though, make the unrealistic assumption that they can fully subsume all data management responsibilities. Unfortunately, most organizations have accrued considerable technical debt, making an overhaul of their existing data management infrastructures prohibitively expensive. Instead, to be useful, a system for interactive data exploration must integrate and work seamlessly with these existing infrastructures, including data warehouses and distributed file systems.

Interactive Data Exploration Accelerators [34] (IDEAs) bridge this gap by applying AQP techniques while sitting on top of, rather than attempting to replace, an existing data storage layer. IDEAs therefore serve as an intelligent middleware layer that seeks to improve the interactivity of visual data exploration frontends like Vizdom through a variety of techniques, including caching, prefetching, and result reuse.

2.2.1 Neither a DBMS nor a Streaming System

An IDEA is neither a DBMS nor a streaming engine, instead has an entirely unique set of semantics. Unlike DBMSs, queries are not one-shot operations that return batch results; rather, workflows are constructed incrementally, requiring fast response times and progressive results that refine over time. Unlike, streaming engines which traditionally deploy predefined queries that do not change over infinite data streams, an IDEA is meant to enable free-form exploration of data sampled from a deterministic system (e.g., a finite data source). Fundamentally, an IDEA acts as an intelligent, in-memory caching layer that sits in front of the much slower data sources, managing both progressive results and the samples used to compute them. Often, an IDEA also has the opportunity to offload pre-filtering and pre-aggregation operations to an underlying data source (e.g., perform a predicate pushdown to a DBMS), or even transform the base data by executing a custom UDF in an analytics framework.

Finally, in contrast to traditional DBMSs and streaming engines, users of visual data exploration tools compose queries incrementally, resulting in simultaneous visualizations of many component results with varying degrees of error. Maintaining different component partial results instead of single, exact answers imposes a completely new set of challenges for both expressing and optimizing these types of queries.

2.2.2 Architecture

Figure 2.1 shows the full execution path through the system, from issuing a query in the frontend to the final progressive result visualization. As the first step in an exploration session, a user will typically connect to a completely new data source (e.g., census data stored in a data warehouse). Upon connecting, an IDEA immediately begins to populate the *Sample Store* by streaming tuples from the underlying data source. The Sample Store caches a subset of tuples from the data source (or the entire dataset if memory permits) and serves as the basis for answering queries using our AQP techniques described in Chapter 3.

In addition to computing the frequencies, the Query Engine also constructs *Tail Indexes*, low-overhead index structures built on a subset of tuples in the Sample Store that help provide high-quality approximations for subsequent queries on rare subpopulations. Finally, the frontend polls the Result Cache to receive the current approximation for each of the group-by values in the specified visualization.

2.2.3 Sample Management

As previously mentioned, an IDEA tries to cache as much data as possible from the underlying data sources in order to provide faster approximate results, since most data sources are significantly slower. For example, even though the memory bandwidth of modern hardware ranges from $40-50$GB/s per socket [19], we recently measured that PostgreSQL and a commercial DBMS can only export $40-120$MB/s, even with a warm cache holding all data in memory. Although DBMS export rates may improve in the future, an IDEA's cache will still remain crucial for providing approximate answers to visual queries and supporting more complex analytics tasks (e.g., ML algorithms).

Ideally, an IDEA would cache the entire dataset in an efficient manner in order to be able to quickly provide approximate results to the visual frontend. However, for large datasets, this is not always possible. Therefore, if the cached data exceeds the available memory, an IDEA needs to carefully evict stored tuples while retaining the most important data items in memory. For example, caching strategies like LRU do not necessarily maintain a representative sample. Therefore, an IDEA uses reservoir sampling instead to evict tuples while preserving randomness. Furthermore, an IDEA also needs to maintain a set of disproportionate stratified samples that overrepresent uncommon data items in order to support operations over rare subpopulations.

The necessity to maintain different types of potentially overlapping samples poses many important research challenges. For example, deciding when and what to overrepresent is a very interesting problem, though an IDEA currently employs a greedy algorithm that attempts to create a stratified sample for each operation. Furthermore, an IDEA can even use stratified samples to overcome the label imbalance problem encountered by many ML algorithms (e.g., overrepresenting the rare subpopulation of individuals with a Ph.D. when training a classifier to predict education level). Such decisions can have profound effects on the overall performance, error estimates, and a user's understanding of the model, all of which we are still actively investigating.

2.3 Backends

The final component of our IDE stack is a backend data source. Since a potential user would ideally like to explore any new dataset, a backend data source can be anything from a raw data file (e.g., CSV file) to a data warehouse or analytics system.

As previously mentioned, an IDEA reads tuples from a given data source and caches a subset of the data to use when approximating a query result. Therefore, the only step required in order to add the functionality necessary to support a new data source is to implement a simple function that returns tuples from the underlying data source. However, given that data sources are often complex systems that may store data in a particular way (e.g., sorted), we must pay special attention to ensure that the underlying data source returns tuples in a random fashion. Since the AQP techniques that IDEAs use to compute approximate query results rely on randomness to get a good approximation, we must use methods to access the underlying data source in a random order. Depending on the data source, we can utilize techniques such as reservoir sampling or randomized index traversal [108] to ensure randomness.

Additionally, a wide variety of systems are commonly used in today's diverse data management landscape since specialized systems for specific use cases (e.g., transactional workloads, complex analytics) and data (e.g., time series, array, text) can offer substantial performance improvements over a one-size-fits-all DBMS [135]. In particular, given the ubiquity of time series data, it is not surprising that there has been considerable interest in systems specifically designed to manage time series data, including Gorilla [116], Metronome [97], InfluxDB [67], OpenTSDB [110], TimescaleDB [139], CrateDB [31], and Druid [147]. Many of these systems treat the time attribute of a dataset in a unique way, usually either physically ordering tuples by their timestamp or by building specialized indexes over timestamp attribute. For example, TimescaleDB organizes data into hypertables partitioned into disjoint chunks that are each individually indexed, enabling efficient search for small time intervals. Therefore, an IDEA must be able to interact with a wide variety of possible data sources, each of which may have different features, including specialized index structures and storage formats.

In addition to being able to support specialized systems, an IDEA should also be able to leverage any auxiliary data structures that a backend system might maintain in order to speed up query processing and increase interactivity. As discussed previously, an IDEA populates an in-memory cache of the data stored in an underlying data source (e.g., data warehouse). In the simplest case, an IDEA streams tuples from the underlying data source without specifying any predicates over the data. However, since users are often interested in small portions of the data (e.g., only customers from a given region, order placed in the last day), an IDEA can, in some cases, request only relevant tuples from the underlying data source. Since indexes can help the underlying data source efficiently provide relevant tuples, an IDEA should therefore try to leverage any existing indexes maintained by the underlying system to ensure that only relevant data items are cached.

Chapter 3

Approximate Query Processing for Interactive Data Exploration

As previously mentioned, traditional DBMSs are designed around a blocking query execution paradigm whereby query results are returned only once a fully complete result is computed. Since dataset sizes are continuing to increase rapidly, it is not surprising that blocking approaches cannot scale to meet the interactivity requirements of interactive data exploration. In fact, as we show in [152], blocking visualizations negatively affect user engagement, while visualizations that are computed progressively are almost equivalent to the theoretical best case where all visualizations are computed instantly. Therefore, to satisfy the strict interactivity requirements, new systems targeting interactive data exploration must instead rely upon *approximate query processing* (AQP) techniques, which provide query result estimates that refine over time with bounded errors.

To better support the conversational querying paradigm that visual data exploration systems promote, we first present the basics of our novel approximate query processing formulation that an IDEA uses. Next, we show how our AQP formulation maximizes the reuse potential of approximate results across queries during an exploration session and present several probability-based rewrite rules. Then, to further improve the accuracy of our approximations, we present a new low-overhead index structure that leverage a user's "think time" and also discuss more complex queries and operations. Finally, using several real-world datasets and exploration sessions, we demonstrate that our IDEA prototype that uses our AQP formulation can achieve interactive latencies in many cases where alternative approaches cannot.

3.1 AQP Formulation

In interactive data exploration tools, many types of visualizations (e.g., bar charts, histograms) serve to convey high level statistics about datasets, such as value distributions or relationships between attributes. As previously mentioned, interaction visual data exploration tools is conversational in nature, since users incrementally build up and iteratively refine queries by manipulating visualizations.

Many AQP systems use some form of biased sampling [2] (e.g., AQUA [3], BlinkDB [4], DICE [76]), but these approaches usually require a priori knowledge of the workload or substantial preprocessing time. While useful for many applications, this approach is at odds with interactive data exploration, which is characterized by the free-form analysis of new datasets where queries are typically unknown beforehand.

On the other hand, AQP systems that perform online aggregation [59] (e.g., CONTROL [58], DBO [72], HOP [28], FluoDB [151]) can generally only provide good approximations for queries over the mass of the distribution, while queries over rare subpopulations yield results with loose error bounds or runtimes that vastly exceed the interactivity threshold. Since the exploration of rare subpopulations (e.g., high-value customers, anomalous sensor readings) often leads to the most significant insights, online aggregation falls short when confronted with these types of workloads.

We therefore argue that neither biased sampling nor online aggregation is ideal in this setting, since systems for interactive data exploration must provide high quality approximate results for any query—even over rare subpopulations—at interactive speeds without requiring foreknowledge of the workload or extensive preprocessing time. While seemingly impossible, these systems have the opportunity to leverage the unique properties of interactive data exploration. Specifically, visual tools have encouraged a more conversational interaction paradigm [68], whereby users incrementally compose and iteratively refine queries throughout the data exploration process. Moreover, this style of interaction also results in several seconds (or even minutes) of user "think time" where the system is completely idle. Thus, these two key features provide an AQP system with ample opportunities to (1) reuse previously computed (approximate) results across queries during a session and (2) take actions to prepare for possible future queries.

In this section, we first introduce our AQP formulation, which is an extension of online aggregation that treats aggregate query results as random variables. We then describe how our prototype IDEA implementation executes an aggregate query using this formulation, as well as how we quantify the error associated with approximate results returned to the user.

3.1.1 Formulating a Simple Query

The simplest type of visualization we consider is a bar chart that expresses the count of each attribute value. For example, Step A of Figure 1.2 shows a bar chart of the *sex* attribute, which can be represented by the following SQL query:

```
SELECT sex, COUNT(*)
FROM census
GROUP BY sex
```

One way to model a group-by attribute is to treat it as a categorical random variable X, where X can assume any one of the possible outcomes from the sample space Ω_X. For example, a random variable modeling the *sex* attribute can take on an outcome in the sample space $\Omega_{sex} = \{Male, Female\}$.

The discrete probability distribution θ_X represents the likelihood that X will assume each outcome, with θ_x denoting the probability that X will assume outcome x. In the example query, for instance, the probability of observing a 'Male' tuple is $\theta_{Male} \approx 2/3$.

As previously mentioned, an AQP engine needs to use techniques to compute an approximation $\hat{\theta}_X$ of the true θ_X, since processing a large dataset in a blocking fashion can easily exceed interactivity requirements. Therefore, for each distinct value x in Ω_X, *maximum likelihood estimation* (MLE) can produce an approximation $\hat{\theta}_x^{MLE}$ of the true θ_x by scanning a random sample of the underlying data to compute:

$$\hat{\theta}_x^{MLE} = \frac{n_x}{n} \tag{3.1}$$

Equation 3.1 shows that each θ_x is approximated by dividing n_x (i.e., the number of observed instances of x in the sample) by the total sample size n. Intuitively, this approximation represents the estimated *frequency* of x in the dataset. Multiplying a frequency estimate $\hat{\theta}_x$ by the total number of tuples N in the full dataset will therefore yield an approximate count for x:

$$count(x) \approx N\hat{\theta}_x \tag{3.2}$$

These techniques can be extended to support histograms (i.e., a count grouped by a continuous attribute like *age*) by partitioning the domain of the group-by attribute into finite bins which can be treated as a nominal attribute. We use similar techniques to compute other aggregates (e.g., average, sum) and more complex visualizations (e.g., heatmaps), as well as queries involving joins.

3.1.2 Executing a Query

To understand how to execute a query and produce a visualization using the formulation presented in the previous section, consider again the example query representing the *sex* bar chart (Step A in Figure 1.2). Figure 2.1 shows the execution path of a query through the system.

Upon connecting, the AQP engine immediately begins to populate the *Sample Store* by streaming in tuples from the underlying data source. Since our techniques rely on a random sample in order to get a good approximation, we use methods to access the underlying data source in a random order (e.g., reservoir sampling, randomized index traversal [108]).

When the user creates the visualization shown in Step A of Figure 1.2 in the frontend by dragging out the *sex* attribute to the screen, the *Translator* receives the visualization parameters (i.e., attribute, aggregate function, selection predicates) and generates a corresponding SQL query. The *Query Engine* then executes this query by spawning several executor threads, which draw tuples from the Sample Store to compute an approximate result.

When reading these tuples, the Query Engine uses our AQP formulation to compute $\hat{\theta}_{sex}$ (using Equation 3.1) for all attribute values in *sex* (i.e., $\hat{\theta}_{Male}$ and $\hat{\theta}_{Female}$). The resulting probabilities (e.g., $\hat{\theta}_{Male} \approx 2/3$) are stored in the *Result Cache*. The Query Engine can leverage these cached results (e.g., Step B can reuse $\hat{\theta}_{Female}$) for computing the results to future queries, explained further in Section 3.2.

Finally, the frontend polls the Result Cache to receive the current approximation for each of the group-by values in the specified visualization.

3.1.3 Quantifying Result Error

As previously mentioned, the Query Engine updates the approximate results that are stored in the Result Cache as more data is processed. Therefore, we need a way of quantifying the uncertainty, or *error*, associated with each approximate result.

In the context of our AQP formulation, the previously described estimators are said to be asymptotically normal with the sample size n; that is, the estimator $\hat{\theta}_x$ is equal to the actual θ_x plus some normally distributed random noise given a sufficiently large sample. The *standard error* approximates this deviation of the observed sample from the true data. Formally, the normalized standard error of $\hat{\theta}_x$ is given by the square root of the variance of a categorical random variable divided by the sample size n, then normalized by $\hat{\theta}_x$:

$$error(x) = \frac{1}{\hat{\theta}_x}\sqrt{\frac{\hat{\theta}_x(1-\hat{\theta}_x)}{n}} \tag{3.3}$$

In order to calculate the error for a query grouped by an attribute X (e.g., the query to compute the *sex* bar chart in Step A of Figure 1.2), we compute the sum of the relative standard errors (Equation 3.3) for all attribute values $x \in \Omega_X$. Note, however, that metrics other than the sum (e.g., average, max) or alternative definitions [81] can also be used to quantify the error associated with an approximate result.

3.2 Approximate Result Reuse

As previously mentioned, our AQP formulation is designed to leverage the unique reuse opportunities that exist in interactive data exploration sessions. In this section, we first show how our formulation extends to queries with selections and allows us to integrate the results of past queries that have already been (partially) computed. To facilitate result reuse, we also introduce several new rewrite rules based in probability theory that enable the Query Engine to optimize queries similarly to how a traditional DBMS uses relational algebra rules to optimize SQL queries. Finally, since we are reusing approximate results, we discuss the intricacies of error propagation and how to choose among several possible rewrites.

3.2.1 Queries with Selections

Section 3.1.1 showed how we can use our AQP formulation to express a single bar chart. One of the key features of data exploration tools, though, is the ability to create complex *filter chains* in order to explore specific subpopulations; that is, visualizations act as active controls and can be linked together, where selections in upstream visualizations serve as filters for downstream visualizations. For example, Step B of Figure 1.2 shows a filtered *salary* bar chart for only the 'Female' subpopulation, which translates to the following SQL query:

```
SELECT salary, COUNT(*)
FROM census
WHERE sex = 'Female'
GROUP BY salary
```

To answer this query, we need to estimate the probability of each possible *salary* value for only the 'Female' subpopulation. More formally, given a group-by attribute X and a selection attribute $Y = y$, we are trying to estimate the *joint probability* $\theta_{x,y}$ for every x value in Ω_X. By extending Equation 3.1 from Section 3.1.1, we can approximate each $\theta_{x,y}$ using the MLE:

$$\hat{\theta}^{MLE}_{x,y} = \frac{n_{x,y}}{n} \tag{3.4}$$

Reformulating Joint Probabilities

Although formally correct, the estimator $\hat{\theta}^{MLE}_{x,y}$ given in Equation 3.4 introduces a new issue that does not arise in the simple case of estimating θ_x. Namely, computing $\hat{\theta}^{MLE}_{x,y}$ the same way as $\hat{\theta}^{MLE}_{x}$ (described in Section 3.1.2) considers all estimates as independent, which could lead to inconsistencies [145] across different visualizations. For instance, an inconsistency could arise in Step B of the example exploration session if the sum of the two estimated *salary* values exceeded the total number of all 'Female' tuples, which is clearly impossible.

Since our AQP formulation treats query results as random variables, we can avoid this issue by leveraging the *Chain Rule* from probability theory to rewrite $\hat{\theta}_{x,y}$ into a different form:

$$\hat{\theta}_{x,y} = \hat{\theta}_{x|y}\hat{\theta}_y \tag{3.5}$$

In this new form, we can therefore estimate $\theta_{x,y}$ by reusing the previously computed estimate $\hat{\theta}_y$ and computing a new estimate for $\theta_{x|y}$, again using the MLE:

$$\hat{\theta}^{MLE}_{x|y} = \frac{n_{x|y}}{n_y} \tag{3.6}$$

For example, we can apply the Chain Rule to rewrite the query from Step B of Figure 1.2 to be conditioned on $sex = `Female'$, where we can estimate the joint probability by computing $\hat{\theta}^{MLE}_{salary|Female}$ and reusing the previously computed $\hat{\theta}_{Female}$. Note that we can equivalently rewrite the joint probability as $\hat{\theta}_{x,y} = \hat{\theta}_{y|x}\hat{\theta}_x$, and the Query Engine is free to choose whichever alternative produces the lowest error result in the shortest time (discussed further in Section 3.1.3). Again, since we multiply the conditional probability by the selected subpopulation (e.g., 'Female'), our estimates are no longer independent; that is, the sum of the estimates for values within a subpopulation must be strictly smaller than the estimate of the subpopulation itself, thereby addressing the visual inconsistency issue.

Very Rare Subpopulations

As previously mentioned, queries over rare subpopulations occur frequently in interactive data exploration settings, since they often produce the most interesting and valuable insights. Although the rewritten MLE that uses the conditional probability will provide a good approximation for the mass of the distribution, it is limited by the number of relevant tuples that are observed, yielding results with high error in the tails of the distribution.

To solve this problem, we can reuse the prior information available in the Result Cache in order to get a better estimate faster than scanning the Sample Store and filtering only for the rare subpopulation. Since the MLE cannot incorporate these priors, we can extend our formulation from Equation 3.1 to be able to include any available prior information. A *maximum a posteriori* (MAP) estimator incorporates an α term to represent prior information, where the estimate represents the mode of the posterior Dirichlet distribution:

$$\hat{\theta}^{MAP}_{x,y} = \frac{n_{x,y} + \alpha_{x,y} - 1}{n + \alpha - 2} \tag{3.7}$$

If no prior information is used (i.e., all possible parameters are equally likely), or if n approaches infinity (i.e., the impact of the prior goes to zero), then the MAP estimate is equivalent to using the MLE. Each of these scenarios occur, respectively, when we have no prior information about a distribution (e.g., as in Step A of the example before the user has issued any queries), or when a selection occurs over the mass of the distribution, thereby guaranteeing a large n in a short amount of time. Therefore, the MAP estimate can handle cases with very rare subpopulations by incorporating a prior while also providing the same performance as an MLE over the mass of the distribution.

For example, in the previous query (Step B), we are trying to compute $\hat{\theta}_{salary|Female}$. Since the Result Cache already contains an estimate for the different *salary* subpopulations (i.e., $\hat{\theta}_{High}$ and $\hat{\theta}_{Low}$), we can leverage these already computed estimates by providing them as a prior to their respective conditional estimates.

Unfortunately, when trying to incorporate a prior, the weighting is the hardest part [118], since overweighting will ignore new data while underweighting might produce a skewed estimate based on too few samples. Therefore, we pick a small α proportional to the expected frequency of the estimate, such that the significance of the prior will quickly diminish as more data is scanned unless the subpopulation is very rare, in which case the prior will help to smooth an estimate based on a tiny sample. As in Equation 3.3, we can use the finite population correction factor to reduce the impact of the prior as more data is scanned (e.g., a prior should have no impact after scanning all tuples). Again, we leave this optimization for future work.

3.2.2 Query Rewrite Rules

The previous section showed how to apply the Chain Rule in order to rewrite a query represented as a joint probability into a form that can provide a better estimate by reusing previously computed results. Thus, a natural follow up question is: can we leverage other rules from probability theory to rewrite different types of queries in order to maximize reuse potential?

Similar to a DBMS query optimizer that uses relational algebra rules to rewrite queries, our AQP formulation unlocks a whole new set of rewrite rules based on probability theory. By using these rewrite rules, the Query Engine can often leverage past results to answer some queries much faster than having to scan the Sample Store in order to compute a result from scratch. In fact, as our experiments show (Section 3.5), rewrite rules can even return results for certain queries almost instantaneously.

In the following, we describe example queries along with corresponding opportunities to apply rewrite rules during the query optimization process. Note that these rewrite rules are only possible due to our AQP formulation, and techniques that treat queries as "one-shot" (e.g., online aggregation) would need to resort to scanning the Sample Store in order to compute the result for each new query.

Bayes' Theorem

In many cases, users often wish to view the distribution of an attribute X filtered by some other attribute Y. Then, in order to get a more complete picture, they will reverse the query to view the distribution of Y for different subpopulations of X. Based on this exploration path (i.e., X filtered on Y switched to Y filtered on X), the Query Engine can use *Bayes' Theorem* to compute an estimate without having to scan any tuples. Formally, Bayes' Theorem relates the probability of an event based on prior knowledge of related conditions:

$$\hat{\theta}_{y|x} = \frac{\hat{\theta}_{x|y}\hat{\theta}_y}{\hat{\theta}_x} \tag{3.8}$$

For example, suppose that the user reverses the direction of the link in Step B of Figure 1.2 (i.e., *sex* is now filtered by *salary*) and selects the 'High' bar, yielding the following SQL query:

```
SELECT sex, COUNT(*)
FROM census
WHERE salary = 'High'
GROUP BY sex
```

In this example, the Result Cache has (partially) complete estimates for all attribute values for the *salary* attribute conditioned on the attribute values for *sex* (i.e., $\hat{\theta}_{Low|Female}$, $\hat{\theta}_{High|Female}$, $\hat{\theta}_{Low|Male}$, $\hat{\theta}_{High|Male}$) from the query in Step B. Using Bayes' Theorem therefore allows us to leverage the previous results for (1) $\hat{\theta}_{sex}$, (2) $\hat{\theta}_{salary}$, and (3) $\hat{\theta}_{salary|sex}$ to instantly compute $\hat{\theta}_{sex|salary}$ without needing to scan any tuples from the Sample Store.

Law of Total Probability

Another common exploration pattern is to view the distribution of an attribute X for different subpopulations of Y, switching between filtering conditions to observe how the distribution of the downstream attribute changes. Recall that categorical random variables have the constraint that all outcomes are mutually exclusive (i.e., the probabilities sum to one). We can therefore leverage the *Law of Total Probability* to rewrite queries involving different selections over mutually exclusive subpopulations in order to reuse past results. The Law of Total Probability defines the probability of an event x as the sum of its marginal probabilities:

$$\hat{\theta}_x = \sum_{y \in \Omega_Y} \hat{\theta}_{x,y} \tag{3.9}$$

In Step C of Figure 1.2, for example, the user has negated the selection condition of the *sex* attribute, which translates to the following query:

```
SELECT salary, COUNT(*)
FROM census
WHERE sex <> 'Female'
GROUP BY salary
```

In this example, the Result Cache already has frequency estimators (1) $\hat{\theta}_{salary}$ and (2) $\hat{\theta}_{salary|Female}$. We can use the Law of Total Probability to marginalize across the *sex* attribute in order to compute $\hat{\theta}_{salary|\ Female}$, again without actually having to scan any data.

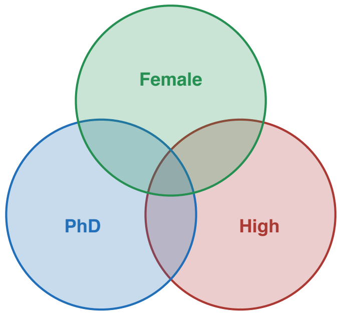
Inclusion-Exclusion Principle

Unlike filter chains, where downstream visualizations represent strict subpopulations of upstream visualizations, users can also specify predicates that represent the intersection of selected subpopulations. For example, the top right bar chart in Step D of Figure 1.2 shows the intersection of the 'Female' and 'PhD' subpopulations, and the bottom right bar chart shows the intersection between non-'Female' and 'PhD' subpopulations, which translates to the following SQL query:

```
SELECT salary, COUNT(*)
FROM census
WHERE sex <> 'Female' AND education = 'PhD'
GROUP BY salary
```

In the example, we must compute $\hat{\theta}_{salary,PhD,\neg Female}$. By using the *Inclusion-Exclusion Principle* (IEP) from probability theory, we can rewrite the query to entirely reuse past results stored in the Result Cache without having to scan any tuples in the Sample Store. In order to understand this rewrite rule, consider the rewritten query and the accompanying Venn diagram:

$$\begin{aligned}\hat{\theta}_{High,PhD,\neg Female} = {} & \hat{\theta}_{High} + \hat{\theta}_{PhD} \\ & - \hat{\theta}_{High,\neg PhD} \\ & - \hat{\theta}_{\neg High,PhD} \\ & - \hat{\theta}_{High,PhD,Female}\end{aligned}$$



Each of the terms in the previous equation maps to a region of the Venn diagram. For example, the red circle represents $\hat{\theta}_{High}$, and the region of the red circle not overlapping with the blue circle represents $\hat{\theta}_{High,\neg PhD}$. Visually, we can see that $\theta_{High,PhD,\neg Female}$ can be rewritten in many ways, and the above equation is one way to rewrite the query that uses only previously computed estimates from the running example available in the Result Cache.

The IEP is a very powerful rewrite rule and can be applied to a broad range of additional queries by considering the relationship between predicate attributes. For example, if the user switches a Boolean operator (e.g., changing the predicate to `sex<>'Female' OR education='PhD'`), we can calculate the frequency of the union of two subpopulations simply by reusing our estimate for the intersection.

We can also use the IEP to take advantage of the mutual exclusivity of certain predicates. In particular, if the user applies a predicate representing the intersection of mutually exclusive subpopulations, we can apply the IEP to determine that no tuples can possibly exist in the result, therefore immediately returning a frequency estimate of zero (e.g., a query with predicate `sex='Male' AND sex='Female'` has a frequency of zero). Similarly, if the user applies a predicate representing the union of mutually exclusive subpopulations, we can again apply the IEP to immediately return a frequency equal to the sum of the subpopulations (e.g., a query with predicate `sex='Male' OR sex='Female'` has a frequency equal to $\hat{\theta}_{Male} + \hat{\theta}_{Female}$).

3.2.3 Result Error Propagation

As previously mentioned, users need to know how much faith to place in an approximate query result. Section 3.1.3 shows that computing the error for a simple query with no selections is relatively straightforward, but, when reusing past results, we need to carefully consider how the (potentially different) errors from each of the individual approximations contributes to the overall error of the new query result. Therefore, we apply well-known propagation of uncertainty principles in order to formally reason about how error propagates across overlapping queries.

For example, consider again the query representing Step B of Figure 1.2. In this case, we approximate the result of the query as $\hat{\theta}_{salary|Female}$ multiplied by $\hat{\theta}_{Female}$, so we need to consider the error associated with both terms. Note that each of these terms has a different error; that is, the error for the estimate $\hat{\theta}_{Female}$ is lower because it was started earlier (in Step A). After computing the sum of the normalized standard error (Equation 3.3) across all attribute values (Section 3.1.3), we combine the error terms for $\hat{\theta}_{salary|Female}$ and $\hat{\theta}_{Female}$ using the propagation of uncertainty formula for multiplication, yielding the estimated error for the final result.

Interestingly, the error estimates associated with the approximate results are also useful during the query optimization process. Since queries can often be rewritten using several different rewrite rules into many alternative forms, the Query Engine can pick the series of rewrites that produces an approximate query result with the lowest expected error through a dynamic programming optimization process that recursively enumerates rewritten alternatives. In some cases, it is possible that leveraging past results might produce an approximate result with higher error than simply computing an estimate by scanning the Sample Store, so the Query Engine must decide if a rewrite is beneficial at all.

3.3 More Complex Queries

So far, we have described how our AQP formulation applies in the context of count queries with selection predicates. We now expand our discussion to queries with (1) multiple group-by attributes, (2) aggregate functions other than count, and (3) joins.

3.3.1 Multiple Group-by Attributes

Rather than changing the selection predicate for a filtered bar chart to understand the relationship between two attributes, users sometimes find it easier to view them in a two-dimensional heatmap, where the gradient of each cell represents the count. For example, a heatmap showing the *sex* attribute plotted against *salary* translates to the following query:

```
SELECT sex, salary, COUNT(*)
FROM census
GROUP BY sex, salary
```

To build a heatmap over two attributes X and Y, we must estimate the joint probability for each cell (i.e., $\hat{\theta}_{x,y}$ for every combination of x and y). We can again use the Chain Rule (Section 3.2.1) to rewrite $\hat{\theta}_{x,y}$ as either (1) $\hat{\theta}_{x|y}\hat{\theta}_{y}$ or (2) $\hat{\theta}_{y|x}\hat{\theta}_{x}$, and then we calculate the error using the previously described error propagation techniques (Section 3.2.3). As explained, since the Query Engine has a choice between how to rewrite the query, we can automatically pick the rewrite that minimizes error in order to return a better approximation.

Since heatmaps can also act as filters for any linked downstream visualization, we similarly index rare attribute values in order to efficiently answer future queries over selected subpopulations, which in this case are comprised of some combination of (x, y) pairs (i.e., a two-part conjunctive predicate). We can then use the previously described index weighting techniques (Equation 3.10) to proportionally scan indexes in order to preserve randomness.

3.3.2 Other Aggregate Functions

Unlike nominal attributes, users can apply other aggregate functions (e.g., average, sum) in addition to counts. In order to compute the estimates and the error for a query involving other aggregate functions, we need to modify our previously described techniques and incorporate properties about these aggregates into our formulation. Furthermore, unlike for a simple count, the error of other aggregate functions for a particular bin is not entirely influenced by the frequency of that bin. Intuitively, observing a tuple with any attribute value will lower the total error for a count visualization by increasing the size of the sample n in the denominator (Equation 3.1). On the other hand, for an aggregate like average, observing a tuple with a value in a particular bin will only impact the error for the observed bin. For this reason, we need to take special care to compute the error for visualizations that depict average and sum aggregates, including normalizing their errors to the same range as count visualizations.

Average

To compute the error of an estimated average for a single bin x of a continuous attribute, we use the sample standard deviation of tuples that belong in that bin. First, we compute the average squared distance of each value in the bin to the bins's mean. Then, we divide the average squared distance by n and take the square root to yield the sample standard deviation.

Sum

The error associated with a sum estimate for an attribute value x requires estimates for both the count and average of the attribute value. For this reason, by default, we always estimate average and count for all continuous attributes, as well as the corresponding error estimates. Then, to estimate the error associated with the average, we use the previously described error propagation techniques (Section 3.2.3) to combine the errors of the average and count estimates.

3.3.3 Joins

So far, we have focused on the scenario where a user explores a single table or denormalized dataset (e.g., materialized join results, star schema data warehouses). However, extending our AQP formulation to work on joins is useful in order to apply our techniques to more scenarios.

Joins pose several interesting questions, since our AQP techniques require a random sample. Unfortunately, DBMSs usually sort or hash data to compute the result of a join, which breaks our randomness requirement. However, existing techniques (e.g., ripple join [56], SMS join [71], wander join [88]) can be used to provide random samples from the result of a join. Therefore, we can apply our previously described AQP techniques over the result of a join between multiple tables to provide fast and accurate approximations. However, we leave extensions to our AQP formulation that optimize join processing as future work.

3.4 Tail Indexes

Since the data exploration process is user-driven and inherently conversational, the AQP engine is often idle while the user is interpreting a result and thinking about what query to issue next. This "think time" not only allows the system to continue to improve the accuracy of previous query results but also to prepare for future queries. Some existing approaches (e.g., ForeCache [16], DICE [76]) leverage these interaction delays to model user behavior in order to predict future queries, but these techniques typically require a restricted set of operations (e.g., pre-fetching tiles for maps, faceted cube exploration) or a large corpus of training data to create user profiles [112]. Instead, as

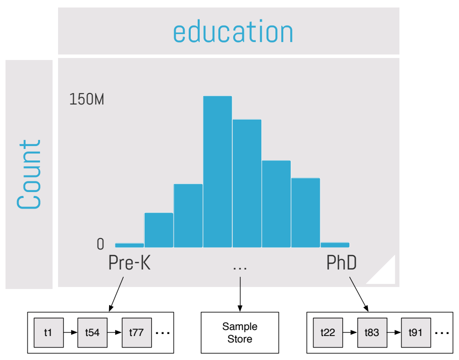


Figure 3.1: A Tail Index built on *education*.

described in Section 3.1.1, our approach leverages a user's "think time" to construct Tail Indexes in the background during query execution to supplement our AQP formulation.

This section first describes how to build a Tail Index on-the-fly based on the most recently issued query in preparation for a potential subsequent query on a related subpopulation. Then, we explain the intricacies of how to safely use these Tail Indexes to support future queries by preserving the randomness properties necessary for our AQP techniques. Finally, we discuss how to extend Tail Indexes to support continuous attributes.

3.4.1 Building a Tail Index

Many of the query rewrite rules described in Section 3.2.2 rely on observing tuples belonging to specific subpopulations (e.g., 'Male'), which will appear frequently in a scan of the Sample Store. However, when considering rare subpopulations (e.g., 'PhD'), tuples belonging to these subpopulations will not be common enough in a scan of the Sample Store to provide low-error approximations within the interactivity threshold. As such, we need to supplement our formulation with indexing techniques in order to provide enough relevant tuples for low-error approximations.

Many existing indexing techniques (e.g., B-trees, sorting, database cracking [62]) organize all tuples without regard for their frequency in the dataset, resulting in unnecessary overhead since our AQP techniques can already provide good approximations over common subpopulations. Furthermore, these indexing techniques destroy the randomness property that our AQP formulation requires, and trying to correct for the newly imposed order would be prohibitively expensive. Therefore, we propose a low-overhead partial index structure [134], called the *Tail Index*, that is built online during a user's exploration session. Tail Indexes dynamically keep track of rare subpopulations to support query rewrites and save space by not indexing tuples with common values, all while also preserving the randomness requirements necessary for our AQP techniques.

Figure 3.1 shows a sample Tail Index built on the *education* attribute from Step D of Figure 1.2. The Tail Index is a hash-based data structure that points to either (1) the Sample Store when an attribute value is common or (2) a linked list of pointers to tuples when the attribute value is rare. In the figure, notice that the attribute values in the mass of the distribution (e.g., 'HS') point to the

Sample Store, whereas the values in the tails of the distribution (e.g., ‘Pre-K’, ‘PhD’) point to lists of indexed tuples.

In order to build this index, though, we need to determine whether a given tuple belongs to a rare subpopulation. Specifically, we decide which attribute values should be indexed by determining if the specified confidence level for a possible future visualization will not be achievable within the interactivity threshold. That is, if the frequency of an attribute value is high enough to provide sufficient tuples from a given subpopulation in order to meet the specified confidence level from a scan of the Sample Store, then the index entry should instead point to the Sample Store. Otherwise, the Tail Index retains the linked list and continues appending new tuples until either the resources need to be reallocated or a maximum index size has been reached.

For longer chains of visualizations with multiple filter conditions, we build an index for each attribute in the chain. For example, if the user has the *sex* visualization linked to *salary*, as in Step B of the example, our techniques would build a multidimensional index on *sex,salary* (i.e., both the ‘Male’ and the ‘Female’ buckets point to an index of *salary*, each of which contains only males or females, respectively). However, since longer chains with selections result in increasingly rare subpopulations, the frequency $\hat{\theta}_x$ of the attribute must be adjusted for the value in the entire population. Therefore, if no upstream visualizations are indexed, $\hat{\theta}_x$ is the joint probability (i.e., $\hat{\theta}_{x,y}$). When an attribute value is indexed, subsequent downstream visualizations should instead use the conditional probability (e.g., $\hat{\theta}_{x|y}$) since logically the index defines a new population.

Typically, users cannot keep track of more than a few attributes at a time [47], ensuring that the indexes will not become larger than a few dimensions. Furthermore, the fact that Tail Indexes keep track of only rare subpopulations further reduces their size.

3.4.2 Using a Tail Index

Recall that the rewrite rules from Section 3.2.2 often require tuples that belong to specific subpopulations (e.g., $\hat{\theta}_{salary|Female}$ requires only tuples with a value of ‘Female’). For selection operations over a single attribute value (e.g., ‘Female’), scanning the indexed tuples avoids the wasted work associated with examining all tuples and ignoring those in which the user has no interest. The rarer a selected subpopulation, the more wasted work the Tail Index will save.

Interestingly, using a Tail Index to answer a query that involves multiple selected values (e.g., selecting both ‘Pre-K’ and ‘PhD’ in Step D of Figure 1.2) is not as straightforward as in a traditional index because of the randomness property that our AQP formulation requires. For example, if a user has selected a subset $\Omega_{Y'}$ of values from Ω_Y, sequentially scanning each list of tuples in full would destroy the randomness, potentially resulting in biased estimates. For example, suppose the user had selected both ‘Pre-K’ and ‘PhD’ to filter *salary* in Step D of Figure 1.2. By scanning each of the subpopulations sequentially (i.e., first scanning all of the ‘Pre-K’ tuples and then all of the ‘PhD’ tuples) in order to compute the approximation, the randomness requirement for our AQP formulation is destroyed.

For this reason, we scan each of the buckets in the index proportionally to its frequency to ensure randomness. To determine the list from which to draw the next tuple, we first compute the weight of each attribute value y' in the set of selected values $\Omega_{Y'}$ as:

$$weight(y') = \hat{\theta}_{y'} \sum_{y' \in \Omega_{Y'}} \frac{1}{\hat{\theta}_{y'}} \tag{3.10}$$

By normalizing the weights of all selected values, each value has a likelihood proportional to its selectivity. Then, the Query Engine can draw from each list based on these weights, ensuring that the samples used to compute the approximation are unbiased.

Finally, for queries over both common and rare subpopulations (e.g., selecting both 'HS' and 'PhD' in Step D of Figure 1.2), one bucket in the index will point to the base data while the other bucket will point to the list of rare tuples. In this case, we can simply scan the base data and ignore the Tail Index, since the error of the approximation will be dominated by the common value, and the rare value will be observed in the base data at the same rate (relative to the common value) that it would be sampled from the index.

3.4.3 Indexing Continuous Attributes

As previously mentioned, we can use binning to treat continuous attributes like nominal attributes. However, the user may often be interested in *zooming* in on the distribution within a specific subrange of a continuous attribute, which is a special case of selection where the bounds of a continuous attribute predicate are incrementally narrowed. For example, consider a visualization of the *age* attribute with bins on the decade scale (e.g., 20, 30, 40) where the user has filtered the range between 20 and 40 years old.

Building a complete index (e.g., a B-tree) is both impractical and unnecessary, since users visualize a high-level, aggregated view of the data and do not examine individual tuples. Thus, the simplest way to index a continuous attribute is to treat it as a nominal attribute, where each bin (i.e., predefined subrange) is equivalent to an attribute value. Depending on rarity, bins in the index store either a pointer to the Sample Store or to indexed tuples from that bin's range, similar to a Tail Index for a nominal attribute. Then, for selections over a set of bins, we can use the techniques described in the previous section for using an index.

However, unlike nominal attributes, zooming presents a new challenge for building Tail Indexes: suppose a common bin contains one or more rare sub-bins at the next zoom level. Then, if the user zooms into one of these sub-bins, we may be unprepared to provide a low-error approximation within the interactivity threshold because the parent bin is not rare and therefore will not be indexed. To try to mitigate this problem, we can transparently divide each bin into sub-bins (e.g., partition *age* into bins of one year instead of ten years), such that if the user performs a zoom action on a rare sub-bin, the system can draw tuples from the index. Since we are always one step ahead of the user, we can use the time between user interactions to continue to compute more fine-grained zoom levels in the event that the user continues zooming on a particular subrange.

3.5 Evaluation

We evaluated our techniques using real-world datasets and workloads that were derived from a past user study [47] in order to show that they can significantly improve interactivity and return higher quality answers faster than alternative approaches. First, we compare our prototype IDEA implementation to standard online aggregation, as well as a state-of-the-art column store DBMS that represents the most optimal blocking solution. Then, we show how our techniques perform when scaling different parameters from the benchmark. All experiments were conducted on a single server with an Intel E5-2660 CPU (2.2GHz, 10 cores, 25MB cache) and 256GB RAM.

3.5.1 Setup

We compare our IDEA prototype against a baseline of standard online aggregation (implemented in our prototype), as well as a column store DBMS (MonetDB 5) using three data exploration sessions that were derived from traces taken from a past user study [47]. Although MonetDB does not compute approximate answers, we wanted to show how the presented AQP techniques compare to a state-of-the-art system based on the traditional blocking query execution paradigm.

To construct each exploration session, we identified the most common sequences that users performed and then synthesized them into 16 distinct queries for each of the three datasets [89] (Census, Abalone, Wine). These exploration sessions model how a user would analyze a new dataset in order to uncover new insights, typically starting out with broader queries that are iteratively refined until arriving at a more specific conclusion. For example, the goal of the Census session is to determine which factors (e.g., *sex*, *education*) influence whether an individual falls into the 'High' or 'Low' annual *salary* bracket. The Abalone session explores data about a type of sea snail to identify which characteristics (e.g., *weight*) are predictive of *age*, which is generally difficult to determine. Finally, the Wine session examines the relationship between chemical properties (e.g., *so2*) and a wine's quality *score*.

To standardize all of the exploration sessions, we scaled each of the datasets to $500M$ tuples while preserving the original distributions of the attribute values. We additionally parse and load all datasets into memory for all systems before executing the benchmarks.

Since both online aggregation and our IDEA prototype can provide approximate results, we set the confidence level for query results to 3.5σ (i.e., the expected deviation of our approximation from the true result is less than 0.05%) before moving on to the query for producing the next visualization in the exploration session. Based on the interaction logs from our user study, we wait one second after achieving a sufficiently low result error before issuing the next query in order to model the time taken by the user to interpret the displayed results and decide the next action to perform (i.e., "think time"). Finally, we use 50 threads per query operation, which maximizes the runtime performance on the given hardware.

In Section 3.5.3, we vary each of these parameters (i.e., data size, error threshold, "think time", and amount of parallelism) to show how our techniques scale compared to online aggregation.

#1	`sex`
#2	`education`
#3	`education WHERE sex='Female'`
#4	`education WHERE sex='Male'`
#5	`sex, education`
#6	`sex WHERE education='PhD'`
#7	`salary`
#8	`salary WHERE education='PhD'`
#9	`sex, salary`
#10	`salary WHERE sex='Female'`
#11	`salary`
#12	`salary WHERE sex='Female'`
#13	`salary WHERE sex<>'Female'`
#14	`salary WHERE sex='Female' AND education='PhD',` `salary WHERE sex<>'Female' AND education='PhD'`
#15	`age`
#16	`salary WHERE 20<=age<40 AND sex='Female' AND education='PhD',` `salary WHERE 20<=age<40 AND sex<>'Female' AND education='PhD'`

Figure 3.2: Census Exploration Session Definition

3.5.2 Overall Performance

Figure 3.5 shows the time (in seconds) required to return an answer at or above the specified confidence interval in (1) MonetDB, (2) Online Aggregation, and (3) our IDEA prototype for every step in each of the simulated exploration sessions. Again, as a system with a blocking query execution model, MonetDB must wait until computing the exact query result (i.e., returning an answer with 100% confidence).

Each step number in the session corresponds to an action that the user performs in the visual interface, and Figure 3.2, Figure 3.3, and 3.4 show the corresponding queries executed to produce each visualization. Since all queries compute the count grouped by an attribute, we show only the group-by attribute followed by any selection predicates. For example, in the Census exploration session, the simulated user first visualizes both *sex* (#1) and *education* (#2), then views the distribution of *education* for only the 'Female' subpopulation (#3). Sometimes, an action in the frontend (e.g., removing a link) will issue multiple concurrent queries, in which case we delimit individual queries with a comma (e.g., #5 and #16 in Census). In these cases, we report the runtime after the results for both queries reach the acceptable confidence level, since the simulated user cannot proceed without low-error approximations for all visualizations on a given step. Since the main goal of our work is to provide a high-quality result within the interactivity threshold (i.e., $500ms$), each cell in Figure 3.5 is color-coded to indicate how close the runtime is to the threshold, ranging from green (i.e., significantly below the threshold) to red (i.e., above the threshold).

As shown in Figure 3.5, our IDEA prototype can return high-quality approximations within the interactivity threshold in many cases where both MonetDB and online aggregation cannot. In some cases, however, MonetDB can return an exact result faster than online aggregation can return an approximate answer with an acceptable error, due to a number of optimizations (e.g., sorting,

#1	`age`
#2	`weight`
#3	`weight WHERE age>=16`
#4	`age`
#5	`weight`
#6	`weight WHERE age<8`
#7	`sex`
#8	`sex WHERE age>=16`
#9	`sex`
#10	`sex WHERE age<8`
#11	`weight, sex`
#12	`age WHERE weight>=2`
#13	`age WHERE weight>=2 AND sex='I'`
#14	`age WHERE weight>=2 AND sex<>'I'`
#15	`age WHERE weight<0.4`
#16	`age WHERE weight<0.4 AND sex='I'`

Figure 3.3: Abalone Exploration Session Definition

compression) performed at data load time. Unfortunately, in interactive data exploration settings, the extensive preprocessing necessary to perform these types of optimizations often represents a prohibitive burden for the user.

Although our techniques cannot always guarantee a result with acceptable error in less than 500ms (e.g., #16 in Census, #2 and #3 in Abalone), our IDEA prototype returns high-quality approximate results as interactively as possible. Our approach is slightly slower than online aggregation for some queries that do not benefit from any of our proposed techniques because of the minor additional overhead associated with result caching and index construction. Note, however, that this overhead is small enough that it never causes our techniques to exceed the interactivity threshold where online aggregation does not already.

We now discuss the performance of our IDEA prototype to the online aggregation baseline in more detail for each of the previously described exploration sessions.

Census

After viewing the distribution of *education* for only the 'Female' subpopulation (#3), the simulated user changes the selection to examine *education* for the 'Male' subpopulation instead (#4). In this case, our prototype rewrites the query using the Law of Total Probability to reuse previously computed results and return a result without having to scan any data from the Sample Store. Similarly, when the simulated user reverses the direction of the link between *sex* and *education* to view the distribution of *sex* for only individuals with a 'PhD' (#6), our prototype uses Bayes' theorem to rewrite the query to return an answer almost immediately. When the simulated user issues two queries to compare the salary for male and female doctorates (#14), we can first execute the much easier (i.e., less selective) query over the non-'Female' subpopulation and use the Inclusion-Exclusion Principal to rewrite the query in order to compute an approximation of *salary* for only the female

#1	`score`
#2	`abv`
#3	`abv WHERE score>=8`
#4	`score`
#5	`abv`
#6	`abv WHERE score<8`
#7	`so2`
#8	`so2 WHERE score>=8`
#9	`so2`
#10	`so2 WHERE score<8`
#11	`so2, abv`
#12	`score`
#13	`score WHERE abv>=13`
#14	`score WHERE 100<=so2<200 AND abv>=13`
#15	`score WHERE abv>=13`
#16	`score WHERE 100<=so2<200`

Figure 3.4: Wine Exploration Session Definition

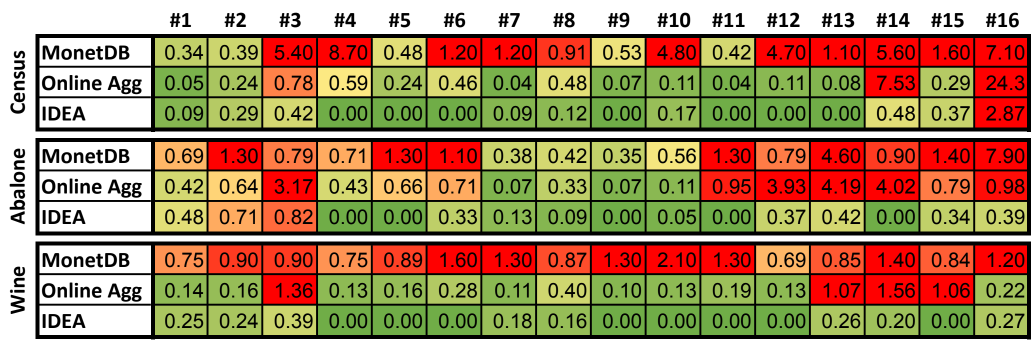
doctorates. To execute this easier query, we use the multidimensional Tail Index created over the *education* and *sex* attributes. Finally, when the user drills down into especially rare subpopulations (#14 and #16), our Tail Indexes allow our IDEA prototype to compute a low-error approximation significantly faster than online aggregation.

Abalone

The simulated user first views the distribution of *age* (#1) and *weight* (#2) individually, then compares the distributions of *weight* for older (#3) and younger (#6) abalones. Since both of these selected subpopulations are relatively rare, our Tail Indexes help us to achieve a low-error result significantly faster than online aggregation. Our Tail Indexes provide similar speedups in steps #13 and #15, yet again achieving interactivity where online aggregation cannot. Finally, when the simulated user compares *age* for infant abalones versus adults (#14), we can leverage the Law of Total Probability to reuse previously computed answers from the Result Cache in order to return an approximation instantly.

Wine

As part of this exploration path, the simulated user compares the alcohol content *abv* of high quality wines (#3) to that of lesser quality wines (#6). Although online aggregation cannot provide a sufficiently low-error approximation for the query over only the high quality wines (a rare subpopulation), our Tail Indexes allow us to compute a high-quality approximation within the interactivity threshold. For the query over the lesser quality wines, we can use the Law of Total probability to reuse the results from #2 and #3 to provide a low-error approximation instantly. Finally, by caching the results of previously executed queries, we can immediately return the results for the queries that the simulated user has already issued in the past (#4, #5, #9, #11, #12, #15).



		#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16
Census	MonetDB	0.34	0.39	5.40	8.70	0.48	1.20	1.20	0.91	0.53	4.80	0.42	4.70	1.10	5.60	1.60	7.10
	Online Agg	0.05	0.24	0.78	0.59	0.24	0.46	0.04	0.48	0.07	0.11	0.04	0.11	0.08	7.53	0.29	24.3
	IDEA	0.09	0.29	0.42	0.00	0.00	0.00	0.09	0.12	0.00	0.17	0.00	0.00	0.00	0.48	0.37	2.87
Abalone	MonetDB	0.69	1.30	0.79	0.71	1.30	1.10	0.38	0.42	0.35	0.56	1.30	0.79	4.60	0.90	1.40	7.90
	Online Agg	0.42	0.64	3.17	0.43	0.66	0.71	0.07	0.33	0.07	0.11	0.95	3.93	4.19	4.02	0.79	0.98
	IDEA	0.48	0.71	0.82	0.00	0.00	0.33	0.13	0.09	0.00	0.05	0.00	0.37	0.42	0.00	0.34	0.39
Wine	MonetDB	0.75	0.90	0.90	0.75	0.89	1.60	1.30	0.87	1.30	2.10	1.30	0.69	0.85	1.40	0.84	1.20
	Online Agg	0.14	0.16	1.36	0.13	0.16	0.28	0.11	0.40	0.10	0.13	0.19	0.13	1.07	1.56	1.06	0.22
	IDEA	0.25	0.24	0.39	0.00	0.00	0.00	0.18	0.16	0.00	0.00	0.00	0.00	0.26	0.20	0.00	0.27

Figure 3.5: Benchmark Runtimes (s)

3.5.3 Scalability

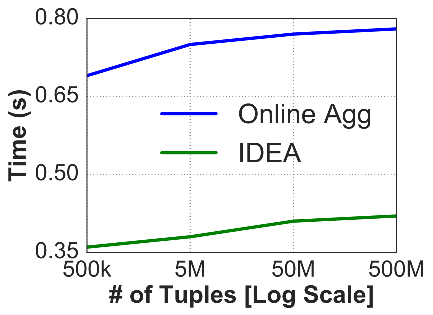
To better understand the limits of our techniques, we conducted scalability experiments in which we varied (1) data size, (2) confidence level, (3) simulated "think time", and (4) amount of parallelism. For all scalability experiments, we show the results for steps #3 and #16 from the Census workload, which represent an early and late stage in the exploration session, respectively. As such, step #3 has much less potential for result reuse and indexing than #16, which has access to cached results from past queries and indexes that have already been created.

Data Size

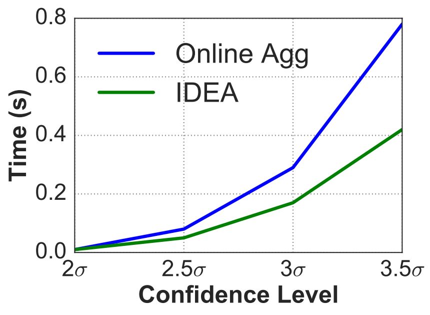
Figures 3.6a and 3.6e show the runtimes for step #3 and #16, respectively, for various data sizes with a fixed confidence level of 3.5σ, "think time" of one second, and parallelism of 50 threads. As shown, for both #3 and #16, the size of the data has little noticeable impact on performance, since both online aggregation and our IDEA prototype scan only a fraction of the tuples (independent of data size) in order to provide a low-error approximation. The slight constant increase in the runtime for both approaches as data size increases is primarily attributable to the overhead associated with managing larger datasets in memory (e.g., object allocation, garbage collection, CPU caching effects), which increase with the amount of data.

Error

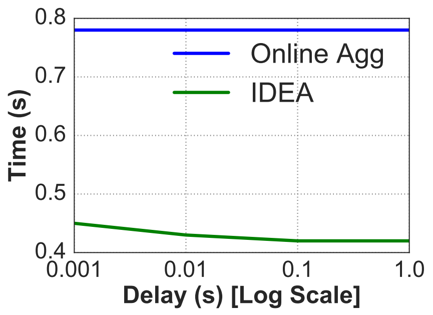
To show how both online aggregation and our proposed AQP techniques perform for different amounts of acceptable error, we show the runtime for both online aggregation and our IDEA prototype to achieve varying confidence levels for step #3 and #16 in the Census exploration session using $500M$ tuples, one second of "think time," and 50 threads per query operation. As shown in Figures 3.6b and 3.6f, our approach is able to provide an approximate answer with the same amount of error as online aggregation in less time. In some cases, such as with a confidence of 3.5σ, our approach outperforms online aggregation by an order of magnitude due to our previously described Tail Indexes.



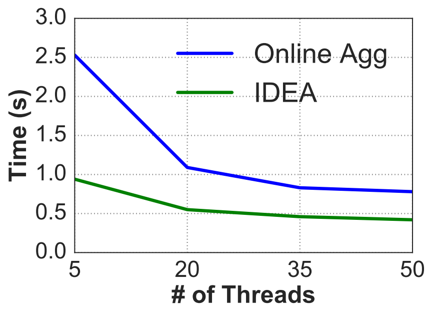
(a) Data Size (#3)



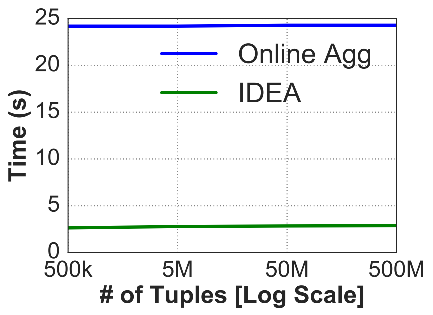
(b) Error (#3)



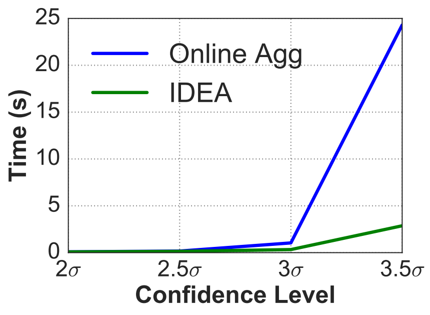
(c) Think Time (#3)



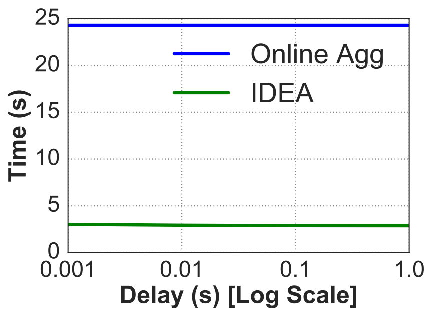
(d) Parallelism (#3)



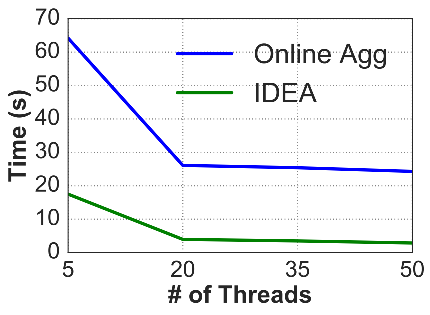
(e) Data Size (#16)



(f) Error (#16)



(g) Think Time (#16)



(h) Parallelism (#16)

Figure 3.6: Scalability Microbenchmarks

Think Time

Since the amount of time between user interactions influences the number of tuples that can be indexed, Figures 3.6c and 3.6g show the time necessary for both online aggregation and our IDEA prototype to compute an approximate result with a confidence level of 3.5σ using $500M$ tuples and 50 threads for each query operation. As shown, the performance of online aggregation is only very slightly affected by "think time". On the other hand, as shown, the more "think time" that our IDEA prototype has to begin preparing for a follow up query, the faster it can compute a low-error visualization.

Parallelism

As previously described, each query operation (e.g., index creation, aggregate approximation) uses multiple threads in order to best take advantage of modern multi-core systems. To show how the amount of parallelism affects system performance, Figures 3.6d and 3.6h show the runtime to compute an approximate result with a confidence level of 3.5σ for both online aggregation and our IDEA prototype using $500M$ tuples, and one second of "think time" for a varying number of threads. As shown, as the number of threads increases, the system can take better advantage of the underlying hardware, resulting in lower query latencies. However, for more than 50 threads, performance begins to degrade due to thrashing and increased contention on shared data structures.

3.6 Related Work

The techniques presented in this work have overlap in three main areas: (1) approximate query processing, (2) result reuse, and (3) indexing.

3.6.1 Approximate Query Processing

In general, AQP techniques fall into two main categories: (1) biased sampling [2] and (2) online aggregation [59]. Systems that use biased sampling (e.g., AQUA [3], BlinkDB [4], DICE [76]) typically require extensive preprocessing or foreknowledge about the expected workload, which goes against the ad hoc nature of interactive data exploration. Similarly, systems that perform online aggregation (e.g., CONTROL [58], DBO [72], HOP [28], FluoDB [151]) are unable to give good approximations in the tails of the distribution, which typically contain the most valuable insights. Our approach, on the other hand, leverages unique properties of interactive data exploration in order to provide low-error approximate results without any preprocessing or a priori knowledge of the workload.

Similar to our approach, Verdict [114] uses the results of past queries to improve approximations for future queries in a process called "Database Learning." However, Verdict requires upfront offline parameter learning, as well as a sufficient number of training queries in order to begin seeing large benefits.

3.6.2 Result Reuse

In order to better support user sessions in DBMSs, various techniques have been developed to reuse results [137, 69, 104, 39]. These techniques, however, do not consider reuse in the context of (partial) query results with associated error. Specifically, our proposed AQP formulation allows us to formally reason about error propagation to quantify result uncertainty for the user, as well as for making query optimization decisions.

3.6.3 Indexing

Database cracking [62] is a self-organizing technique that physically reorders tuples in order to more efficiently support selection queries without needing to store secondary indexes. However, database cracking would destroy the randomness of the underlying data, a property that our AQP techniques rely upon in order to ensure the correctness of approximate results.

Partial indexes [134] are built online for only the subset of data of interest to the user. Our Tail Index takes these ideas a step further and attempts to keep track of only rare subpopulations in order to minimize overhead while still meeting interactivity requirements. Moreover, our techniques necessitate careful consideration of how to draw tuples from Tail Indexes in order to preserve randomness.

Chapter 4

Time Series Data Exploration

After extensively studying common interactive data exploration workloads and use cases, the exploration of time-dependent data emerged as an important use case. Although live monitoring applications (e.g., dashboards) and stream processing may help identify potential issues (e.g., server failure, device offline, malicious attack), interactive data exploration for time series data can again offer huge benefits and help an analyst discover more meaningful insights. For example, using historical machine generated data, an analyst can determine when rooms are likely to be unoccupied or when a server is likely to experience a hard drive failure. Additionally, unlike traditional stream processing, this type of historical data analysis is free-form and exploratory in nature; an analyst continuously issues new queries in order to uncover previously unknown insights from the data.

Since visual tools like Vizdom allow users to derive useful insights from data, it is important that a visual data exploration stack is able to seamlessly integrate with a backend system that is designed to efficiently handle large machine-generated datasets. Moreover, with the massive amount of time series data that is generated daily, along with the fact that data sizes are expected to grow by as much as 40% annually [61], a huge number of open source projects, research prototypes, and commercial systems that specifically cater to machine-generated data use cases [6, 31, 147, 116, 78, 67, 97, 110, 122, 139] have been proposed. Therefore, one of the key questions we sought out to answer as part of this work was to determine if it was possible to leverage these highly specialized systems in the context of our interactive data exploration stack described in Chapter 2.

However, with the increasing number of available solutions for storing and analyzing machine-generated data comes the urgent need for a framework to systematically evaluate competing approaches. While existing benchmarks address either streaming aspects (e.g., Linear Road [8], StreamBench [95], RIoTBench [130]) or time series data mining (e.g., CATS [87]), benchmarks for analytical queries over time series data have received almost no attention, despite the importance of these workloads in the data-driven decision making process. The absence of a standardized benchmark has created a situation not unlike the Wild West, with companies publishing a flood of blog posts claiming that their own system is “the best” [65, 64, 66, 140, 138, 30, 38].

Therefore, one of the main goals of this work is to sort out these contradictory results and identify—in an unbiased fashion—the most beneficial techniques. While other studies have previously considered either raw CRUD performance [15, 148] or simple aggregation queries [9, 92] in the machine-generated data domain, we could find no existing in-depth, use-case-driven benchmarking study that evaluates how systems might perform in the wild. Consequently, as a first step, we needed to decide on a fair way to evaluate the systems, which naturally prompted the question: could we simply use an existing analytical benchmark like TPC-H [143], or are workloads for machine-generated data fundamentally different?

At first glance, existing OLAP benchmarks may seem sufficient, since they generally include some time dimension (e.g., the many date fields in TPC-H). However, after analyzing real-world machine-generated data applications, we noticed that almost all applications universally required "constraints and group-by's at multiple temporal hierarchies" [99] (e.g., timestamp extraction/manipulation, aggregation grouped by derived time values), whereas traditional OLAP benchmarks merely use the time dimension as just another filter condition. Moreover, even though many applications use the relational model in practice rather than a specialized temporal model [128], the sequential nature of time series data fundamentally influences design decisions related to storage and access patterns [31, 139], and this key feature is absent from traditional OLAP benchmarks.

Even more importantly, though, the subtle patterns inherent in machine-generated data (e.g., fewer hits to a website on holidays, more activity in an office building during the working hours) are crucial for creating a realistic benchmark [79]. Without these patterns, which are almost impossible to synthetically replicate at scale, we cannot reason about how systems will *actually* perform in practice. For example, many specialized time series compression algorithms [116, 5, 125] and new indexing techniques [84, 48] heavily leverage patterns to achieve huge performance improvements, and applying these techniques to synthetic data that does not accurately capture real-world patterns may yield false conclusions.

To address these shortcomings of existing OLAP benchmarks, we present MgBench, a new analytical benchmark specifically for machine-generated time series data. MgBench is the first benchmark for machine-generated data that uses multiple large real-world datasets and realistic queries derived from in-depth interviews with real stakeholders. Such a benchmark is a substantial undertaking that required gathering large and interesting datasets, as well as synthesizing representative queries that capture the core components of these workloads.

MgBench contains three representative case studies, each of which tests a different aspect of machine-generated data analysis. To allow others to replicate the benchmark in new environments or evaluate additional systems, we make the datasets, queries, and scaling tools for each of the case studies publicly available [98].

In addition to the benchmark specification, we also present an evaluation of five popular open source systems to help us understand the best system to use for a particular application and determine why systems perform the way that they do. More specifically, in Section 4.5, we answer two important questions: (1) what is the best existing system for a given workload, and (2) what should designers for the next generation of systems focus on?

Case Study	Attributes			Size (SF1)		Queries	
	Timestamps	*String*	*Numeric*	*Tuples*	*Raw Text*	*Key Challenge*	*Tested Features*
Performance Monitor	*Regular*	2	18	52M	5.2GB	*Aggregation Speed*	`COUNT`, `SUM`, `AVG`, `MIN`, `MAX` Wide Tuple Storage `NULL` Suppression
Web Server	*Irregular*	2	2	705M	58GB	*String Processing*	String Functions (`LENGTH`, `REPLACE`) `LIKE` Comparison Long String Storage
IoT Events	*Irregular*	5	1	26M	1.5GB	*Query Optimization*	Time Functions (`CAST`, `EXTRACT`) Nested/Correlated Subqueries Advanced SQL (`CASE`, `HAVING`)

Table 4.1: Summary of Case Studies

In the next chapter, we present a new index structure that uses our insights derived from these real-world case studies in order to improve the interactivity of visual time series data analysis.

4.1 The Benchmark

In order to develop MgBench, we reached out to dozens of industry practitioners to learn more about their specific use cases. Our conversations with these stakeholders helped us to develop a representative benchmark that includes three diverse workloads, each stressing a different type of challenge, that we used to evaluate the tradeoffs associated with different approaches. This study focuses primarily on ad hoc analytical queries, which we found represented a critical component of real-world use cases, as well as the most significant amount of optimization effort.

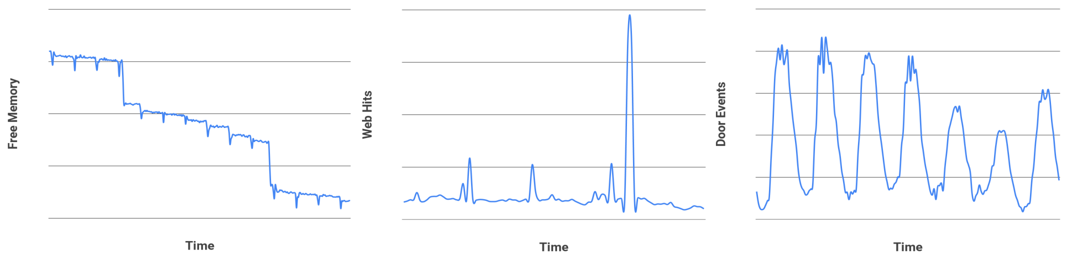
As part of this work, we make the benchmark publicly available, including a full specification that contains the datasets, schema definitions, and SQL queries, in order to allow others to test new systems. In addition to the data and the queries, we include a tool to scale the data for each of the case studies. With this tool, the data can be scaled up (e.g., by increasing the frequency of IoT sensor readings) while preserving the patterns that exist in the base data.

This section describes MgBench at a high level. Section 4.3 presents a much more detailed discussion of each query for each of the use cases, including the business question that each query answers as well as how each system performs.

4.1.1 Design Principles

Unlike traditional analytical benchmarks (e.g., TPC-H) where the time dimension is just another attribute upon which the data is filtered, MgBench incorporates time as a key component. For example, in addition to filtering the data to specific time ranges, some queries ask for specific times to be returned in the select clause, while others perform aggregations grouped by extracted time values (e.g., hourly, daily).

Each query represents a realistic business question raised in our interviews with stakeholders, and each case study is designed to stress benchmarked systems in different dimensions. For example, the *Performance Monitor* case study tests system features important for raw aggregation performance, whereas the *Web Server* case study focuses heavily on string processing.



(a) Performance Monitor **(b) Web Server** **(c) IoT Events**

Figure 4.1: Real-World Patterns in Case Study Datasets

Even more importantly, since machine-generated data naturally contains patterns, MgBench uses real data that captures these subtle patterns (e.g., more motion activity during business hours, higher web traffic on certain days), as shown in Figure 4.1. We are also especially careful to preserve these patterns when scaling the different datasets, and we discuss in detail how we perform this scaling for each use case.

In terms of schema design, one common trend we observed in real-world applications was the use of a single large, semi-denormalized table to store log data. This anecdotal evidence is further reinforced by the fact that many TSDBs have no built-in support for join operations. Therefore, each schema in MgBench conforms to this wide-table layout pattern.

4.1.2 Case Studies

MgBench is comprised of three case studies, which are summarized in Table 4.1. Each case study consists of a real-world dataset and six corresponding queries that represent business questions over different time scales, including two short-range (i.e., a few hours up to one week), two medium-range (i.e., one week up to a few months), and two long-range (i.e., a few months up to one full year) queries. We believe that, due to the centrality of the time dimension for machine-generated data, having queries with a range of time-based filters is crucial and serves to highlight some of the design decisions made by certain systems.

In the following, we provide a high-level summary of each of the three case studies.

Performance Monitor

The first case study, Performance Monitor, contains data and business questions that help IT administrators diagnose problems that arise in a large IT department.

The data for this case study comes from logs for all machines (e.g., workstations, servers, grid nodes, virtual machines) managed by a major US university. The measurements contain 18 values, including CPU utilization, amount of available memory, and network I/O for a given machine at a given timestamp. The dataset contains measurements for 695 machines for one year, resulting in about 55 million tuples with a raw text size of 5.2GB. Each row contains 21 attributes (one timestamp, two strings, and 18 floating point values).

Patterns: These logs contain readings that are recorded at set time intervals (e.g., every 5 minutes) with slight variability (e.g., plus/minus a few seconds). Some machines record metrics at finer granularities than others (e.g., web servers are more critical and therefore report more frequently than personal workstations). Again, although the reading time interval is generally constant with slight variations, other metrics (e.g., CPU utilization, amount of free memory) contain distinct trends that align with usage. For example, as demonstrated in Figure 4.1a, the amount of available free memory on a personal workstation degrades in a clear step-wise pattern over time due to a memory leak.

Scaling: Scaling up the data for this use case involves simulating finer-grained logging intervals (i.e., each machine reports metrics more frequently). Therefore, when scaling the base data by scale factor n, we need to create and append $n - 1$ new readings between existing log records for each machine. Since machines report in (mostly) regular intervals, we can insert $n - 1$ new readings between timestamps t_i, t_j at every $(t_j - t_i)/n$ intervals, plus/minus some normally distributed random noise to simulate the slight variation in reporting times. To create the new metric values, we first compute the mean μ and variance σ^2 of the percent change for a given metric. Then, to create a new value at t_{i+1} we add a value drawn from the distribution $\mathcal{N}(\mu, \sigma^2)$ to the value of the metric at t_i.

Queries: Since this case study contains many metrics for a given timestamp and relatively simple queries, this workload primarily tests raw aggregation and scan performance. Additionally, since individual records are relatively wide (21 attributes), this case study helps test the system's storage format (e.g., row vs column). Unlike the other case studies, this case study does not include any string processing operations, since all of the metrics are floating point numbers, although efficient null value handling is important to handle missing values (e.g., machine shutdown/server crash, lost log message).

Web Server

The second case study, Web Server, involves analyzing log entries from the web server of a major US university's Computer Science Department.

The data contains requests to the website for a full year and has approximately 689 million tuples, which amounts to a raw text size of 54GB. Each entry represents a web request and contains information including the time of the request, client IP address, request string (i.e., URL), status code returned by the server, and size of the requested resource. The resulting schema has five attributes (one timestamp, two strings, and two integers).

Patterns: Unlike Performance Monitor, which has a regular interval between timestamps created as part of an automated process, web requests arrive at random intervals, as they are human-dependent. Yet, given the huge volume of requests to the department's website, the timestamp distribution is dense; that is, for the smallest time granularity (one second), there exists at least one log record in each bucket. As shown in Figure 4.1b, this dataset contains many interesting patterns that map to real-world events. For example, the server receives noticeably less traffic during holidays (e.g., Christmas, July 4th) whereas it receives increased traffic to course web pages during exam periods. Interestingly, there are also several huge spikes in traffic that correspond to anomalous events, like denial of service (DoS) attacks.

Scaling: To scale the data for this use case involves generating more web requests that follow the general date and URL (i.e., popular requested resource) trends in the data. In order to capture the patterns in the time dimension (e.g., weekends, holidays), we first compute the PDF of requests per day of the year. Then, for each day, we compute the daily (`request`,`status_code`,`object_size`) PDF. To generate a new tuple, we make a weighted selection from the day PDF, and then select a weighted choice from the triple PDF to create a request value. Finally, values for client IP addresses are randomly selected from the space of valid IP addresses, since these values are sparse (i.e., drawn from a space of about 4 billion possible values).

Queries: The associated queries, which range from analyzing user behavior to finding malicious attacks, stress different parts of the systems than the previous Performance Monitor case study. In terms of data storage, it is very important that systems have efficient techniques for storing the long resource string field (e.g., `/courses/cs123/home.html`). Moreover, since most queries in this case study involve the resource string, efficient string processing functions are also particularly important, including string search, comparison, and manipulation functions, especially when using aggressive string compression techniques.

IoT Events

The final case study, *IoT Events*, examines data created from over 100 IoT sensors deployed throughout the Computer Science building of a major US University.

These sensors (e.g., door sensors, motion sensors, temperature sensors, and power monitors) record data about building use, gathered over a period of one year. A single row represents an action that occurred or measurement and contains information such as the location, type of reading (e.g., door opened, motion started), and reading value for measurement readings (e.g., temperature, power usage). The collected data amounts to 26 million tuples or 1.5GB of raw text. Each row in the dataset has one timestamp, five strings, and one floating point value.

Patterns: Similar to Web Server, timestamp intervals in this case study are random, since readings are driven by human activity. Since the installed IoT devices monitor building activity and usage, the data for this case study contains many unique patterns. Unsurprisingly, as shown in Figure 4.1c, doors in the building are opened/closed more often during the working hours and weekdays compared to evening hours and weekends. The dataset also contains longer-term usage patterns, including increased activity during exam periods and decreased activity during academic breaks. Additionally, natural patterns like temperature variations during the day and seasonal temperature trends are prevalent.

Scaling: Scaling up the data for this use case requires adding new events that capture the patterns that occur for each (1) event type, (2) event time, and (3) possible value (i.e., for temperature and power). First, we select from a weighted distribution of event types (door, motion, temperature, power) since some types of sensors report more often than others. For example, power readings are generated by fluctuations in electrical devices, which occur much more frequently than devices report temperature changes in the building. Then, for that type of reading, we select a device ID from the PDF of all device IDs for that type, since certain devices are more active than others (e.g., the front door of the building vs the top floor balcony), as well as a reading time (day, hour) from a PDF specific to the device type. Values for device location and floor are simply derived from device ID. Finally, to get an event reading value, we interpolate between the two nearest adjacent event reading values.

Queries: The queries for this data come from interviewing university employees from facilities and maintenance. Compared to the previous two case studies, the associated data contains more diverse measurements from different types of sensors and, in general, the associated queries are much more complex, requiring more advanced SQL features (e.g., `CASE` and `HAVING` clauses, correlated subqueries). Therefore, this case study in particular attempts to stress each system's query optimizer and complex plan execution strategies.

4.1.3 Ground Rules

In order to make future uses of the benchmark consistent, we impose some ground rules on the benchmark for reporting "official" results.

As a general rule, systems are forbidden from pre-processing the data in a way that leverages explicit knowledge of the workload (e.g., extracting path depth from the `resource_path` field to speed up Web Server Q1), since the queries are meant to be ad hoc analytical queries. The tested systems are additionally only allowed to index the time dimension, similar to the secondary index restrictions imposed by TPC-H.

In this paper, we consider each base dataset size (shown in Table 4.1) as $1\times$ scale factor (SF1). Any future use of this benchmark should clearly report the scale factor as a multiple of this baseline.

4.2 The Systems

As previously mentioned, one of our goals was to identify the implications of different design decisions on machine-generated data workloads. Therefore, we evaluate a wide variety of systems using our benchmark in order to methodically study these different design decisions.

Although there are a huge number of systems used both in practice and from academia, we could only select a limited number to include in our study. We tried to pick a well-rounded group of systems in order to understand the key tradeoffs in different design decisions for machine-generated data use cases. Moreover, we were unable to include several relevant commercial systems (e.g., kdb [78]) due to non-permissive licenses prohibiting benchmark result publication. However, we hope that MgBench, including the real-world dataset, pattern-aware scaling tool, and realistic business queries, will be valuable for use by third parties in the future, either to guide the design

Type	Name	Version	Storage	License
DBMS	PostgreSQL	10.4	Row	GPLv2
	MonetDB	11.29.7	Column	MonetDB License
Analytics	SparkSQL	2.1.3	Column	Apache 2.0
TSDB	CrateDB	3.0.3	Column	Apache 2.0
	TimescaleDB	0.10.1	Row	Apache 2.0

Table 4.2: List of Tested Systems

of their own system or make more informed decisions about potential system selection based on the similarity of their own applications to the benchmark use cases.

Additionally, we placed certain common requirements on all systems for inclusion in the study. First, we required that all systems have a largely complete SQL syntax since this is what we found to be used most often after talking to industry practitioners. Thus, a variety of NoSQL systems (e.g., MongoDB [103], Dynamo [37], Couchbase [29]) and time series systems with only a CRUD API (e.g., OpenTSDB [110], Atlas [14], Cube [35], Gorilla [116]) were excluded from the study. Furthermore, we required that all systems be able to handle the necessary data types (e.g., strings, null values) that were common in real world use cases we examined and were therefore unfortunately forced to exclude some systems (e.g., InfluxDB [67]). Although some of these restrictions could be worked around at the application level, it would generally be highly inefficient and we wanted to ensure an apples-to-apples comparison at the systems level in order to accurately capture the pros and cons of certain design decisions for future systems developers.

Using these constraints, we selected a variety of systems, summarized in Table 4.2, that represent very distinct approaches. In the following, we consider the strengths and weaknesses of each approach, as well as our rationale for choosing certain systems. We also provide a brief description of our expectations for the performance of each system before we conducted the study, as well as a summary of how each system actually performed (sometimes in surprising contrast to our expectations). The interested reader is referred to the next section, where we provide a detailed discussion of the setup, queries, and detailed results.

4.2.1 DBMSs

The first category of systems we evaluated were the traditional DBMSs. These systems utilize the relational model and provide strong guarantees for transaction execution (e.g., ACID). Despite these similarities, there is significant variability in the physical implementation of different DBMSs.

PostgreSQL

PostgreSQL [121] is a popular open-source DBMS that utilizes a row storage format and represents the traditional, disk-resident database design approach. Row stores are the traditional approach to managing relational data and, as a consequence of their design, will often need to bring in several unexamined attributes for many OLAP-style queries. Additionally, indexes can often have a large impact on query runtime, since they are unable to sort individual columns.

Tuning: PostgreSQL has a huge configuration file with hundreds number of tuning knobs. To aid in the tuning process, we used the PGTune [119] tool to generate a baseline configuration file tuned for OLAP with the specifications (e.g., number of CPUs, available memory) of the machines we used during the experimental evaluation. Then, we hand-tuned individual parameters (e.g., size of shared buffer, amount of parallelism) that we found to be particularly important to query performance.

Expectation: Row store DBMSs, including PostgreSQL, store all fields in a tuple contiguously. Therefore, for any query, the entire tuple must be brought into the CPU from main memory, even if only relatively few of the attributes are accessed. For this reason, we expect the runtime for queries that only access a few of the columns in a table to suffer. Conversely, queries that access most of a (or the entire) row, should be faster than column-oriented systems, since tuple reconstruction costs can be high in column store systems. Since most of the queries touch only a few of the attributes in the table, we expect row-oriented systems to be generally inferior.

Reality: As expected, PostgreSQL performs significantly worse on queries that touch only a few columns of wide tuples, which is most noticeable in the Performance Monitor use case. Surprisingly, however, the row-based storage layout was beneficial in several queries for the Web Server use case, and PostgreSQL actually outperformed all other systems in one query (Q5) due to the fact that it did not need to perform any decompression when filtering to specific resource strings. Overall, however, PostgreSQL underperformed most other systems.

MonetDB

Column stores have emerged in the past decade as the clear winner for analytics workloads, and so we also evaluate the performance of MonetDB [102], an in-memory column store. For OLAP-style queries, the performance gains therefore come from storing the data efficiently, in particular avoiding the need to access unused column values and utilizing heavy compression (e.g., dictionary encoding, run-length encoding, delta encoding).

Tuning: MonetDB is almost entirely self-configuring and therefore has no user-modifiable tuning parameters. However, MonetDB has several self-configuring aspects that tune the system based on the data and application, such as decisions about dictionary compression and data storage (e.g., database cracking [62], column imprints [132]).

Expectation: As previously mentioned, since column store DBMSs only need to bring in the attributes that the query touches, we expect MonetDB to do well on queries which access only a few columns. Additionally, since MonetDB can use compression to reduce the amount of data transferred to the CPU, we expect the performance of queries that work on highly compressible attributes (e.g., strings with only a few distinct values) to be very good. However, we expect that queries over attributes that require decompression will suffer, such as `LIKE` comparisons that require full string search.

Reality: MonetDB performs quite favorably, finishing in first place in many instances. Surprisingly, even though MonetDB does not treat time as a special dimension, it often outperforms the closest runner up by as much as $2 - 5\times$. As expected, queries involving string manipulation (Web Server Q1) or LIKE comparisons (Web Server Q1) were difficult for MonetDB, where it finished close to last place. Interestingly, we also found that MonetDB generally performs favorably for queries that involve a large number of group-by keys.

4.2.2 Analytics Systems

The second category of systems we tested are systems designed for performing complex analytics. Recently, these types of systems (e.g., Spark [150], Flink [77]) have become extremely popular among industry practitioners. As a representative of this category, we selected the most widely used option that fit our data model needs, SparkSQL [10].

SparkSQL

SparkSQL [10] is a popular Spark module for running SQL queries on top of underlying Spark RDDs. SparkSQL uses all of the functionality of Spark to execute queries in a distributed and efficient manner while also leveraging additional information (e.g., schema design) to further optimize query execution. SparkSQL maximizes scalability by parallelizing queries as much as possible.

Tuning: Although earlier versions of SparkSQL required extensive manual tuning to enable experimental optimizations and select the optimal configuration parameters, more recent versions have good parameters chosen by default. The two key parameters that still need to be set for Spark are (1) JVM memory limits and (2) available parallelism based on the number of CPU cores.

Expectation: Like MonetDB, SparkSQL individually compresses columns in order to minimize the amount of data transferred and stored. To this end, we expect it to perform somewhat similarly to MonetDB, with the caveat that it is less mature and not as specialized as a traditional OLAP-optimized DBMS. SparkSQL, though, is designed to be highly parallel in order to be able to scale to clusters of machines. This can be beneficial in some cases, and harmful in others, depending on the amount of overhead and extra steps required to parallelize part of a query plan. Therefore, we expect that these overheads will be noticeable on the smaller scale factors but that SparkSQL will scale well to the larger scale factors.

Reality: In general, SparkSQL performed much better than we expected across the board; as a fairly recent project, it stands up well compared to mature systems that have more than a decade of development behind them. For example, SparkSQL is competitive with MonetDB, a highly optimized, OLAP-focused DBMS, in the Performance Monitor use case (Section 4.1.2) and often outperforms MonetDB in the Web Server use case (Section 4.1.2), where string processing is paramount. As expected, SparkSQL generally has excellent scalability for the larger datasets and often performs best for queries that require a full table scan.

4.2.3 TSDBs

A time series database system (TSDB) is specialized to handle workloads with heavy focus on the time dimension. Some systems even have special domain specific functionality/primitives for more advanced time series manipulation (e.g., KDB has time series forecasting functionality for financial applications). Because of their central focus on time, these systems should in theory perform best in our benchmarks. In order to identify the advantages of different design decisions, we tested two different TSDBs.

CrateDB

CrateDB [31] is an open-source, highly scalable, column-oriented system designed specifically for managing large volumes of machine-generated data built from and on top of the popular Elasticsearch [43] and Lucene [96] projects. All tables in CrateDB are sharded and distributed if deployed on a cluster of machines, with optional shard replication for k-safe access. Each table shard is a Lucene index that is broken down into segments for finer-grained search capabilities. Like a traditional DBMS, CrateDB also provides ACID guarantees for queries, and also provides a mostly complete SQL subset for accessing data.

Tuning: Although CrateDB ships with sensible default configuration values, there were several tuning knobs which influenced performance. Most notably, to avoid the overhead associated with 32-bit vs 64-bit pointers in the JVM, the developers advised us to create multiple CrateDB instances, each of which has access to no more than 30GB RAM. This ensured that we utilized the entire amount of system memory while avoiding 64-bit pointers.

Expectation: Since CrateDB is specifically designed for managing time-dependant data, we expect that it will perform well across the board. Specifically, for queries that access small time ranges (e.g., yesterday), we expect it to be able to easily access the relevant tuples and not have to scan any unnecessary data. This design decision should give it an advantage over MonetDB, which is also a column store, but it does not treat the time dimension as a core part of its design. On the other hand, we expect CrateDB to have similar performance to MonetDB for queries with no time-based selection predicates since they make similar design decisions.

Reality: Overall, CrateDB performed very well in the Performance Monitor case study, finishing at or near the top of most queries across all scale factors. Surprisingly, though, CrateDB did not perform as well in the Web Server and IoT Events case studies, except in cases involving filtering to very specific time ranges. Contrary to our expectations, the treatment of time as a central component of the system design tended not to offer large performance improvements over MonetDB; we mostly observed little to no noticeable performance improvement, even in cases of extremely short range queries (e.g., Q1 of Performance Monitor asks for only the past twelve hours and is only a factor of 3x faster, see Section 4.3.1). Moreover, CrateDB often exhibits orders-of-magnitude worse performance compared to MonetDB for longer time range queries. Finally, despite its focus on scalability, CrateDB did not scale well in several instances, particularly the Web Server case study.

TimescaleDB

The last tested TSDB is TimescaleDB [139], which has a particularly unique design. Unlike other time series systems that are built from the ground up, TimescaleDB is actually built into PostgreSQL as a transparent plugin for optimizing time series data. More specifically, TimescaleDB organizes data into hypertables partitioned into disjoint chunks that are each individually indexed, enabling efficient search for small time intervals. With minimal changes, users can install TimescaleDB and run queries transparently via their existing PostgreSQL deployment.

Tuning: Since TimescaleDB builds on top of PostgreSQL, almost all of the same configuration options are used. Again, we used the PGTune tool as a baseline for generating our configuration file and further hand-tuned all of the parameters to OLAP for a machine with many cores and abundant main memory.

Expectation: Again, due to the shared codebase with PostgreSQL, we expect TimescaleDB to similarly suffer for queries that involve only a few attributes. However, since TimescaleDB natively incorporates the time dimension into its storage and organization of data as hypertables, we expect it to outperform PostgreSQL on queries that involve any time-based selection condition.

Reality: Surprisingly, TimescaleDB generally performs much better than expected. We assumed that TimescaleDB would inherit much of the baggage associated with PostgreSQL and therefore perform poorly for most analytics workloads, only achieving minor performance improvements over PostgreSQL in terms of the speed of time-based lookups. In fact, TimescaleDB often performs comparably to column store systems like MonetDB and CrateDB, especially over small time ranges. However, these advantages start to diminish at larger scale factors, as data filtering comprises a decreasing amount of total query runtime. Furthermore, similar to performance of CrateDB relative to MonetDB, TimescaleDB performs noticeably worse compared to PostgreSQL for queries over large time ranges despite sharing the same code base. This slowdown is due the overhead associated with its hypertable storage.

4.3 Detailed Results

In this section, we present a detailed description of the queries for each case study described in Section 4.1. Additionally, we provide performance results for each of systems described in Section 4.2 on the base dataset sizes, as well as scaled (raw) sizes of approximately $100GB$ and $200GB$.

In all cases, we report the average runtime of each system over ten runs once the system has been warmed up (i.e., the data is cached in memory). These detailed results are shown in Figures 4.2, 4.3, 4.4, where the color gradient in each row from green (fastest) to red (slowest) for that particular query. Systems that lack the functionality (e.g., correlated subqueries) required for certain queries are marked with a value of `N/A` in the result tables.

To ensure an accurate comparison and that systems were tuned and configured appropriately, we also reached out to developers from each of the tested systems. Again, all benchmark details, including the schema, data, and queries, are publicly available [98].

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	1,169	51	652	**20**	376
Q2	1,498	99	883	**17**	480
Q3	1,723	29	531	**26**	1,143
Q4	1,313	69	235	**17**	799
Q5	1,871	**121**	1,348	507	1,477
Q6	4,427	**311**	614	860	4,987

(a) Scale Factor 1x (5.2GB)

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	15,173	349	8,143	**83**	5,951
Q2	20,209	757	13,200	**656**	7,513
Q3	23,914	**766**	4,712	988	18,621
Q4	17,229	526	3,774	**144**	12,991
Q5	25,569	**964**	7,480	12,053	23,670
Q6	70,378	8,909	**7,668**	19,781	85,267

(b) Scale Factor 20x (103GB)

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	27,656	568	15,862	**261**	27,860
Q2	36,758	1,785	29,466	**779**	36,913
Q3	43,422	1,404	11,412	**1,283**	43,551
Q4	31,289	1,107	7,411	**979**	31,490
Q5	45,451	**2,293**	17,027	24,907	47,086
Q6	127,694	17,680	**14,611**	35,135	128,298

(c) Scale Factor 40x (206GB)

Figure 4.2: Performance Monitoring Case Study Runtimes (ms)

For reproducibility, all experiments were conducted on a single `n1-ultramem-40` Google Cloud instance [49], which has 40 vCPUs (Intel E7-8880 v4) and 961 GB RAM. Although some systems (e.g., SparkSQL, CrateDB) have distributed capabilities, we used only a single machine to ensure a fair comparison with the exclusively single-node systems. However, the use of high-end hardware (e.g., CPUs with many cores, abundant main memory) still provided the distributed-capable systems with a wealth of resources to leverage.

Due to space constraints, we do not provide detailed runtime analysis for every query, but rather try to focus our attention mainly on results that are particularly interesting, insightful, or surprising.

4.3.1 Performance Monitor

Figure 4.2 shows the runtime for each system for each query for three different scale factors of the Performance Monitor case study. In general, CrateDB and MonetDB appear to be the best systems for this particular case study. Certain properties of the data and queries (e.g., wide table with many attributes, queries that touch only a few attributes, mostly numeric values) mean that column-oriented systems have an inherent advantage over row-based systems.

For queries over short time ranges in particular, TSDBs are, unsurprisingly, superior to their regular DBMS analog. For example, CrateDB beats its most similar counterpart, MonetDB, in Q1-Q4 almost universally across all scale factors, compared to Q5 and Q6, which do not have any time-based filtering operations. Similarly, TimescaleDB dramatically outperforms PostgreSQL on the same four queries, at least for the two smaller scale factors.

Counterintuitively, SparkSQL performs better on some of the longer-range time queries (e.g., Q6 which spans a full year) than the shorter time range queries (e.g., Q1 which spans only the most recent 12 hours), particularly noticeable in the larger scale factor experiments. This is due to the fact that SparkSQL does not have any indexes and thus must perform a full table scan for all queries, so the query runtime is dictated by the complexity of the projection, aggregation, and group-by features of the query. Therefore, we observe this interesting result where short-range queries with more complex project/group-by/aggregate components actually take (sometimes much) longer than queries over the full time range for SparkSQL.

In terms of the ability to handle larger data sizes, SparkSQL has by far the most impressive scalability results, which should not be surprising since it is designed foremost as a highly parallel and scalable system. In fact, after the first scale factor, SparkSQL actually becomes the best for Q6 and second place for Q5. TimescaleDB has generally poor scalability, eventually becoming worse than PostgreSQL for all queries, even over the short time ranges, due to the overhead of hypertable management associated with larger dataset sizes. CrateDB scales well for Q1, but for short time ranges (e.g., Q2, Q3), the data grows larger than the CPU cache and falls out to main memory. The effect of this can be seen in the sharp order-of-magnitude differences between scale factor 1x and scale factor 20x in Q2 and Q3, but then only slight increase in runtime in the same queries when going from scale factor 20x to 40x. We observe similar behavior in Q4, but, in this case, it is only slightly outside the CPU cache on scale factor 20x and fully outside on scale factor 40x, making the effect more dramatic between 20x and 40x in this case.

Query 1

Over the past twelve hours, what are the basic metrics for the web servers?

To determine the cause of suddenly slow responses to web requests, this query computes basic statistics for only the web servers. These statistics include the minimum, maximum, and average values of CPU usage as well as the number of bytes in/out over the past hour. This is a very selective query, since it operates on a small time interval and computes the aggregates for only the web servers.

Unsurprisingly, since CrateDB and MonetDB are column stores, they were able to outperform all other systems for this query by at least an order of magnitude.

Query 2

Over the past day, which lab machines have been down?

Since computer labs are often booked for classes and must be fully functional, this query determines which machines in any of the computer lab classrooms have been down over the past day. This query operates on a short time window and checks for machines which report null values for metrics.

Again, CrateDB was the best and MonetDB was second. Interestingly, the difference between CrateDB and MonetDB is much smaller in the scale factor 20x. As previously explained, this is because of CrateDB's caching behavior; that is, CrateDB experiences a major slowdown from 1x to 20x, but only a slight slowdown between 20x and 40x.

Query 3

What are the hourly averages for basic metrics from a specific machine over the past two weeks?

To assess a user's complaint of a sluggish workstation, this query retrieves basic performance data (e.g., CPU load, memory utilization) for the specified workstation from the past two weeks, grouped into hourly averages. Query results can then populate a time series visualization, revealing any anomalous spikes or patterns of unusual behavior, and thereby allow IT personnel to address the complaints (e.g., by adding more memory to the machine). This query therefore represents a relatively simple aggregation over an intermediate time scale (i.e., two weeks), grouped by an extracted time value, for only a single machine.

CrateDB and MonetDB perform similarly for Q3. Interestingly, CrateDB beats MonetDB by only a few milliseconds in the smallest scale factor, performs worse in the 20x scale factor, and is better again in the 40x scale factor. Again, we suspect this is related to the CPU cache behavior of CrateDB, in particular because of the implementation language of the two systems. Specifically, MonetDB is written in C whereas CrateDB is written in Java, which causes the data storage to be much larger in CrateDB. At scale factor 20x, then, MonetDB is just barely outside the CPU cache, whereas CrateDB is significantly larger than CPU cache. Then, at the largest scale factor 40x, both systems are well outside the CPU cache, and CrateDB is better again because of its efficient time range search.

Query 4

Over the past month, for each server, how many times was the machine blocked waiting for an I/O request?

To determine which servers are bottlenecked by I/O requests, this query can help an IT professional figure out which machines to upgrade (e.g., install an SSD) or if an unsuspected process is busy writing data to disk. This query sorts machines by the number of times that the CPU was fully blocked waiting for an I/O request during the past month.

Yet again, CrateDB and MonetDB are clear winners for this query. CrateDB exhibits better scaling behavior than observed in the previous queries in terms of falling out of CPU cache because the query touches fewer columns.

Also of note, although still several times slower than CrateDB and MonetDB, SparkSQL performs surprisingly well compared to its own performance on the previous queries.

Query 5

For each of the past twelve weeks, what is the aggregate amount of memory available across all VMs?

For planning purposes, the amount of aggregate available main memory can help determine if the IT department needs to add or upgrade servers. Therefore, this query computes the total available memory for each virtual machine over the past twelve weeks. Unlike the previous queries, this query has no selection predicate on the time attribute.

Notably, as the selected time ranges become longer, CrateDB and MonetDB trade places, with the latter performing much better than all other systems for all scale factors in this query. In particular, MonetDB exhibits excellent (sublinear) scalability.

Query 6

What is the total aggregate network I/O traffic (in, out, total) of all file servers grouped by day?

This query can help to determine the total amount of data that is transferred throughout the day to/from the file servers. The results of this query can be used in a visualization to see how the network I/O traffic changes over time, and can help an IT professional determine a good time to take nodes offline for maintenance or when to add new file servers. Since such trends are often longer term (e.g., seasonal), this query does not have any time-based predicate.

MonetDB performs 2x better than the second place, SparkSQL, for scale factor 1x, but SparkSQL actually overtakes MonetDB for the two larger scale factors, becoming markedly better. This query, in particular, showcases SparkSQL's excellent scalability.

Also important to note is TimescaleDB, which actually performs worse than vanilla PostgreSQL for all scale factors on this query. This is due to the overhead of the hypertable management that TimescaleDB uses for optimizing time based queries and, since this query contains no time-based filter predicate, TimescaleDB incurs extra overhead with no benefit.

4.3.2 Web Server

Figure 4.3 shows the performance of each of the tested systems for the Web Server case study over the base dataset as well as for two scale factors. As previously mentioned, this case study aims to evaluate how systems store and process strings, since the largest attribute is the requested resource.

In general, MonetDB performs the best for four of the six queries, although it has notable difficulty with both Q1 and Q4. Surprisingly, other systems are sometimes significantly faster, such as CrateDB and TimescaleDB in Q1, which perform at least two orders of magnitude better across all scale factors.

The row-based systems (i.e., PostgreSQL and TimescaleDB), interestingly, perform much better relative to their performance against the column stores in the Performance Monitor case study, with PostgreSQL even having the lowest runtime for Q5 across all scale factors. This effect is due to the

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	12,802	14,757	5,011	**13**	243
Q2	13,780	**360**	11,944	1,622	566
Q3	14,472	**1,036**	50,940	6,022	5,690
Q4	77,483	**2,253**	51,032	55,363	68,611
Q5	**46,460**	112,687	46,666	454,015	53,116
Q6	92,366	**2,128**	20,230	175,284	95,434

(a) Scale Factor 1x (54GB)

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	21,522	32,187	9,308	**62**	491
Q2	23,788	**756**	25,773	8,103	1,358
Q3	63,108	**9,026**	135,551	62,207	54,398
Q4	218,985	**35,060**	141,776	234,185	235,513
Q5	**124,412**	N/A	148,951	3,377,872	125,495
Q6	119,729	**2,805**	37,705	783,519	137,372

(b) Scale Factor 2x (106GB)

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	39,827	66,352	26,401	**219**	833
Q2	44,355	**1,554**	52,717	24,957	2,447
Q3	165,827	**25,050**	207,407	254,428	143,992
Q4	557,518	**94,023**	365,047	810,282	555,953
Q5	**276,460**	N/A	288,007	9,930,943	294,702
Q6	181,964	**4,412**	84,043	2,609,120	207,228

(c) Scale Factor 4x (210GB)

Figure 4.3: Web Server Case Study Runtimes (ms)

specifics of the data in this use case, where the tuple size is dominated by a single large string field (i.e., the request string). Therefore, the advantages of a column store are much less pronounced in this case study, sometimes even being harmful for string processing operations when aggressive columnar compression has been applied.

Several queries in this case study stress a new dimension of query processing: how systems handle aggregation queries which contain many group-by keys. Interestingly, in the majority of these cases (e.g., Q3, Q4), MonetDB exhibits speedups of at least 5-25x across all scale factors.

Similar to the previous case study, we observe the same pattern between the relative performance of PostgreSQL and TimescaleDB with the different time predicates, but the difference is much less pronounced, since query processing time is dominated by other factors (e.g., string processing, large group-by clauses), thereby hiding some of the benefits and overheads related to TimescaleDB's hypertable storage design.

Query 1

In the past day, which requests have resulted in an internal server error?

To determine which web pages are not working due to server-side errors in web applications, this query lists all web requests that returned a status code in the 500 range during the past day. Using this, a web administrator can fix any errors to ensure users can access appropriate web pages.

CrateDB is the clear winner, beating the second place finisher (TimescaleDB) by an order of magnitude. In this query in particular, CrateDB's ability to quickly access tuples from only the past day is beneficial, since the remainder of the query consists of one other simple selection predicate and a very straightforward group-by aggregation. CrateDB performs consistently well across all scale factors.

Surprisingly, MonetDB is the worst across all scale factors, and even vanilla PostgreSQL is better. It is unclear why MonetDB performs so poorly in this query, and we can contacted the developers to investigate.

Query 2

Over the past week, find the logs for requests for resources that are known to be security attacks.

In order to try to stop future attacks or to investigate a previous attack, this query finds all of the web requests over the past week that try to access resources that should be protected (e.g., the `/etc/passwd` file). These sorts of malicious requests often come from individuals that are looking to find any unprotected way to gain access to the network. However, the malicious string can appear in various forms and locations in the resource request (e.g., `home.php?q=/../etc/passwd`), so the query uses a wild card character before and after each pattern. In addition to the relatively selective time-based selection, the wildcard string filtering component tests yet another dimension of each system.

Again, TimescaleDB is competitive with the winner for this query (MonetDB) because of the relatively short time range of only one week, which is interesting that a row store performs almost as well as a column store.

Despite the short time range, CrateDB performs significantly worse due to its dictionary encoding scheme for the request URLs. Whereas MonetDB chooses not to perform dictionary encoding due to the large number of distinct URLs, CrateDB always performs dictionary encoding on string columns and, therefore, must pay the decompression penalty for every URL in order to perform the `LIKE` comparison.

Query 3

In the past two weeks, which clients are making the most requests?

Since denial of service (DoS) attacks can cause outages and consume valuable resources, this query aims to determine which remote hosts are repeatedly issuing requests to the webserver over the past two weeks. Based on the query results, an administrator can add the malicious IP addresses to a blacklist in order to free up resources and prevent such an attack. Therefore, this query counts the number of distinct requests per client IP address over the past two weeks, ordered by the number of requests.

MonetDB is again the clear winner for this query across all scale factors due to its ability to efficiently handle a large number of distinct group-by keys (i.e., the query result requires to group by every IP address in the time range before selecting the top 10 most frequent). On the other hand, SparkSQL does not do well with many distinct group-by keys, and is in last place on both the 1x and 20x scale factors, being replaced by CrateDB in the 40x scale factor, where it too succumbs to the large number of distinct group-by keys.

Query 4

What are the daily unique visitors over the past month?

To study the past patterns of traffic to the webserver, the webmaster can issue a simple query that counts the number of requests the webserver has received grouped by day for the past month. This information allows the IT department to more intelligently provision for peak load as well as find good times to take the website down in order to perform system maintenance. This query therefore requires computing a simple count grouped by a derived attribute (a cast of the timestamp value to get the day) which has many group by keys, stressing yet another dimension of system performance.

Similar to Q3, MonetDB again dramatically outperforms all other systems due to the requirement of counting only distinct client IP addresses, which requires either a hash-based set data structure or a full sort of the data to avoid counting duplicates. SparkSQL therefore scales similarly as in Q3, but CrateDB scales much worse.

Query 5

For the past three months, what is the average path depth for requests to all top level directories?

The number of subdirectories that a resource contains (i.e., path depth) roughly translates to the number of times a user must follow a link to find a resource. To improve user experience, the webmaster can issue this query in order to determine if visitors to the website are accessing information that is buried in the lower layers of the subdirectories. Based on this result, the webmaster can move resources around so that frequently accessed pages are easily accessible. Since this query counts the number of subdirectories in each top-level directory over the past three months, it involves various string operations as well as a time-based selection.

Surprisingly, PostgreSQL is the best on all scale factors. SparkSQL is second place, but scales less well than Postgres, which is also very surprising. Although PostgreSQL reads in the entire row, this query is CPU-bound operating on the large resource string, masking the inefficiencies of a row store. Additionally, since the query involves expensive string manipulation and is CPU bound, the fact that PostgreSQL does not perform compression is beneficial in this case.

CrateDB is the worst, primarily due to its highly unoptimized string manipulation functions, and the need to decompress highly compressed strings to do the string manipulations.

Strangely, MonetDB writes several gigabytes to the local disk during query execution, causing it to crash at larger scale factors. We believe this is a bug in the query processing and have contacted the system developers about it.

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	633	79	N/A	N/A	**29**
Q2	674	**156**	4,860	299	341
Q3	232	**28**	1,155	N/A	288
Q4	650	**34**	860	44	144
Q5	711	**14**	789	65	881
Q6	2,375	4,091	**599**	27,569	2,720

(a) Scale Factor 1x (1.7GB)

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	22,867	**555**	N/A	N/A	1,317
Q2	24,531	**1,226**	94,153	20,666	4,336
Q3	23,300	**245**	12,526	N/A	5,774
Q4	22,636	**91**	7,733	850	3,919
Q5	25,983	**159**	6,638	4,315	36,158
Q6	118,006	229,302	**12,953**	1,677,338	137,648

(b) Scale Factor 60x (100GB)

	PostgreSQL	MonetDB	SparkSQL	CrateDB	TimescaleDB
Q1	41,594	**993**	N/A	N/A	41,643
Q2	44,224	**2,717**	252,777	50,881	44,393
Q3	41,544	**526**	26,007	N/A	41,363
Q4	41,359	**157**	11,697	3,650	40,831
Q5	46,599	**288**	11,614	25,933	47,304
Q6	189,695	476,909	**26,702**	3,876,191	208,372

(c) Scale Factor 120x (200GB)

Figure 4.4: IoT Case Study Runtimes (ms)

Query 6

What is the average and maximum instantaneous bandwidth usage?

A web master is often interested in determining how much traffic is routed through the webserver. This query therefore computes the average and peak bandwidth for all traffic that the webserver has handled. Based on the query's results, the web master can better provision resources (e.g., add additional machines, upgrade internet connectivity) for the peak loads that the server receives. This query involves simple aggregates (sum, average, max) for many group by keys with no time-based selection.

MonetDB is the best and scales much better than the other systems. Similar to Q6 of the Performance Monitor case study, TimescaleDB experiences noticeable overhead compared to PostgreSQL due to its hypertable storage format. CrateDB is noticeably worse than all other systems across all scale factors despite the relatively straightforward aggregation query since it cannot efficiently handle query queries with many group-by keys.

4.3.3 IoT Events

Figure 4.4 shows the runtime for each of the systems for the final case study, IoT Events, over the base data as well as two scaled sizes. Unlike the other two case studies, this case study primarily stresses query optimization, time manipulation functions (e.g., `CAST, EXTRACT`), and advanced SQL operations (e.g., `CASE`, `HAVING`).

Since query optimization and planning is particularly important in this case study, a more mature DBMS like MonetDB wins in many queries, and a similarly mature system like PostgreSQL performs relatively well compared to the much simpler query patterns in the Performance Monitor and Web Server benchmarks. Moreover, since these queries involve more complex operations, the TSDBs do not exhibit the same performance improvements for short time range queries as in the previous two case studies. In fact, TimescaleDB only performs best in one query for one scale factor, and CrateDB never finishes in first place.

The correlated subquery performance of different systems is interesting, but unfortunately we have relatively few data points because certain systems have limited (i.e., SparkSQL) or no (i.e., CrateDB) support for this operation.

Surprisingly, the TSDBs, which are intended to natively handle time, do not seem to have any noticeably better performance for timestamp manipulation/value extraction compared to the DBMSs.

In terms of scalability, PostgreSQL and MonetDB exhibit relatively good scaling performance, as in the previous two case studies, while CrateDB and TimescaleDB exhibit similar slowdowns as in the previous two case studies. SparkSQL again exhibits excellent scalability, except in Q2 due to a poor execution plan.

Query 1

For the past week, which devices report temperatures that vary greatly from the hourly average temperature of each floor?

This query finds locations in the building that with large differences in temperature from the average hourly temperature per floor in the past week. Such a query can help to identify, for example, if a window has been left open in an office during the winter, which could cause a pipe to freeze when the temperature drops at night.

Surprisingly, TimescaleDB outperforms MonetDB by almost a factor of three on the smallest scale factor but does not scale well to the larger dataset sizes, where MonetDB becomes the frontrunner. TimescaleDB's performance is still an order of magnitude better than PostgreSQL on scale factor 60x, but degrades to essentially the same performance as PostgreSQL on scale factor 120x due to the previously discussed overhead associated with TimescaleDB's hypertable management.

Unfortunately, SparkSQL and CrateDB cannot run this query due to the correlated subquery in the `SELECT` clause, so they are shown as N/A in all scale factors.

Query 2

What are the top ten power consuming devices over the past week?

In order to study the energy consumption of the building, this query returns the devices that have consumed the most power over the past week. From this query, facilities personnel can upgrade outdated equipment or create a schedule to turn off the device in order to save power. Since the power readings are measured at a given instant, the query first filters the data to the specified time range and aggregates the total power consumed by each device per day. Then, for each device, the query averages the daily power consumption. Like the previous query, this query also contains two nested subqueries but does not have as many group by keys in the aggregation.

MonetDB is the best in this query across all scale factors. CrateDB and TimescaleDB are roughly 2x worse for the base scale factor, but do not scale as well as MonetDB and end up performing worse than even PostgreSQL at the largest scale factor. Surprisingly, SparkSQL exhibits incredibly poor scalability for this query, taking up to five times longer than the next worst performing system. This is in stark contrast to its performance in every other query in the benchmark, where it exhibits the best scalability out of all the systems and is likely due to a suboptimal plan that is not fully parallelized.

Query 3

For each room in the building, when is the least disruptive weekday and time to schedule maintenance work based on use over the past month?

This query determines, for each room, the best time to perform scheduled maintenance tasks (e.g., change light bulbs, fix broken desks, replace projectors). The query first filters the data to the specified time range and extracts the day of week and hour from each motion event. Then, the query filters the data to working hours (weekdays between 9am and 5pm) and counts the number of motion events per hour. Next, the query averages the number of events per device during each hour in a week. Finally, the query selects the day and hour of the week that has the fewest number of motion events. Since this query contains several nested queries, timestamp manipulations, and filtering conditions, it stresses many parts of each system.

Both MonetDB as well as TimescaleDB perform best in this specific query. MonetDB is able to efficiently perform aggregations with many group by keys and also has highly optimized timestamp manipulation functions, allowing it to quickly perform the required date calculations. Finally, MonetDB's query optimizer chooses a good plan to execute the multiple subqueries that avoids materialization. On the other hand, TimescaleDB is able to leverage the time-based property of this query in order to return the query result almost as fast as MonetDB.

Unfortunately, similar to Q1, CrateDB does not support correlated subqueries (this time in the `WHERE` clause) and is therefore unable to run this query.

Query 4

Over the last month, when and where in the building are there large temperature variations?

Since there are large temperature variations throughout many buildings, this query aims to find out where in the building, and during what time these variations occur. This information is valuable to facilities management, who try to maintain a comfortable environment throughout the building.

Using this data, the facilities department can modify existing heating/cooling systems, identify malfunctioning equipment, or install new devices (e.g., air handlers, thermostats). This query tests time manipulation functions (e.g., `CAST`, `EXTRACT`), `HAVING` clause, and group-by aggregation of `DISTINCT` keys.

MonetDB is the clear winner across all scale factors. However, for the base scale factor, MonetDB and CrateDB are quite close, and the performance gap widens at larger scale factors since CrateDB does not scale nearly as well.

Query 5

Which doors in the building are most active?

Since some doors are opened thousands of times per day, they require frequent maintenance for devices like pneumatic door closers and handicap-accessible door motors. Therefore, to inform facilities employees, this query ranks each of the doors based on the number of times that they are opened or closed. This allows workers to check high traffic doors and perform preventative maintenance. This query therefore examines all doors over the entire time period and performs a simple group-by aggregation to compute the count.

Again, MonetDB is the clear winner for all queries. CrateDB is similar in performance for the scale factor 1x, but exhibits even worse scaling behavior than in Q4, so much so that SparkSQL overtakes it for second place in the largest scale factor.

Query 6

For each category of power outlet, what is the total power consumption?

To determine which types of devices consume the most amount of power, this query calculates the total power consumption for each device type (e.g., printers, coffee machines, vending machines). Based on the result of this query, building administrators can determine which devices are using the most power and take actions in order to try to reduce the overall power consumption of the highest consuming devices. Since the type of each device (e.g., printer) can be derived from the device's name (e.g., printer_bw3), this query involves a case statement and `LIKE` predicate. Once the device's type is derived, the query performs a simple group-by sum aggregation over the power readings.

SparkSQL is able to execute this query faster than any other system, primarily since its compression algorithm for strings enables efficient `LIKE` predicates and its code generation helps optimize the execution of the case statement. Additionally, this query is a simple table scan over the entire data, which is what SparkSQL is optimized for. Similarly, PostgreSQL also performs a full table scan and its string representation enables efficient evaluation of like predicates.

4.4 Related Work

Although several benchmarks for evaluating DBMSs have been previously presented, our work differs from existing work in the following ways. Specifically, our work differs from existing benchmarks since (1) it tests multiple time-specific tasks that extend beyond simple time-based filtering and (2) our data captures the skew and patterns in real workloads that are not captured by other benchmarks.

Although popular OLAP benchmarks (e.g., TPC-H [143], TPC-DS [142]) involve time, they only test time-based filtering (e.g., select orders from the past month). On the other hand, MgBench tests more complex time operations (e.g., time-based aggregation, timestamp manipulation) and uses real-world data with subtle yet important trends.

Similar to Keogh et al. [79], we argue that real-world data and queries are necessary to evaluate performance in the domain of machine-generated time series data. Our work, however, focuses on system-level aspects for OLAP-style queries rather than stand-alone data mining algorithms.

The FinTime [70] benchmark contains data and queries for testing systems designed for analyzing stock market data. Unlike our benchmark, these queries focus only on stock market specific tasks (e.g., computing moving averages) and is therefore not generalizable to other time-oriented domains.

Like Linear Road [8], RIoTBench [130], and work from Shukla et al. [131], our goal is to allow others to evaluate systems using real-world data and workloads. Similarly, the TPCx-IoT benchmark [141] performs real-time data ingestion and streaming analytics for IoT gateways. However, our benchmark focuses on measuring the performance of systems for analytical queries, not streaming use cases characterized by continuous queries.

Liu et al. [92] present a benchmark that focuses on analyzing data collected from smart electricity meters. The presented data is highly domain specific, and the queries test only simple operations (e.g., scan, selection, aggregation) while ignoring more complex aspects (e.g., string manipulations, subqueries).

Although IoTAbench [9] offers a data generator that tries to mimic the patterns of real-world readings (e.g., power readings), readings occur at fixed intervals (e.g., 10 mins) and therefore only capture a small subset of possible scenarios. In many cases, data arrives at non-uniform intervals, making storage and analysis more difficult. Our benchmark incorporates this property (i.e., in the Web Server and IoT Events use cases) and therefore tests a broader range of machine-generated data use cases.

Finally, Mitschang et al. [15] present a detailed description of popular open source time series databases as well as some simple benchmarks (e.g., insert and single value lookup performance). Instead, we present a benchmark which contains representative use cases in order to study the performance of competing systems in real-world scenarios.

4.5 Key Takeaways

One of the goals of this work is to provide a detailed analysis of existing systems for managing machine-generated data and offer forward-looking guidance to future system designers. In this section, we present answers to the following two important questions: (1) which existing system is best for machine-generated data analysis and (2) looking forward, how can we design next-generation systems that have the potential to offer substantial (i.e., multiple orders-of-magnitude) performance improvements over current approaches?

4.5.1 Choosing an Existing System

As previously mentioned, a plethora of different types of systems currently exist, as well as a deluge of apparently contradictory "benchmarking" results [65, 64, 66, 140, 138, 30, 38]. Given the diverse nature of machine-generated data analysis applications, picking a system for a particular use case is not straightforward. In fact, as our results show, there is no system that is dominant in all of the presented use cases. In the following, we provide high-level guidance on when to use a DBMS, an analytics framework, and a TSDB.

DBMSs

As our results show, general-purpose DBMSs are, in fact, a surprisingly good choice in order to achieve reasonable performance on most analytical workloads for machine-generated data. As shown in our results, specialized time-based systems have only marginal performance improvements when compared to state-of-the-art OLAP DBMSs. Only in very narrow cases (e.g., Q1 of Web Server) do the TSDBs achieve orders-of-magnitude performance improvements over traditional DBMSs. Since machine-generated data analysis often involves a wide variety of dimensions other than time (e.g., strings, numeric values) and operations (e.g., aggregates, string operations), specialized TSDBs that are optimized only for slice and dice operations along the time dimension are not always the best choice, especially if queries are likely to touch many attributes and/or perform more complex, CPU-bound functions (e.g., string manipulation). Therefore, the default choice should be a DBMS, since they generally perform decently across the board.

Analytics Systems

Generally, for smaller dataset sizes, SparkSQL does not perform very well, due to its high startup cost and added overhead associated with a distributed system. In most cases, however, SparkSQL scales incredibly well, offering the best relative performance at higher scale factors. In addition to being highly scalable, SparkSQL also performs well for queries that require a full table scan, since it is optimized for scanning large datasets. Although outside the scope of this paper, SparkSQL also provides added flexibility, since queries can leverage the much more powerful Spark API. For extremely large datasets, use cases where data is likely to grow dramatically in size, or instances where more advanced analytics are required (e.g., machine learning), we recommend SparkSQL or similar analytics systems.

TSDBs

As we show across all three case studies, the TSDBs generally shine when considering extremely short time-based range predicates. However, in some cases, even though a query has a highly selective time-based predicate, other operations (e.g., string manipulation) or predicates over other attributes can dominate query runtime. For example, although Q2 of the Web Server case study selects only one week, the string predicates are more selective and computationally expensive. Moreover, as our evaluation shows, the TSDBs often start to lose their advantages at higher scale factors, since the time spent narrowing down the tuples in the selected time range represents a smaller percentage of the total query runtime. Therefore, we recommend using a TSDB only for workloads that have relatively simple queries with highly selective time-based predicates.

4.5.2 Building a Next-Generation System

So far, our discussion has focused exclusively on understanding the tradeoffs associated with existing systems. This section describes some forward-looking recommendations for future system designers.

As the results of our study show, the tested TSDBs are better than traditional DBMSs only in some cases, and often they perform notably worse. However, when the TSDBs are better, they often show speedups of only 2-5x. This is because, for the most part, they simply implement better versions of storage formats, partitioning, and indexing.

However, in order to achieve orders-of-magnitude performance improvements, we will instead need radically different designs.We believe that the single biggest opportunity for the next generation of systems is to figure out how to fully leverage the patterns inherent in real-world data. As highlighted in Section 4.1, these types of patterns are abundant, subtle, and often difficult to identify.

One category of patterns comes from the timestamps themselves. For example, some applications (e.g., server monitoring) report metrics at pre-defined, fixed intervals, whereas others (e.g., IoT sensors) report only when some event occurs. This discrepancy will require an adaptive approach to handle patterns arising from both regular (Performance Monitor) and irregular (Web Server, IoT Events) intervals.

Another category of patterns stems from the data values. For example, temperatures predictably change over the course of a day, and even longer-term temperature trends exist between different times of year. By leveraging these types of pattern shapes and seasonality, we believe that we can much better design certain system components.

In the following, we give some examples of how system components can be better designed by considering patterns.

Compression

With the rapid growth of machine-generated data, compression is an essential part of any system. To reduce storage consumption, some TSDBs discard older data [126, 53], while others aggregate older time intervals to higher granularities [1, 57].

Alternatively, lossless compression algorithms reduce the data storage requirement without sacrificing data precision. For example, traditional columnar compression techniques (e.g., run-length encoding, delta encoding), as well as more general-purpose compression algorithms (e.g., Huffman

Coding, LZ), are often effective at reducing size, but even better would be if we modify them to be time-specific.

Some existing TSDBs apply specialized algorithms for compressing time series data. For example, Gorilla [116] uses techniques like delta-of-delta timestamps and XOR'd floating point values that they have found work well in practice. Similarly, Akumuli [5] uses a modified version of the LEB128 variable encoding algorithm, which is an adaptive algorithm that can handle regular and irregular timestamp intervals.

However, we believe these time series compression algorithms only scratch the surface of what is now possible. Beyond these low-level patterns, we also believe that high-level trends are possible to leverage (e.g., compressed trickles [125]). For instance, deep learning techniques have recently been applied to the task of image compression [127, 73], since convolutional neural networks are particularly good at recognizing high-level patterns in the data. We believe that these same techniques could be harnessed to recognize and compress high-level patterns in machine-generated time series data and have the potential to offer huge space savings.

Indexing

Although generally efficient, existing TSDBs that index the time dimension do not leverage high-level patterns that exist in the underlying data. We believe that in order to meaningfully improve the performance of such a system, they must begin to incorporate higher-level patterns into time-optimized index structures.

For example, both Learned Indexes [84] and A-Trees [48] logically represent indexes as functions that map a key (e.g., a timestamp) to a storage location (e.g., position in a sorted array). These approaches then try to learn/approximate this function, essentially taking advantage of any trends that exist in the underlying data. These works are early examples of how to leverage patterns that exist in the underlying data to outperform state-of-the-art alternatives. We believe that this is an exciting direction to explore further.

Partition Pruning

While indexes can help to efficiently locate the data items necessary for answering a query, partition pruning allows a system to avoid processing entire data partitions that contain no relevant data items.

A wide variety of compact data structures (e.g., bitmaps, zone maps, column imprints [132]) allow the system to avoid examining portions of the data that are not relevant to a query. By augmenting a time-based data partitioning strategy (e.g., TimescaleDB's hypertable chunks) with these techniques, systems could efficiently handle filter predicates on both the time and non-time data dimensions.

However, these pruning data structures are maintained based on fixed-size data partitions. Instead, we believe that we can leverage the patterns to adaptively partition the data, allowing the system to encode low-information density regions of the pattern in larger bins and high-information density regions of the pattern with finer-grained bins. This would reduce the number of partitions that the system must both store and search, which could dramatically improve query performance. Even more interestingly, we can store more than just basic pruning mechanisms instead encoding pattern-relevant data summaries (e.g., function representations) that enable multi-dimensional pruning.

Chapter 5

Approximate Index Structures for Time Series Data

Tree-based index structures (e.g., B+ trees) are one of the most important tools that DBAs leverage to improve the performance of analytics and transactional workloads. In particular, as shown in the previous chapter, indexing can be especially beneficial when analyzing time series data since users are often interested in specific time intervals (e.g., last month, yesterday). However, for main-memory databases, tree-based indexes can often consume a significant amount of memory. In fact, a recent study [154] shows that the indexes created for typical OLTP workloads can consume up to 55% of the total memory available in a state-of-the-art in-memory DBMS. This overhead not only limits the amount of space available to store new data but also reduces space for intermediates that can be helpful when processing existing data.

To reduce the storage overhead of B+ trees, various compression schemes have been developed [11, 52, 17, 157]. The main idea behind these techniques is to remove the redundancy that exists among keys and/or to reduce the size of each key inside a node of the index. For example, prefix and suffix truncation can be used to store common parts of keys only once per index node, reducing the total size of the tree. Additionally, more expensive compression techniques like Huffmann coding can be applied within each node.

Although each of the previously mentioned compression schemes reduce the size of an index node, the memory footprint of these indexes still grows linearly with the number of distinct values to be indexed, resulting in indexes that can consume a significant amount of memory.

This observation is especially true for data such as timestamps or sensor readings that are generated in a wide variety of applications (e.g., Internet autonomous vehicles) and other data types such as geo-coordinates or strings have similar properties. Even worse, the number of unique values for such data types typically grow over time, resulting in indexes that are constantly increasing in size. Consequently, a DBA has no way to restrict memory consumption other than dropping an index completely.

To tackle this problem, we present A-Tree, a novel learned index structure [84] which can be bounded in space independent of the number of distinct values that need to be indexed. In the basic version of A-Tree, we assume that the table data to be indexed is sorted by the index key, making our index similar to a clustered index whereby only the first key of each page is inserted into the index. Unlike typical clustered indexes which split the data into fixed-sized pages, A-Tree uses piece-wise linear functions to partition the data into variable-sized segments, allowing us to use the linear functions to quickly approximate the position of an element. By approximating the trends within the data, A-Tree can reduce the memory consumption of an index by orders of magnitude compared to a traditional B+ tree. Furthermore, we also show how our techniques extend to secondary (i.e., non-clustered) indexes.

At the core of our index structure is a parameter that specifies the amount of acceptable error (i.e., a constant that is the maximum distance between the predicted and actual position of any key). Unlike existing index structures, our error parameter allows us to balance the lookup performance and the space consumption of an index. To navigate this tradeoff, we present a cost model that helps a DBA choose an appropriate error term given either (1) a lookup latency requirement (e.g., 500ns) or (2) a storage budget (e.g., 100MB). Using a variety of several real-world datasets as well as a synthetically generated worst-case dataset, we show that our index structure is able to provide performance that is comparable to full and fixed-paged indexes (even for a worst-case dataset) while reducing the storage footprint by orders of magnitude.

Using linear functions to approximate the distribution makes A-Tree a learned index [84]. However, in contrast to the initial techniques in [84], our proposed techniques allow us to (1) bound for the worst-case lookup performance of a key, (2) efficiently support insert operations, and (3) enable paging (i.e., the entire data does not have to reside in a continuous memory region). Additionally, A-Tree is different from Correlation Maps (CMs) [83] which try to leverage correlations between an unclustered attribute any a primary index instead of creating a full index. Unlike CMs, which partition the secondary attribute into buckets of equal sizes and map each value range to ranges of a clustered primary key, our approach (1) does not rely on an existence of an existing index, and (2) produces variable-ranged segments that better model the underlying data distribution. Furthermore, the problem of approximating distributions using piece-wise functions is also not new [45, 21, 40, 129, 80, 26, 24, 146, 91]. To the best of our knowledge, though, none of these techniques have been applied to indexing and therefore do not consider operations that indexes must support.

Another interesting observation is that our variable-sized paging approach is orthogonal to node-level compression techniques such as the previously mentioned prefix- or suffix truncation. In other words, since A-Tree internally uses a tree structure to organize our variable-sized pages, we can still apply these well-known compression techniques to further reduce an index's size. Similar observations hold also for techniques such as adaptive indexing [63], holistic indexing [117], and database cracking [62], which refine as the data is queried to improve performance. To that end, these techniques are also orthogonal since our technique leverages the data distribution, instead of knowledge about the workload, to compress the index size.

In the following, we first present an overview of our new index structure called A-Tree and discuss an efficient one-pass segmentation algorithm (Section 5.2) that incorporates a tunable error parameter that balances the lookup and insert performance of our index. Then, we discuss the core index operations — lookups (Section 5.3) and insertions (Section 5.4). We then propose a cost model that helps a DBA determine an appropriate error threshold given either a latency or storage requirement (Section 5.5). Finally, using several real-world datasets, we show that our index provides similar (or in some cases even better) performance compared to existing index structures while consuming orders of magnitude less space (Section 5.6).

5.1 Overview

At a high level, indexes (and B+ trees over sorted attributes in particular) can be represented by a function that maps a key (e.g., a timestamp) to a storage location. Using this representation, A-Tree partitions the key space into a series of disjoint linear segments that approximate the true function, since it is (generally) not possible to fully model the underlying data distribution. At the core of this process is a tunable error threshold, which represents the maximum distance that the predicted location of any key inside a segment is from its actual location. Instead of storing all values in the key space, A-Tree stores only (1) the starting key of each linear segment and (2) the slope of the linear function in order to compute a key's approximate position using linear interpolation.

In the following, we first discuss how the idea of using functions to map key values to storage locations intuitively works. Then, we discuss how we leverage this function representation to efficiently implement our approximate index structure on top of a B+ tree for clustered indexes (over a sorted attribute). Finally, we show how our ideas can also be applied to compress secondary indexes.

5.1.1 Function Representation

One key insight to our approach is that we can abstractly model an index as a monotonically increasing function that maps keys (i.e., values of the indexed attribute) to storage locations (i.e., its page and the offset within that page). To explain this intuition, we first assume that all keys to be indexed are stored in a sorted array, allowing us to use an element's position in the array as its storage location.

As an example, consider an IoT dataset, which contains events from various devices (e.g., door sensors, motion sensors, power monitors) installed throughout a university building. In this dataset, the data is sorted by the timestamp of an event, allowing us to construct a function that maps each timestamp (i.e., key) to its position in the dataset (i.e., position in a sorted array), as shown in Figure 5.1. Unsurprisingly, since the installed IoT devices monitor human activity, the timestamps of the recorded actions follow a pattern (e.g., little activity during the weekend and night hours, corresponding to large timestamp intervals spanning relatively few positions in the sorted list).

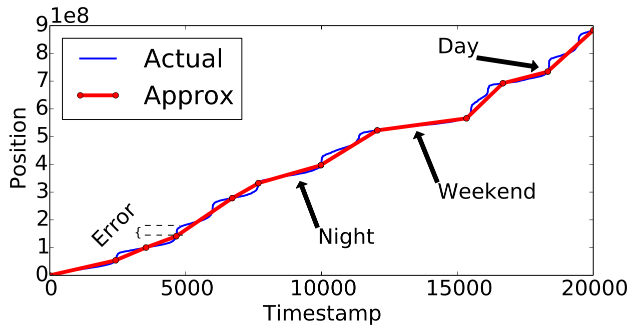


Figure 5.1: Key to Position Mapping for IoT Data

Since a function that represents an index can be arbitrarily complex and data-dependent, the precise function that maps keys to positions may not be possible to learn and is expensive to build and update. Therefore, our goal is to approximate the function that represents the mapping of a key to a position. By approximating this function, the predicted location for a given key does not necessarily point to the true location of the key.

To compactly capture trends that exist in the data while being able to efficiently build a new index and handle updates, we use a series of piece-wise linear functions to approximate an arbitrary function. As shown in Figure 5.1, for example, our segmentation algorithm (described further in Section 5.2) partitions the timestamp values into several linear segments that are able to accurately reflect the various trends that exist in the data (e.g., less activity during the weekend). Since the approximation captures trends in the data, it is agnostic to key density (a trend with sparse keys can be captured as well as a trend with dense keys).

Although more complex functions (e.g., higher order polynomials) can be used to approximate the true function, piece-wise linear approximation is significantly less expensive to compute. This dramatically reduces (1) the initial index construction cost, and (2) improves insert latency for new items (see also Section 5.4).

Although piece-wise linear approximation can help us model any arbitrary function, the resulting function is not precise (i.e., a key's predicted location is not necessarily its true position). We therefore define the *error* associated with our approximation as the maximum distance between the actual and predicted location of any key, as shown below, where $pred\_pos(k)$ and $true\_pos(k)$ return the predicated and actual position of an element k respectively.

$$error = \max(|pred\_pos(k) - true\_pos(k)|) \;\forall\; k \;\in\; keys \tag{5.1}$$

This formulation allows us to define the core building block of A-Tree, a *segment*. A segment is a contiguous region of a sorted array for which any key is no more than a specified error threshold from its interpolated position. Depending on the data distribution and the error threshold, the segmentation process will yield a different number of segments that approximate the underlying data. Therefore, importantly, the error threshold enables us to balance main memory consumption

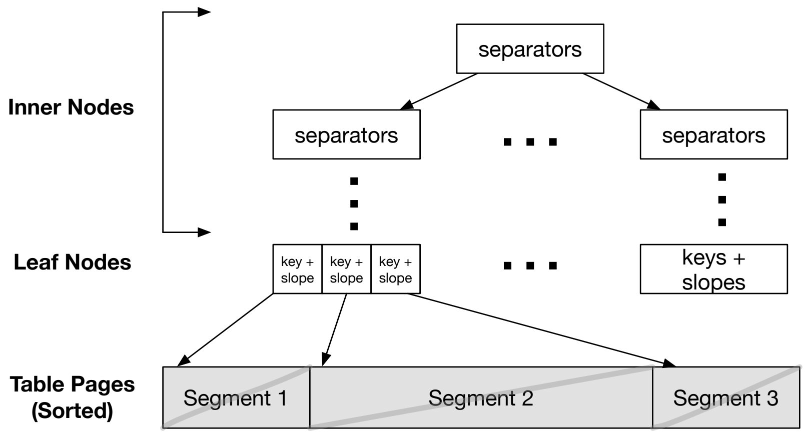


Figure 5.2: A clustered A-TREE index

and performance of our index. After the segmentation process, A-TREE stores the boundaries and slope of each segment (instead of each individual key) in a B+ tree, reducing the overall memory footprint of the index.

5.1.2 A-Tree Design

As previously mentioned, our segmentation process partitions the key space of an attribute into disjoint linear segments such that the predicted position of any key inside a segment is no more than a bounded distance from the key's true position away. A-TREE organizes these segments in a tree to efficiently support insert and lookup operations.

In the following, we first discuss clustered indexes, where records are already sorted by the key that is being indexed. Afterwards, we show how our technique can be extended to provide similar benefits for secondary indexes.

Clustered Indexes

In a traditional clustered B+ tree (e.g., an index-organized table), the table data is stored in fixed-sized pages and the leaf level of the index contains only the first key of each of these table pages. Unlike a clustered B+ tree, in an A-TREE, the table data is partitioned into variable-sized segments (pages) that satisfy the given error threshold. Each segment is in essence a fixed size array, but it is important to note that successive segments can be allocated independently (i.e., non contiguously).

Figure 5.2 shows the structure of a clustered A-TREE index. As shown, the underlying data is partitioned into a series of variable-sized segments that approximate the distribution of keys to be indexed. Depending on the error parameter and the data distribution, several consecutive keys can thus be summarized into a single segment. As we show in our experiments that use several real-world data sets, our index is able to effectively reduce the storage footprint of an index by orders of

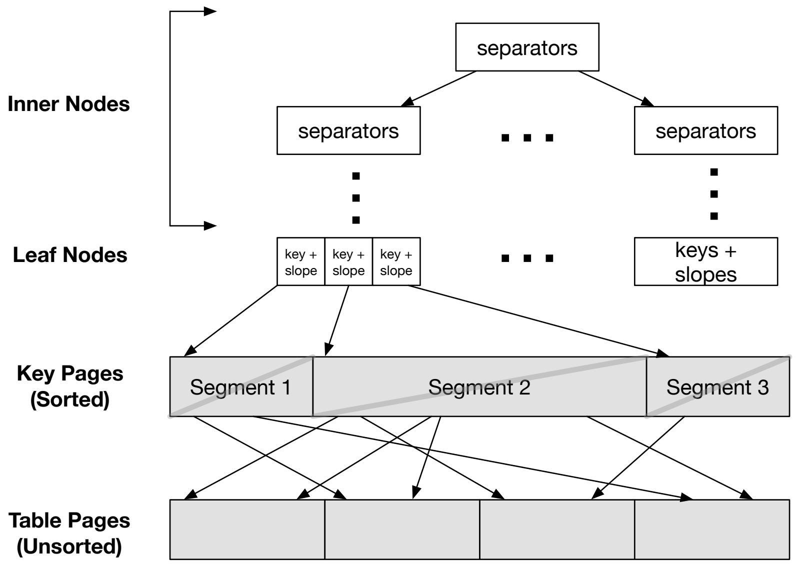


Figure 5.3: A non-clustered A-Tree index

magnitude without sacrificing performance. Details of the segmentation algorithm that divides the table data into variable-sized segments are discussed in Section 5.2.

Unlike a traditional B+ tree, each leaf node in an A-Tree stores the segment's slope, starting key, and a pointer to a segment. This allows us to use interpolation search in each segment since the data within this segment is approximated by a linear function given by the slope.

The inner nodes of an A-Tree are the same as a normal B+ tree (i.e., lookup and insert operations are identical to a normal B+ tree). However, once a lookup or an insert reaches the leaf level, A-Tree needs to perform additional work. For lookups, we need to use the slope and the distance to the starting key in order to calculate the key's approximate position (offset in the segment). Since the resulting position is approximate, A-Tree must then perform a local search (e.g., binary, linear) to find the item, discussed further in Section 5.3.

Insert operations also require additional work upon reaching a leaf level page, since, unlike a normal B+ tree, we must ensure that the error threshold is always satisfied. Therefore, we present two different insertion strategies (described in detail in Section 5.4). The first strategy performs in-place updates to the segment (as a baseline) while the second strategy uses a more advanced buffer-based strategy to hold inserted data items. In both cases, we must (1) ensure that the specified error is always satisfied even after inserting a new data item, and (2) split a segment into one (or more) new segments once the error threshold is exceeded.

Finally, instead of internally using a B+ tree to locate the segment for a key, A-Tree could instead use any other index structure. For example, if the workload is read-only, other high performance index structures (e.g., FAST [82]) can be used. In Section 5.6.3 we show how A-Tree performs when using different internal data structures, including FAST.

Non-clustered Indexes

Secondary indexes can dramatically improve the performance of queries involving selections over a non-primary key attribute. Without secondary indexes, these queries must examine all tuples, which is often prohibitive. Unlike a clustered index, a non-primary key attribute is not sorted and may contain duplicates.

The primary difference between a clustered and an non-clustered A-Tree is that an non-clustered A-Tree requires an additional "indirection layer" (called "Key Pages" in Figure 5.3). This layer is essentially an array of pointers that is the same size as the data but is sorted by the key that is being indexed. For example the first position in this indirection layer will contain a pointer to the smallest value of the key being indexed. Note that a secondary B+ tree that uses fixed sized paging also requires this indirection layer.

The first step in creating an non-clustered A-Tree is to build the indirection layer by sorting the data by the indexed key (e.g., temperature, age) and materializing the array of pointers to each data item in the sorted order. Next, like in the clustered index case, the segmentation algorithm scans the indirection layer and produces a valid set of segments that are then inserted into the upper level tree.

All operations on a non-clustered A-Tree internally operate on the indirection layer. For example, when looking up a key, the returned position of the data item is its position in the indirection layer. Then, to access the value, A-Tree then follows the pointer in the indirection layer at the predicted position.

The sorted level of key pages in a non-clustered A-Tree introduces additional overhead compared to a clustered A-Tree index but this overhead occurs in any non-clustered (secondary) index. However, as shown in our experiments, a non-clustered A-Tree is significantly smaller than a non-clustered B+ tree with fixed-size pages since it has fewer leaf and internal nodes.

5.2 Segmentation

In the following, we describe how A-Tree partitions the key space of an attribute into variable-sized segments that satisfy the specified error. After this process, each segment is inserted into a B+ tree to enable efficient insert and lookup operations, described further in Section 5.3 and Section 5.4.

5.2.1 Design Choices

A common objective function when fitting a function is to minimize the least square error (minimizing the second error norm E_2). Unfortunately, such an objective does not provide a guarantee for the maximal error and therefore does not provide a bound on the number of locations which must be scanned after interpolating a key's position. Therefore, our objective is to satisfy a maximal error (E_∞), demonstrated in Figure 5.4.

In addition to satisfying a maximal error, our segmentation process must also be able to efficiently

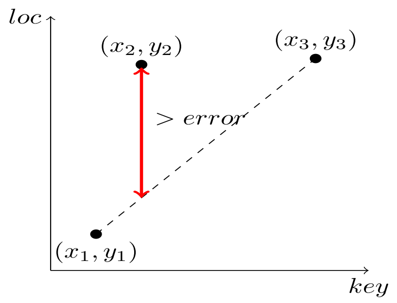


Figure 5.4: A segment from (x_1, y_1) to (x_3, y_3) is not valid if (x_2, y_2) is further than $error$ from the interpolated line.

segment very large datasets, since initially creating an index as well as inserting a new data item into a full segment requires segmenting the data. While several optimal (in the number of produced segments) piece-wise linear approximation techniques exist, most of these techniques are prohibitively expensive (e.g., a dynamic programming algorithm [85] has a runtime of $O(n^3)$ using $O(n)$ memory). Second, existing online piece-wise linear approximation algorithms [44, 80] have a high storage complexity and/or do not guarantee a maximal error, making them not a feasible solution for our problem. In contrast, our proposed segmentation algorithm (Section 5.2.2), is linear in runtime, uses a small constant amount of memory, and guarantees a maximal error in each segment. Therefore, to be able to efficiently (1) construct the index, and (2) support inserting new keys, we need a highly efficient one-pass linear algorithm. This focus on efficiency led us to choose linear piece-wise functions, since higher order approximations are often significantly more expensive to build and update.

In the following, we describe a proposed segmentation algorithm, similar to FSW [146, 91], which is linear in runtime, constant in memory use, and guarantees a maximal error. Importantly, though, we address (1) how to extend these techniques to indexing, including looking up and inserting data items, (2) prove that, in the worst case, segments are bounded in size, and (3) analyze the algorithm and show that the number of segments it produces is comparable to an optimal algorithm.

5.2.2 Segment Definition

As previously described, a segment is a region of the key space that can be represented by a linear function whereby all keys inside are within a bounded distance from their lineally interpolated position. The simplest way to define a segment is to select the first point (first key) and the last point (the last key) in the segment. Using this definition, we can fit a linear function to the locations of keys in the segment (using the start key, end key, and the number of positions).

Recall that our objective when segmenting the data is to satisfy a maximal error (i.e., a key's predicated position is at most $error$ number of elements away from its real position). This leads to an important property of a maximal segment (a segment is maximal when the addition of a key will violate the specified error) proved in Theorem 1.

Theorem 1. *The minimal number of locations covered by a maximal linear segment is $error + 1$.*

Proof. Consider 3 arbitrary points (x_1, y_1), (x_2, y_2), (x_3, y_3), where $x_1 < x_2 < x_3$ and $y_1 < y_2 < y_3$. By definition, the linear function starts at the first point in a segment, and ends at the last point in the segment. The linear segment is not feasible if the distance on the y axis (loc) is larger than the specified error. Therefore, given the 3 points, a linear segment starting at (x_1, y_1) and ending at x_3, y_3 is not feasible if:

$$y_2 - err > \frac{y_3 - y_1}{x_3 - x_1}(x_2 - x_1) + y_1 \tag{5.2}$$

By rearranging the inequality we get:

$$\begin{aligned} err &< y_2 - y_1 - \frac{y_3 - y_1}{x_3 - x_1}(x_2 - x_1) && (5.3)\\ &= (y_3 - y_1) \cdot \left(1 - \frac{x_2 - x_1}{x_3 - x_1}\right) - (y_3 - y_2) && (5.4)\\ &\leq (y_3 - y_1) \cdot \left(1 - \frac{x_2 - x_1}{x_3 - x_1}\right) - 1 && (5.5) \end{aligned}$$

In (5.4) y_3 was added and subtracted, and in (5.5) we use the fact that y_2 and y_3 are integers (thus $y_3 - y_2 \geq 1$). This provides a lower bound for the distance between the first point in a segment and the first point in the following segment:

$$\begin{aligned} y_3 - y_1 &> \frac{err + 1}{1 - \frac{x_2 - x_1}{x_3 - x_1}} = (err + 1) \cdot \frac{x_3 - x_1}{x_3 - x_2} > err + 1 && (5.6)\\ &\Rightarrow y_3 - y_1 > err + 1 && (5.7) \end{aligned}$$

Since (x_3, y_3) is the first point outside of the segment, the number of locations in the segment is $y_3 - 1 - y_1 \geq err + 1$. □

Due to the fact that minimum number of locations covered by a segment is related to the error, the total size of an A-Tree is also bounded. In the worst case, an A-Tree will never be larger than an index that uses fixed-sized pages of size *error*.

5.2.3 Segmentation Algorithm

Although an optimal algorithm produces the fewest number of segments, it is not practical for large datasets. Even linear time algorithms may not be efficient enough to be practical since multiplicative constants have a very significant practical effect. Therefore, to be able to efficiently (1) construct the index, and (2) support inserting new data items, we need a highly efficient one-pass linear algorithm.

We therefore present a greedy streaming algorithm ShrinkingCone (Algorithm 1), which, given a starting point (key) of a segment, attempts to maximize the length of a segment while satisfying

Algorithm 1 ShrinkingCone Segmentation

```
 1  $sl_{high} \leftarrow \infty$
 2  $sl_{low} \leftarrow 0$
 3  the first key is the segment origin
 4  for every $k \in$ keys (in increasing order)
 5      do if $k$ is inside the cone:
 6          then update $sl_{high}$
 7               update $sl_{low}$
 8          else key $k$ is the origin of a new segment
 9               $sl_{high} \leftarrow \infty$
10               $sl_{low} \leftarrow 0$
```

a given error threshold. ShrinkingCone is similar to FSW [91] but considers only monotonically increasing functions and can produce disjoint segments. The main idea behind ShrinkingCone is that a new key can be added to a segment if and only if it does not violate the error constraint of any previous key in the segment, assuming that it is the last key in the segment.

A cone is defined by the triple: origin point (the key and its location), high slope (sl_{high}), and low slope (sl_{low}). The combination of the starting point and the low slope gives the lower bound of the cone, and the combination of the starting point and the high slope gives the upper bound of the cone. The cone represents the family of feasible linear functions for a segment starting at the origin of the cone (the high and low slopes represent the range of valid slopes). When a new key is added to a segment, high and low slopes are calculated using the key and the key's position plus *error* and minus *error* (respectively). The lowest high slope and the highest low slopes are then selected (between the newly calculated and previous slopes), thus the cone either narrows (the high slope decreases and/or the low slope increases), or stays the same: these are the update steps in lines 6-7. If a new key to be added to the segment is outside of the cone, there must exist at least one previous key in the segment for which the error constraint will be violated. Therefore, a new key not inside the cone cannot be included in the segment, and becomes the origin point of the new segment.

Figure 5.5 shows how the cone is updated: point 1 is the origin of the cone. Point 2 updates both the high and low slope. Point 3 is inside the cone, however it only updates the upper bound of the cone (point 3 is less than *error* above the lower bound). Point 4 is outside of the updated cone, and therefore will be the first point of a new segment.

5.2.4 Algorithm Analysis

While the ShrinkingCone algorithm has a runtime of $O(n)$ and only uses a small constant amount memory (to keep track of the cone), it is not optimal. More than that, for a given maximal error and an adversarial data set the number of segments that it produces can be arbitrarily worse than an optimal algorithm, as we prove in 5.2.5. In the following, we first prove that ShrinkingCone is not competitive (i.e., it can be arbitrarily worse than an optimal solution). Then, although the

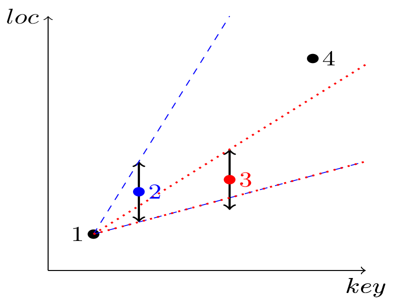


Figure 5.5: ShrinkingCone - Point 1 is the origin of the cone. Point 2 is then added, resulting in the dashed cone. Point 3 is added next, yielding in the dotted cone. Point 4 is outside the dotted cone and therefore starts a new segment.

algorithm is not optimal, we show that the algorithm yields results that are close to optimal using a variety of real-world datasets.

5.2.5 Competitive Analysis

We prove that ShrinkingCone can be arbitrarily worse than the optimal solution (not competitive)

Proof. Given the error threshold $E = 100$, consider the following input to ShrinkingCone:

1. 3 keys $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ where $y_1 = 1, y_2 = 2, y_3 = 3$ and $x_3 - x_2 = x_2 - x_1 = \frac{E}{2}$ (this is Step 1 in Figure 5.6).

2. The key $x_4 = x_3 + \frac{1}{E}$ repeated $E + 1$ times (using $E + 1$ consecutive locations), and the key $x_5 = x_4 + \frac{1}{E}$ without repetitions (using 1 location).

   After that repeat for $i \in [1, N]$ the following pattern: the key $x_{2(i+2)} = x_{2(i+2)-1} + E$ repeated $E + 1$ times, and a single appearance of the key $x_{2(i+2)+1} = x_{2(i+2)} + \frac{1}{E}$ (this is Step 2 in Figure 5.6).

3. The key $x_{2(N+1+2)} = x_{2(N+1+2)-1} + \frac{E}{2}$ (Step 3 in Figure 5.6).

The algorithm will then create the following segments (an illustration is shown in 5.6):

- $[x_1, x_4]$ (with slope $\frac{3}{E+\frac{1}{E}}$): adding the key x_5 will result in the slope $\frac{3+E+1}{E+\frac{2}{E}}$ which will not satisfy the error requirement for x_4, $y_1 + \frac{3+E+1}{E+\frac{2}{E}} \cdot (x_4 - x_1) - y_4 = 1 + \frac{3+E+1}{E+\frac{2}{E}} \cdot (E + \frac{1}{E}) - 4 = 100.98 > E$.

- Each of the next segments will contain exactly two keys (where the first key appears once, and the second key appears $E + 1$ times), since otherwise the error for the second key will be $\frac{1+E+1}{E+\frac{1}{E}} \cdot E - 1 = 100.98 > E$. Just like before, the $E + 1$ repetitions of a single key will cause a violation of the error (due to the spacing between subsequent keys).

Therefore, the algorithm will create $N + 2$ segments given this input.

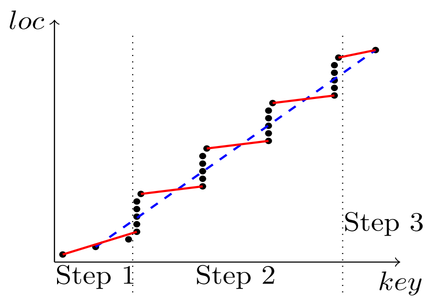


Figure 5.6: Competitive analysis sketch: the dots are the input, the dashed line is the optimal segmentation (the first dot is a segment), and the solid lines are the segments created by ShrinkingCone

On the other hand, the optimal algorithm will need only 2 segments: the first segment is the first key, and the second segment covers the rest of the input since the line starting at the second key and ending at the last key is never further away then E from any key, due to the construction of the input. The slope of the second segment will be $\frac{3+(N+1)\cdot(E+2)}{E+(N+1)\cdot(E+\frac{1}{E})}$, and the first key in the segment is about $\frac{E}{2}$ away on the x axis from the first repeated key. Since the repeated keys are spaced evenly (distance on the x axis of $E+\frac{1}{E}$), the linear function will not violate the error threshold for any key. An illustration is shown in 5.6.

Since N can be arbitrarily large, the algorithm is not competitive. $\square$

5.2.6 Real World Analysis

Although ShrinkingCone is not optimal, the maximum number of segments it produces is at most $\min\left(\frac{|keys|}{2}, \frac{|D|}{error+1}\right)$, where $|D|$ is the size of the dataset. This guarantee stems from Theorem 1: no input with less than 3 keys spanning at least $error+2$ positions will cause ShrinkingCone to create a new segment. Therefore, importantly, this allows us to bound the total size of A-Tree. More specifically, A-Tree will produce no more pages than a B+ tree that uses fixed-sized pages (of size $error$) in the worst case.

To evaluate ShrinkingCone, we implemented the optimal algorithm (runtime of $O(n^2)$ and memory consumption of $O(n^2)$) using 10^6 elements sampled from real-world datasets: NYC Taxi Dataset [106], OpenStreetMap [111], Weblogs [98], and IoT [98]. Table 5.1 shows the number of segments generated by the optimal algorithm and by ShrinkingCone. As shown, the number of segments that our algorithm yields is comparable to the number of segments in the optimal case for several error thresholds. For error thresholds for which we did not include results, the optimal algorithm exceeded the amount of available memory on a server with 768GB RAM.

Dataset	error	ShrinkingCone	Optimal	Ratio
Taxi drop lat	10	5358	4996	1.07
Taxi drop lat	100	351	271	1.29
Taxi drop lat	1000	51	48	1.06
Taxi drop lon	10	1198	1138	1.05
Taxi drop lon	100	371	325	1.14
Taxi drop lon	1000	40	37	1.08
Taxi pick time	10	6238	4359	1.43
Taxi pick time	100	165	137	1.2
OSM lon	10	7727	6027	1.28
OSM lon	100	101	63	1.6
Weblogs	10	16961	14179	1.2
Weblogs	100	909	642	1.42
IoT	10	8605	6945	1.24
IoT	100	723	572	1.26

Table 5.1: ShrinkingCone compared to optimal

5.3 Index Lookups

One of the most important operations of an index is to lookup a single key or a range of keys. However, since each entry in the leaf level of A-Tree points to a segment, performing a lookup requires first locating the segment that a key belongs to and then performing local search inside the segment. In the following, we first describe how A-Tree performs lookup operations for a single key and then show how we can extend this technique to range predicates.

5.3.1 Point Queries

The process of searching an A-Tree for a single element involves two steps: (1) searching the tree to find the segment that the element belongs to, and (2) finding the element within a segment. These steps are outlined in Algorithm 2.

Tree Search

Since, as previously described, each segment is stored in a standard B+ tree (with its starting position as the key and the segment's slope and a pointer to the table page as its value), we must first search the tree to find the segment that a given key belongs to. To do this, we begin traversing the B+ tree from the root to the leaf, using standard tree traversal algorithms. More specifically, starting from the root of the tree, we recursively follow the pointer to the child node whose range includes the key to be located. These steps, outlined in the SearchTree function of Algorithm 2, terminate when reaching a leaf node which points to the segment that contains the key.

Since the B+ tree is used to index every segment, the runtime for searching for the segment that a key belongs in is $O(log_b(p))$, where b is the fanout of the tree (i.e., number of separators inside a node) and p is the number of segments created during the segmentation process.

Algorithm 2 Lookup Algorithm

```
LOOKUP($tree, key$)
1  $seg \leftarrow$ SearchTree ($tree.root, key$)
2  $val \leftarrow$ SearchSegment ($seg, key$)
3  return $val$

SEARCHTREE($node, key$)
1  $i \leftarrow 0$
2  while $key < node.keys[i]$
3        do $i \leftarrow i + 1$
4  if $node.value[i].isLeaf()$
5     then $j \leftarrow 0$
6           while $key < node.values[j]$
7                do $j \leftarrow j + 1$
8           return $node.values[j]$
9  return SearchTree($node.values[i], key$)

SEARCHSEGMENT($seg, key$)
1  $pos \leftarrow (key - seg.start) \times seg.slope$
2  return BinarySearch($seg.data, pos - error, pos + error, key$)
```

Segment Search

Once the segment for a key has been located, A-Tree must find the key's position inside the segment. Recall that segments are created such that an element is no more than a constant distance (*error*) from the element's position determined through linear interpolation. Other techniques for interpolation search inside a fixed-sized index page are discussed in [51].

To compute the approximate location of a key k within a given segment s, we subtract the key from the first key that appears in the segment $s.start$. Then, we multiply the difference by the segment's slope $s.slope$, as shown below in the following equation.

$$pred\_pos = (k - s.start) \times s.slope \tag{5.8}$$

After interpolating the element's position, the true position of an element is guaranteed to be within the error threshold. A-Tree locally searches the following region using binary search, although any search algorithm, including linear search, binary search, or exponential search can also be used.

$$true\_pos \in [pred\_pos - error, pred\_pos + error] \tag{5.9}$$

Since segments satisfy the specified error condition, the cost of searching for an element inside a segment is bounded.

5.3.2 Range Queries

Range queries, unlike point queries, have the additional requirement that they examine every item in the specified range. Therefore, for range queries, the selectivity of the query (i.e., the number of tuples that satisfy the query's predicate) has a large influence on the total runtime of the query.

However, like point queries, range queries must also find a single tuple: either the start or the end of the range. Therefore, A-Tree uses the previously described point query lookup techniques in order to find the beginning of the specified range. Then, since segments either store keys contiguously (clustered index) or have an indirection layer with pointers that is sorted by the key (non-clustered index), A-Tree can simply scan from the starting location until a key is found that lies outside of the specified range. For a clustered index, scanning the relevant range performs only sequential access, while for a non-clustered index, range queries requires random memory accesses (which is true for any non-clustered index).

5.4 Index Inserts

Along with locating an element, an index needs to be able to handle insert operations. In some applications, maintaining a strict ordering guarantee is necessary, in which case A-Tree should ensure that new items are inserted in-place. However, in situations where this is not the case, we've developed a more efficient insert strategy that improves insert throughput. In the following, we discuss each of these strategies for inserting new items into an A-Tree. Then, in Section 5.6.1, we show how these strategies compare for various workloads and parameters.

5.4.1 In-place Insert Strategy

In a typical B+ tree that uses paging, pages are left partially filled and new values are inserted into an empty slot using an in-place strategy. When a given page is full, the node is split into two nodes, and the changes are propagated up the tree (i.e., the inner nodes in the tree are updated).

Although similar, inserting an item into an A-Tree requires additional consideration since any key in the segment must be at most the specified error amount ($error$) away from its interpolated position. Importantly, in-place inserts require moving keys in the page to preserve the order of the keys.

Without any a priori knowledge about the error of a given key, any attempt to move the key requires a check to guarantee that the error satisfied. To make matters worse, a single insert may require moving many keys (in the worst case, all keys in the page) to maintain the sorted order. Thus, we must have a priori knowledge about any given key to determine if it can be moved in any direction while preserving the error guarantee.

Similar to the fill factor of a page, we divide the specified $error$ in 2 parts: the segmentation error e (error used to segment the data), and an insert budget ε (number of locations any key can be moved in either direction). To preserve the specified error, we require that $error = e + \varepsilon$. By

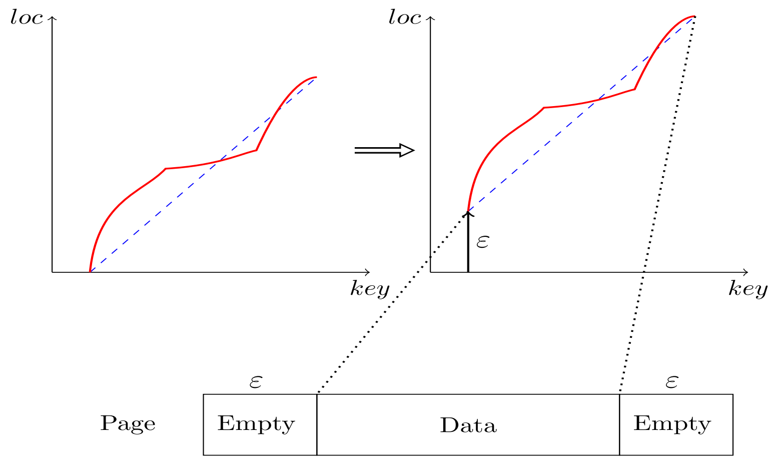


Figure 5.7: Page layout - data is offset by ε locations with ε empty spaces before and after the data

keeping an insert budget for each page, A-Tree is able to ensure that inserting a new element will not violate the error for the page.

Based on this formulation, pages are constructed in the following manner. Given a segment s, the page has a total size of $|s| + 2\varepsilon$ ($|s|$ is the number of locations in the segment). The data is placed in the middle of the new page, yielding ε empty locations at the beginning and end of the page. Figure 5.7 shows the resulting page for a single segment.

With this strategy, it is possible to move any key in a direction which has free space without violating the error condition. Therefore, to insert a new item using an in-place insert strategy, A-Tree first locates the position in the page where the new item belongs. Then, depending on which end of the page (left or right) is closer, all elements are shifted (either left or right) into the empty region of the page. Once all of the empty spaces are filled, the segment needs to be re-approximated (using the segmentation algorithm described in Section 5.2). If the segmentation algorithm produces more than one segment, we create n new segments (where n is the number of segments produced after running Algorithm 1 on the single segment that is now full). Finally, each new segment is inserted into the upper level tree, and any references to the old segment are deleted.

5.4.2 Delta Insert Strategy

Since the segments in A-Tree are of variable size, the cost of an insert operation using the previously described in-place insertion strategy can be high, particularly for large error thresholds or uniform data that produces large segments with many data items. Specifically, on average, $\frac{|s|}{2}$ keys may need to be moved for an insert operation, where $|s|$ is the number of locations in the segment. Therefore, a better approach for inserts in A-Tree should amortize the cost of moving keys in the segment.

To reduce the overhead of moving data items inside a page when inserting a new item, each segment in an A-Tree contains an additional fixed-sized buffer instead of extra space at each end.

Algorithm 3 Delta Insert Algorithm

```
InsertKey($tree, seg, key$)
1  $seg \leftarrow$ SearchTree ($tree.root, key$)
2  $seg.buffer.insert(key)$
3  if $seg.buffer.isFull()$
4     then
5           $segs =$ SEGMENTATION($seg.data, seg.buffer$)
6           for $s \in segs$
7                do $tree.insert(s)$
8           $tree.remove(seg)$
9  return
```

More specifically, as shown in Algorithm 3, new keys are added to the buffer portion of the segment for which the key belongs to (line 2). This buffer is kept sorted to enable efficient search and merge operations.

Once the buffer reaches its predetermined size ($buffer\_size$), the buffer is combined with the data in the segment and then re-segmented using the previously described segmentation algorithm (Algorithm 1) to create a series of valid segments that satisfy the error threshold (line 6). Note that depending on the data, the number of segments generated after this process can be one (i.e., the data inserted into the buffer does not violate the error threshold) or several. Finally, each of the new segments generated from the segmentation process are inserted into the tree (line 6-7) and the old page is removed (line 8).

Storing additional data inside a segment impacts how to locate a given item, as well as how the error is defined. More specifically, searching for an item requires searching the segment's data (as described in the SearchSegment function in Algorithm 3) as well as searching for the item in the buffer. Since adding a buffer for each segment can violate the error guarantees that A-Tree provides, we transparently incorporate the buffer's size into the error parameter for the segmentation process. More formally, given a specified error of $error$, we transparently set the error threshold for the segmentation process to ($error - buffer\_size$). This ensures that a lookup operation will satisfy the specified error even if the element is located in the buffer.

The overall runtime for inserting a new element into an A-Tree is the time required to locate the segment and add the element to the segment's buffer. With p pages stored in an A-Tree, and a fanout of (i.e., number of keys in each internal separator node), inserting a new key has the following runtime:

$$insert\ runtime : O(log_b p) + O(buffer\_size) \tag{5.10}$$

Note, when the buffer is full and the segment needs to be re-segmented, the runtime has an additional cost of $O(d)$, where d is the sum of a segment's data and buffer size. Additionally, if the write-rate is very high, we could also support merging algorithms that use a second buffer similar to how column stores merge a write-optimized delta to the main compressed column. However, this is an orthogonal consideration that heavily depends on the read/write ratio of a workload and is outside the scope of this paper.

5.5 Cost Model

Since the specified error threshold affects both the performance of lookups and inserts as well as the index's size, the natural question follows: how should a DBA pick the error threshold for a given workload? To navigate this tradeoff, we provide a cost model that helps a DBA pick a "good" error threshold when creating an A-Tree. At a high level, there are two main objectives that a DBA can optimize: performance (i.e., lookup latency) and space consumption [12]. Therefore, we present two ways to apply our cost model that help a DBA choose an error threshold.

5.5.1 Latency Guarantee

For a given workload, it is valuable to be able to provide latency guarantees to an application. For example, an application may require that lookups take no more than a specified time threshold (e.g., 1000ns) due to SLAs or application-specific requirements (e.g., for interactive applications). Since A-Tree incorporates an error term that in turn affects performance, we can model the index's latency in order to pick an error threshold that satisfies the specified latency requirement.

As previously mentioned, lookups require first finding the relevant segment and then searching the segment (data and buffer) for the element. Since the error threshold influences the number of segments that are created (i.e., a smaller error threshold yields more segments), we can use a function that returns the number of segments that are needed to satisfy a given error threshold. This function can either be learned for a specific dataset (i.e., segment the data using different error thresholds) or a general function can be used (e.g., make the simplifying assumption that the number of segments decreases linearly as the error increases). We use S_e to represent the number of resulting segments for given dataset using an error threshold of e.

Therefore, the total estimated lookup latency for an error threshold of e can be modeled by the following expression, where b is the tree's fanout, *buff* is a segment's maximum buffer size, and c is a constant representing the latency (in ns) of a cache miss on the given hardware (e.g., $50ns$). Moreover, the cost function assumes binary search for the area that needs to be searched within a segment bounded by e as well as searching the complete buffer.

$$\textsc{latency}(e) = c\,\Big[\underbrace{log_b(S_e)}_{\text{Tree Search}} + \underbrace{log_2(e)}_{\text{Segment Search}} + \underbrace{log_2(buff)}_{\text{Buffer Search}}\Big] \tag{5.11}$$

Setting c to a constant value implies that all random memory accesses have a constant penalty but caching can often change the penalty for a random access. In theory, instead of being a constant, c could be a function that returns the penalty of a random access but we make the simplifying that c is a constant.

Using this cost estimate, the index with the smallest storage footprint that satisfies the given latency requirement L_{req} (in nanoseconds) is given by the following expression, where E represents a set of possible error values (e.g., $E = \{10, 100, 1000\}$) and SIZE is a function that returns the estimated size of an index (defined in the next section).

$$e = \underset{\{e \in E \ \mid \text{LATENCY}(e) \ \leq \ L_{req}\}}{\arg\min} \big(\text{SIZE}(e)\big) \tag{5.12}$$

In addition to modeling the latency for lookup operations, we can similarly model the latency for insert operations. However, there are a few important differences. First, inserts do not have to probe the segment. Also, instead of searching a segment's buffer, inserts require adding the item to the buffer in sorted order. Finally, we must also consider the cost associated with splitting a full segment.

5.5.2 Space Budget

Instead of specifying a lookup latency bound, a DBA can also give A-Tree a storage budget to use. In this case, the goal becomes to provide the highest performance (i.e., lowest latency for lookups and inserts) while not exceeding the specified storage budget.

More formally, we can estimate the size of a read-only clustered index (in bytes) for a given error threshold of e using the following function, where again S_e is the number of segments that are created for an error threshold of e, and b is the fanout of the tree.

$$\text{SIZE}(e) = \underbrace{(S_e \cdot log_b(S_e) \cdot 16B)}_{\text{Tree}} + \underbrace{(S_e \cdot 24B)}_{\text{Segment}} \tag{5.13}$$

The first term is a pessimistic bound on the storage cost of the tree (leaf + internal nodes using 8 byte keys/pointers), while the second term represents the added metadata about each segment (i.e., each segment has a starting key, slope, and pointer to the underlying data, each 8 bytes).

Therefore, the smallest error threshold that satisfies a given storage budget S_{req} (given in bytes) is given by the following expression where again E represents a set of all possible error values (e.g., $E = \{10, 100, 1000\}$).

$$e = \underset{\{e \in E \ \mid \text{SIZE}(e) \ \leq \ S_{req}\}}{\arg\min} \big(\text{Latency}(e)\big) \tag{5.14}$$

As we show in Section 5.6.7, our cost model can accurately estimate the size of an A-Tree over real-world datasets, providing DBAs with a valuable way to balance performance (i.e., latency) with the storage footprint of an A-Tree.

5.6 Evaluation

This section evaluates A-Tree and the presented techniques. Overall, we see that A-Tree achieves comparable performance to both a full index as well as indexes that use fixed-sized paging while using orders of magnitude less space.

First, in Section 5.6.1, we compare the overall performance of A-Tree, measuring its lookup and insert performance for both clustered and non-clustered indexes using a variety of real-world datasets. Next, we compare the two proposed insert strategies in Section 5.6.2. Then, in Section 5.6.3 we show how A-Tree can leverage other index structures to further improve performance. Section 5.6.4 presents results for range queries. Then, Section 5.6.5 and Section 5.6.6 show how A-Tree performs for an adversarial synthetically generated dataset (i.e., worst-case data distribution) as well as the scalability of our index over different dataset sizes. In Section 5.6.7, we show that our cost model can accurately pick an error threshold that satisfies a specified lookup latency or space consumption requirement. Section 5.6.8 and Section 5.6.9 show a breakdown of lookup time as well as how A-Tree performs for various buffer fill factor values. Finally, in Section 5.6.10 we compare A-Tree to Correlation Maps and in Section 5.6.11 we measure the construction cost of A-Tree

All experiments were conducted on a single server with an Intel E5-2660 CPU (2.2GHz, 10 cores, 25MB L3 cache) and 256GB RAM and all index and table data was held in memory.

5.6.1 Exp. 1: Overall Performance

In the following, we evaluate the overall lookup and insert performance of A-Tree. For these comparisons, we benchmark A-Tree against both a full index (i.e, a dense index) as well as an index that uses fixed size pages (i.e., a sparse index) as baselines. A full index can be seen as best case baseline for lookup performance and thus gives us an interesting reference point. As this paper was written without involvement of Google, the authors had no access to the code of [84] and thus could not use it as another baseline.

For the two baselines (full and fixed-sized paging), we use a popular B+ tree implementation (STX-tree [136] v0.9) since our A-Tree prototype also uses this tree to index the variable sized segments. Importantly, as we show in Section 5.6.3 any other tree implementation can also serve as the organization layer under A-Tree. For example, a tree that compresses internal nodes can be used to index our generated segments and will still provide benefits (i.e., compressing the data that A-Tree stores in its underlying tree can improve performance), but this benefit is orthogonal to the benefits that A-Tree achieves through its approximate and variable sized segmentation of the underlying data.

Datasets

Since performance of our index depends on the distributions of elements in a given dataset, we evaluate A-Tree on real-world datasets with different distributions. Later, in Section 5.6.5, we show that our techniques are still valuable using a synthetically generated worst-case dataset. For

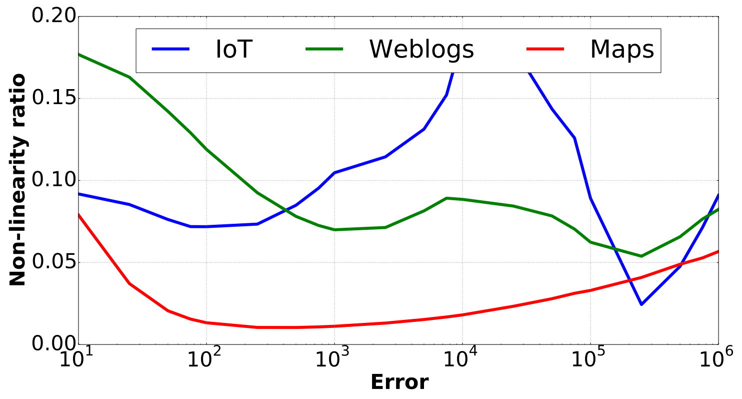


Figure 5.8: Non-linearity

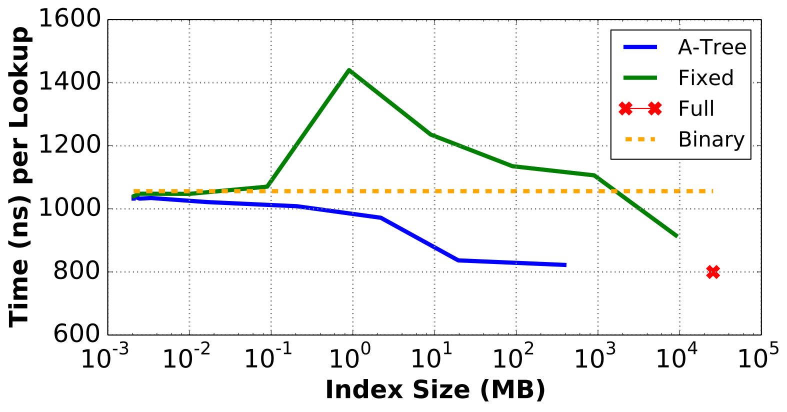
our evaluation, we use three different real-world datasets, each with very different underlying data distributions: (1) Weblogs [98], (2) IoT [98], and (3) Maps [111].

The Weblogs dataset contains $\approx 715M$ log entries for every web request to the computer science department at a university over the past 14 years. This dataset contains subtle trends, such as the fact that more requests occur during certain times (e.g., school year vs summer, daytime vs night time). On the other hand, the IoT dataset contains $\approx 5M$ readings from around 100 different IoT sensors (e.g., door, motion, power) installed throughout an academic building at a university. Since these sensors generally reflect human activity, this dataset has interesting patterns, such as the fact there is more activity during certain hours because classes are in session. For each of these datasets, we create a *clustered* A-Tree using the timestamp attribute which represents the time at which a resource was requested. Finally, the Maps dataset contains the longitude of $\approx 2B$ features (e.g., museums, coffee shops) across the world. Unsurprisingly, the longitude of locations is relatively linear and does not contain many periodic trends. Unlike the previous two datasets, we create a *non-clustered* A-Tree over the longitude attribute of this dataset.

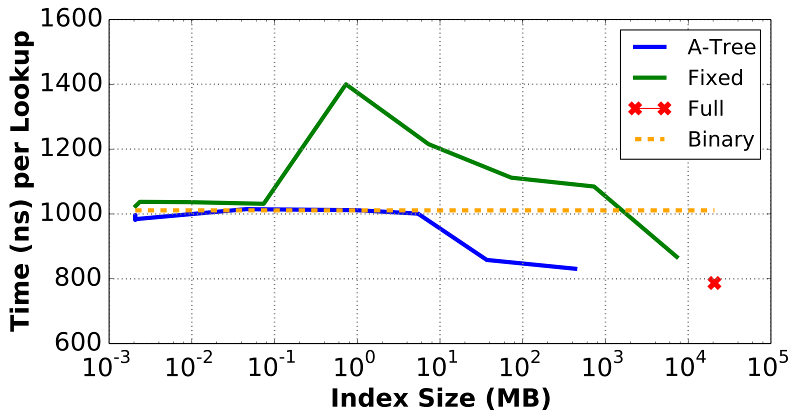
For our approach, the most important aspect of a dataset that impacts A-Tree's performance is the data's periodicity. For now, think of the periodicity as the distance between two "bumps" in a stepwise function that maps keys to storage locations as shown as in Figure 5.14a (blue line). If the error of our index is larger than the periodicity (green line), the segmentation results in a single segment. However, if the error is smaller than the periodicity (red line), we need multiple segments to approximate the data distribution.

Therefore, we use the *non-linearity ratio* to show the periodicity of a dataset. To compute this ratio, we first calculate the number of segments required to cover the dataset for a given error threshold. We then normalize this result by the number of segments required for a dataset of the same size with periodicity equal to the error (which is the worst case, or the most "non-linear" in that scale).

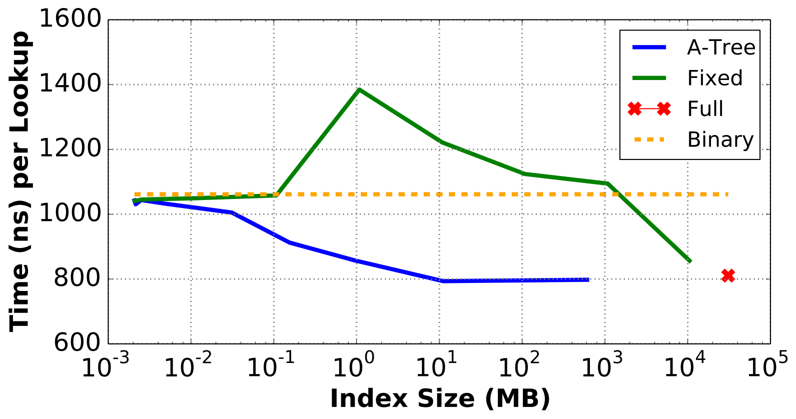
To show that all datasets contain a distinct periodicity pattern, Figure 5.8 plots the non-linearity



(a) Weblogs



(b) IoT



(c) Maps

Figure 5.9: Latency for Lookups (per thread)

ratio of each of dataset. The IoT dataset has a very significant bump, signifying that there is very strong periodicity the scale of 10^4, likely due to patterns that follow human behavior (e.g., day/night hours). Weblogs has multiple bumps which are likely correlated to different periodic patterns (e.g., daily, weekly, and yearly patterns). The Maps dataset, unlike the others, is linear at small scales (but has stronger periodicity at larger scales).

Lookups

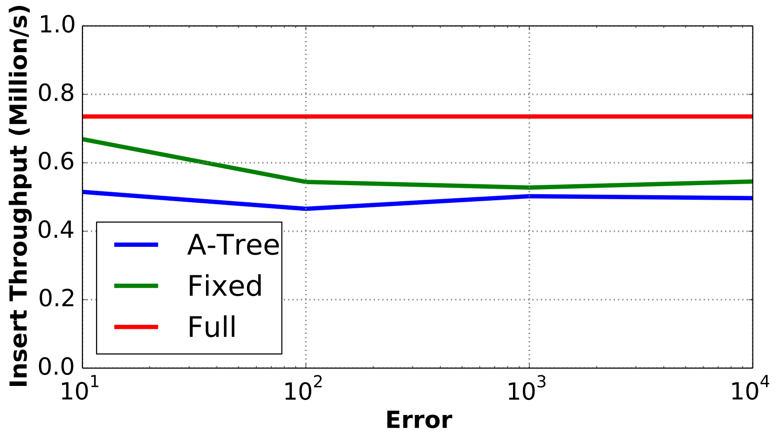
The first series of benchmarks show how A-Tree compares to (1) a full index (i.e., dense), (2) an index that uses fixed-sized paging (i.e., sparse), and (3) binary search over the entire dataset. We include binary search since it represents the most extreme case where the error is equal to the data size (i.e., our segmentation creates one segment). For the Weblog and IoT dataset, we created a clustered index using the timestamp attribute which is the primary key of these datasets. We created a non-clustered (i.e., secondary) index over the longitude attribute of the Maps dataset, which is not unique.

The results (Figure 5.9) show the lookup latency for various index sizes for the Weblogs (scaled to 1.5B records), IoT (scaled to 1.5B records), and Maps (not scaled, 2B records) datasets. More specifically, since the size of both A-Tree and the fixed-size paging baseline can be varied (i.e., A-Tree's error term and the page size influence the number of indexed keys), we show how the performance of each of these approaches scales with the size of the index. Note that the size of a full index cannot be varied and is therefore a single point in the plot. Additionally, since binary search does not have any additional storage requirement, its size is zero but is visualized as a dotted line.

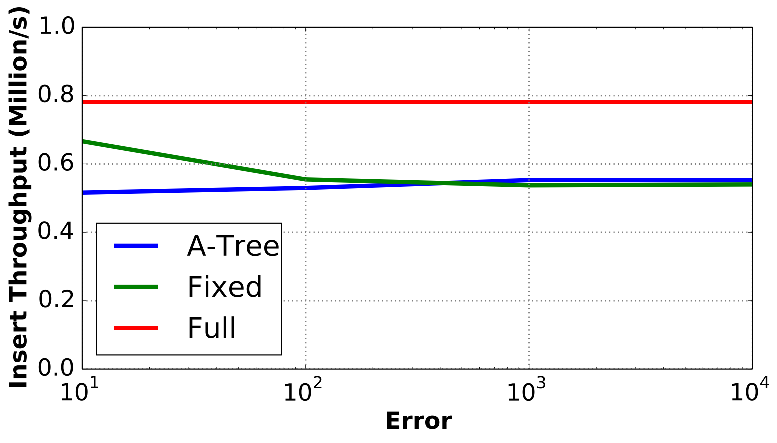
In general, the results show that A-Tree always performs better than an index that uses fixed-sized paging. Most importantly, however, A-Tree offers significant space savings compared to both fixed-sized paging and a full index. For example, in the Maps dataset, A-Tree is able to match the performance of a full index using only $609MB$ of memory, while a full index consumes over $30GB$ of space. Moreover, compared to a tree that uses fixed-sized paging, an A-Tree which consumes only $1MB$ of memory is able to match the performance of a fixed-sized index which consumes over $10GB$ of memory, offering a space savings of four orders of magnitude.

Furthermore, as expected, for very small index sizes (i.e., very large page sizes in fixed-sized paging and a high error threshold in A-Tree), both A-Tree and fixed-sized paging mimic the performance of binary search since there are only a few pages that are searched using binary search. On the other hand, as the index grows, the performance of both fixed-sized paging as well as A-Tree converge to that of a full index due to the fact that pages contain very few elements. Note that the spike in the graph for the fixed-sized index is due to the fact that the index begins to fall out of the CPU's L2 cache.

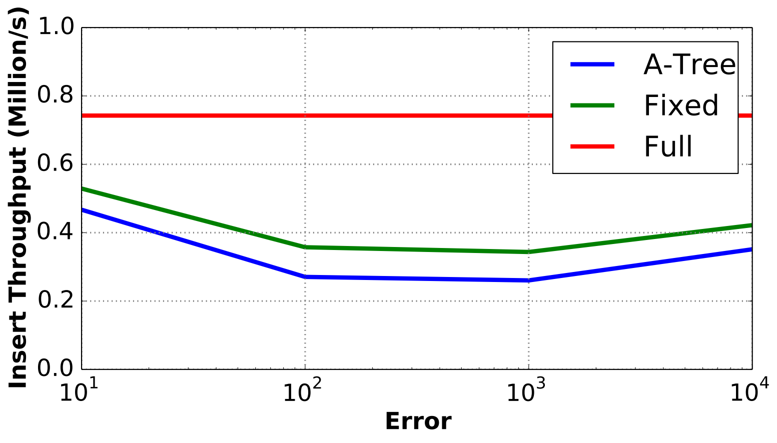
Finally, as expected, the data distribution impacts the performance of A-Tree. More specifically, we can see that A-Tree can quickly match the performance of a full tree with the Maps dataset, compared to the Weblogs and IoT datasets. This is due to the fact that the Maps dataset is relatively linear, when compared to the Weblogs and IoT datasets as shown in Figure 5.8.



(a) Weblogs



(b) IoT



(c) Maps

Figure 5.10: Throughput for Inserts (per thread)

Inserts

Next, we compare the insert performance of A-Tree to both a full index, as well as an index that uses fixed-sized paging as previously described. To insert new items, A-Tree uses the previously described delta insert technique since it provides the best overall performance, which we show in next section where we perform an in-depth comparison of the delta and the in-place insert strategies.

More specifically, to ensure a fair comparison and that A-Tree is not unfairly write-optimized, we set the size of the delta buffer to half of the specified error (i.e, for an error threshold of 100, the underlying data is segmented using an error of 50 and each segment's buffer has a maximum size of 50 elements). Similarly, for the index with fixed-sized pages, the page size is given by the half of the error threshold we used for A-Tree and the same amount (i.e., half of the error used in A-Tree) is used as the buffer size. As usual, once the buffer is full, the page is split into two pages.

The results, shown in Figure 5.10, compare each of the index's insert throughput for various error thresholds. As shown, A-Tree is able to achieve insert performance that is, in general, comparable to an index that uses fixed-sized paging. Unsurprisingly, a full B+ tree is able to handle a higher write load than either an A-Tree or an index that uses fixed-sized paging since both need to periodically split pages that become full. Additionally, A-Tree needs to execute the segmentation algorithm, explaining the performance gap between A-Tree and fixed-sized paging.

Interestingly, however, A-Tree is faster than fixed-sized paging in some cases.For a large error, there typically are fewer segments generated, which reduces the number of times that A-Tree needs to merge the buffer with the segment's data and execute the segmentation algorithm.

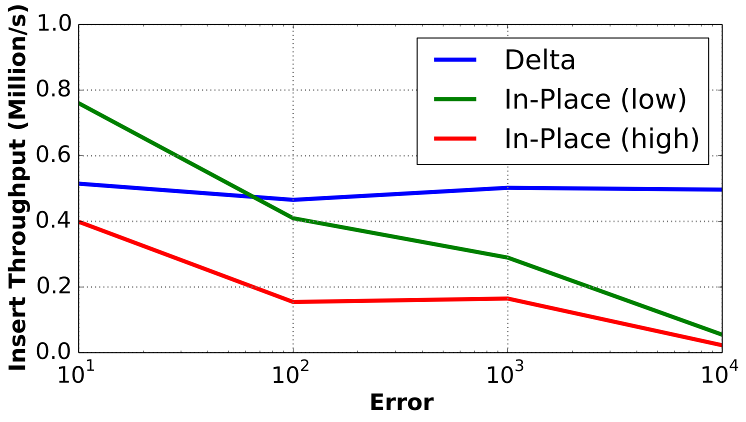
5.6.2 Exp. 2: In-place vs Delta Inserts

In the following, we compare the two insert strategies described in Section 5.4 and show how they perform for various datasets, fill factors, and error thresholds.

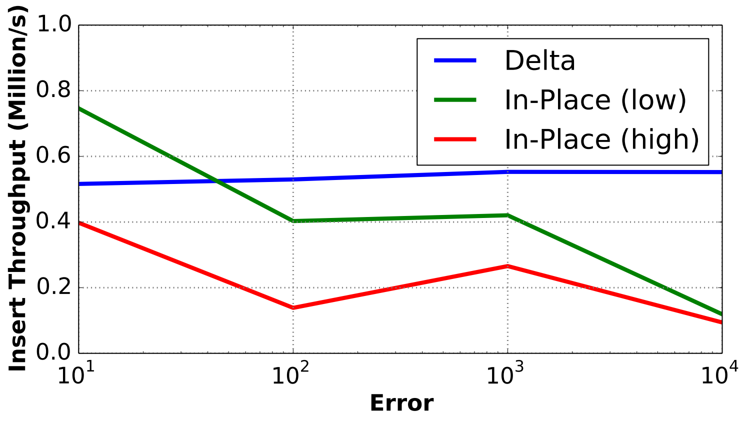
Figure 5.11 shows the insert throughput for the delta and in-place insert strategies for each of the three described datasets. As previously described, for in-place inserts, the amount of free space reserved at the beginning and end of page (ε) can be tuned based on characteristics of the workload. Therefore, we show results for both a low (i.e., ε is 25% of the error) and high (i.e., ε is 75% of the error) fill factor. For the reported delta insert results, we use the same setup as the previously described insert experiment (i.e., we set the size of the delta buffer to half of the specified error).

As shown, the delta insert strategy generally offers the highest insert throughput for error thresholds higher than 100. For higher error thresholds, the in-place insert strategy must move a significant number data items when inserting a new item since segments created with a higher error threshold contain more keys. However, for low error thresholds, the in-place insert strategy outperforms the delta strategy, since there are significantly fewer data items that need to be shifted.

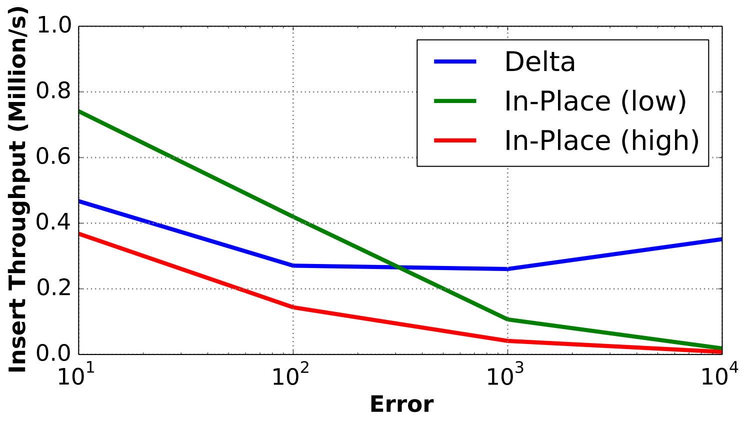
The fill factor impacts (1) how many data items are in a given segment and must be copied when inserting a new element, and (2) how often a segment fills up and needs to be re-segmented. As shown, in general, the in-place strategy with a low fill factor offers the highest insert performance.



(a) Weblogs



(b) IoT



(c) Maps

Figure 5.11: Insertion Strategy Microbenchmark

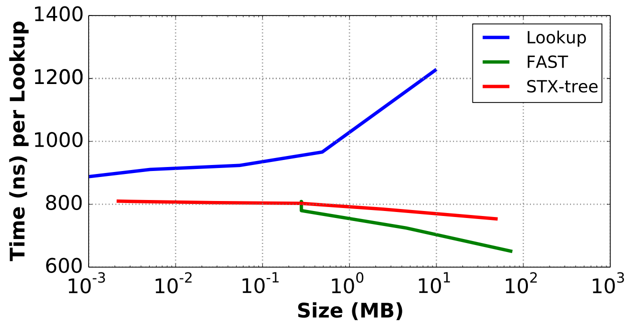


Figure 5.12: Internal Organization

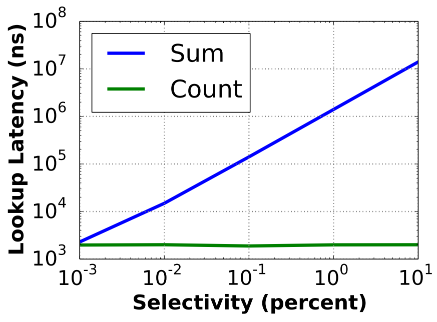


Figure 5.13: Range Queries

5.6.3 Exp 3: Other Indexes

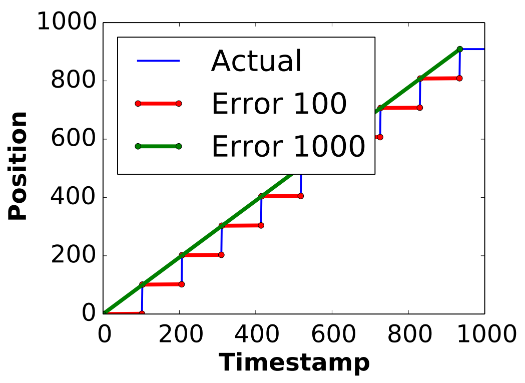
As previously mentioned, A-Tree internally uses STX-tree [136] by default to organize the variable sized pages generated through our segmentation process (Section 5.2). However, depending on the workload characteristics, other internal data structures could also be used to index segments.

Therefore, to show that our techniques are generalizable, we compare using an STX-tree to (1) FAST [82], a highly optimized index structure for read-only workloads and (2) a simple lookup table that simply stores the variable sized pages in sorted order. Figure 5.12 shows the results for various index sizes (i.e., error thresholds) over a subset of the Weblogs dataset (since FAST requires a power of two elements). As shown, a lookup table provides superior space savings (since there are no internal separator nodes) but with a higher lookup latency. On the other hand, an A-Tree that internally uses the highly optimized FAST index can provide faster lookups but often consumes more space. Therefore, an A-Tree is able to leverage alternative indexes that may be more performant depending on the workload characteristics (i.e., read-only).

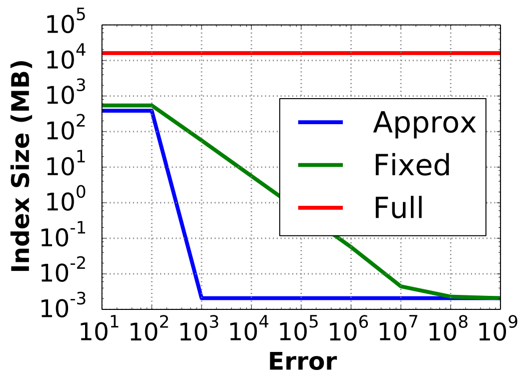
5.6.4 Exp 4: Range Queries

In addition to point queries, A-Tree also supports range queries whereby an arbitrary number of tuples must be examined to compute the result. Figure 5.13 shows the performance of an A-Tree for range queries for both a sum and count aggregate for various selectivities for the Weblogs dataset.

Interestingly, to compute the result for a count query, an A-Tree only needs to subtract the starting position from the ending position of the range (i.e., a count aggregate over a range is essentially two point lookups), resulting in a constant lookup latency. On the other hand, computing the sum of an attribute over a range requires examining every tuple in the range, resulting in significantly more work for larger ranges.



(a) **Worst Case Data**



(b) **Worst Case Index Size**

Figure 5.14: Worst Case Analysis

5.6.5 Exp. 5: Worst Case Analysis

Since the data distribution influences the performance of A-Tree, we synthetically generated data to illustrate how our index performs with data that represents a worst-case. To do this, we generate data using a step function with a fixed step size of 100, as shown in Figure 5.14a. Since the step size is fixed, an error threshold less than the step size yield a single segment per step. However, given an error threshold larger than the step size, our segmentation algorithm will be able to use a single segment to represent the entire dataset.

Figure 5.14b shows the performance for various sizes of each index built over this worst case dataset. As shown, for error thresholds of less than 100, the size of an A-Tree is the same as a fixed-sized index but still smaller than a full index. This is due to the fact that for the error thresholds less than the step size, A-Tree creates segments of size 100 (step size), resulting in a large number of nodes in the tree. On the other hand, for an error threshold of larger than 100, A-Tree is able to represent the step dataset with only a single segment, dramatically reducing the index's size. As shown, importantly, an A-Tree will never contain more leaf nodes than an index that uses fixed-sized paging.

5.6.6 Exp. 6: Data Size Scalability

To evaluate how A-Tree performs for various dataset sizes, we measure the lookup latency for the Weblogs dataset using various scale factors where both the error threshold and fixed-page size are set to 100, which is optimal for this dataset. Since the performance of our index depends on the underlying data distribution, we scale the dataset while maintaining the underlying trends. We omit the result for all other datasets here (IoT and Maps) since they follow similar trends.

Figure 5.15 shows that the tree-based approaches (i.e., A-Tree, a full index, fixed-sized paging) scale better than binary search due to the better theoretical asymptotic runtime ($log_b(n)$ vs $log_2(n)$) and cache performance. Additionally, A-Tree's performance over various dataset sizes closely

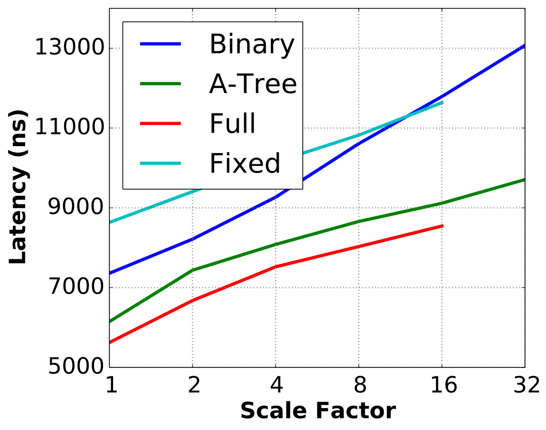


Figure 5.15: Scalability

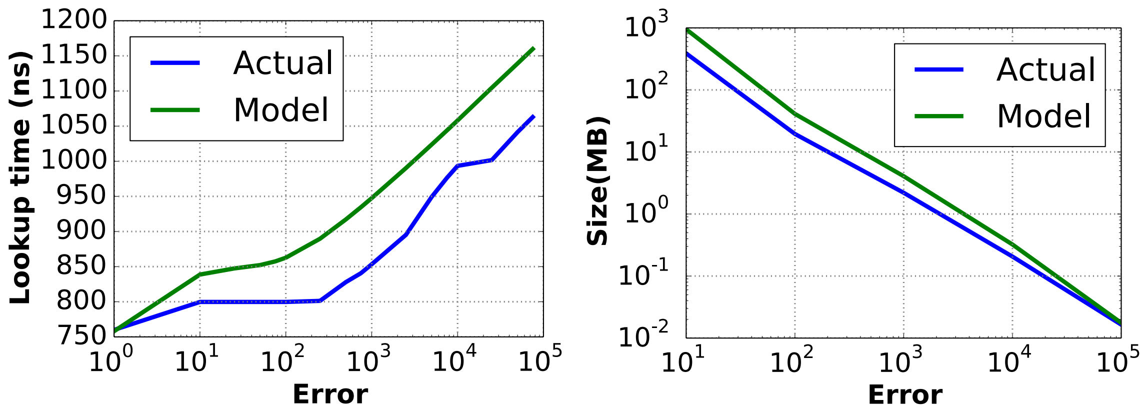
follows that of a full index which offers the best performance, demonstrating that our techniques offer valuable space savings without sacrificing performance. Most importantly, neither a full index nor an index that uses fixed-sized paging could scale to a scale factor of 32 since the size of the index exceeded the amount of available memory, which again shows that A-Tree is able to offer valuable space savings.

5.6.7 Exp. 7: Accuracy of Cost Model

Since, as previously described, the error threshold influences both the latency as well as space consumption of our index, our cost model presented in Section 5.5 aims to guide a DBA when determining what error threshold to use for an A-Tree. More specifically, given a latency requirement (e.g., $1000ns$) or a space budget (e.g., $2GB$), our cost model automatically determines an appropriate error threshold that satisfies the given constraint.

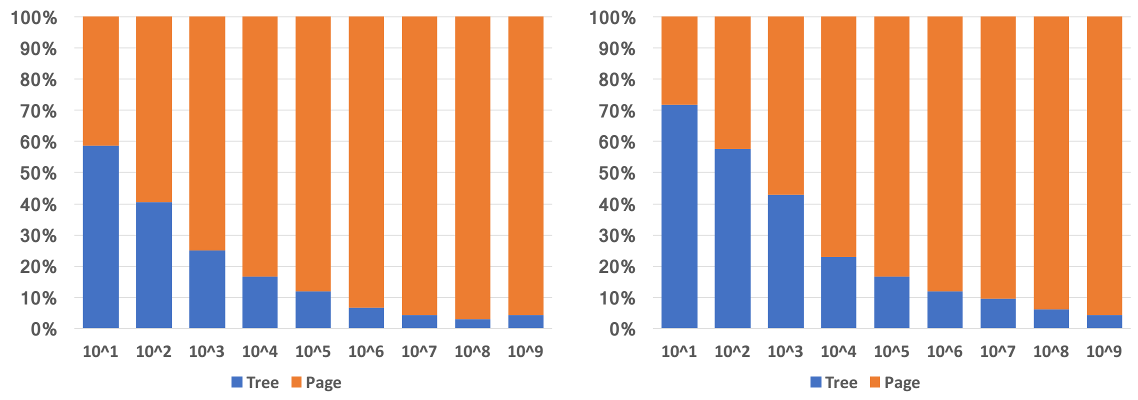
Figure 5.16a shows the estimated and actual lookup latency for various error thresholds on the Weblogs dataset using a value of $50ns$ for c (the cost of a random memory access) determined through a memory benchmarking tool on the given hardware. As shown, our latency model predicts an accurate upper bound for the actual latency of a lookup operation. Our model slightly overestimates the latency since it does not incorporate CPU caching effects. Since we overestimate the cost, we ensure that a specified latency threshold will always be observed.

To evaluate our size cost model, we show the predicted and actual size of an A-Tree for various error thresholds in Figure 5.16b. As shown, our model is able to accurately predict the size of an index for a given error threshold while ensuring that our estimates are pessimistic (i.e., the estimated cost higher than the true cost).



(a) Latency (b) Size

Figure 5.16: Cost Model Accuracy



(a) A-Tree (b) **Fixed-sized Index**

Figure 5.17: Lookup Breakdown

5.6.8 Exp. 8: Lookup Breakdown

A lookup operation, as described in Section 5.3 involves two steps (i.e., locating the segment where a key belong to and the searching the segment's data in order to find the item within the segment). Therefore, we examine the amount of time spent in each of these two steps for A-Tree as well as an index that uses fixed-sized paging for various error thresholds.

The results, presented in Figure 5.17, show that in both cases the majority of time is spent searching the tree to find the page where the data item belongs for smaller error thresholds (and page sizes),. Since A-Tree is able to leverage properties of the underlying data distribution in order to create variable sized segments, the resulting tree is significantly smaller. Therefore, A-Tree spends less time searching the tree to find the corresponding segment for a given key.

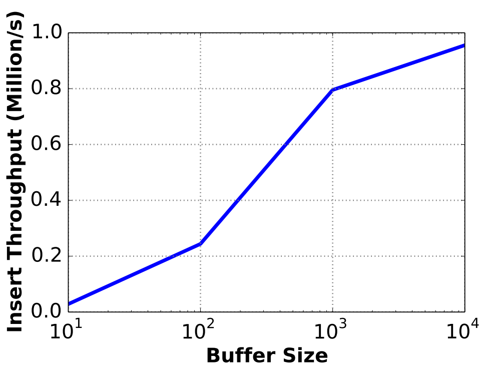


Figure 5.18: Insert Throughput / Varying Buffer Size

5.6.9 Exp 9: Varying Fill Factor

As previously mentioned, the buffer size of a segment determines the amount of space that a segment reserves to hold new data items. Once the segment's insert buffer reaches this threshold, the data from the segment and the segment's buffer is merged, and A-Tree executes the previously described segmentation algorithm to generate new segments that satisfy the specified error threshold.

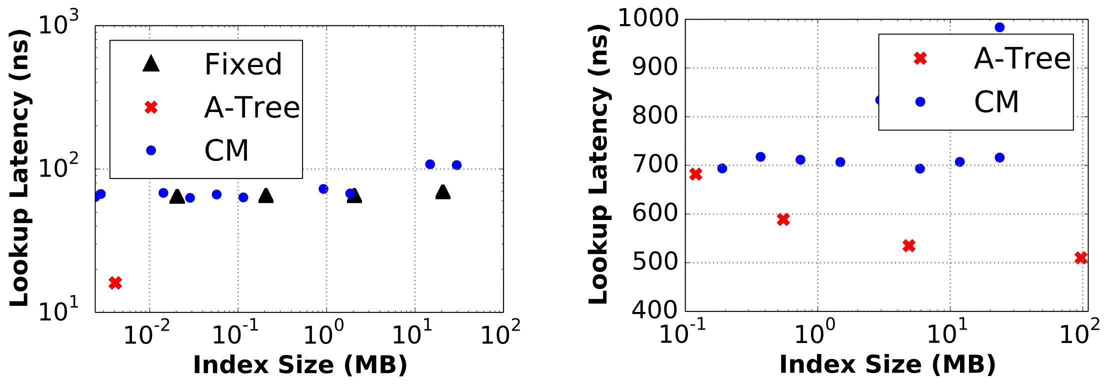
Therefore, in Figure 5.18, we vary the buffer size and measure the total throughput that A-Tree can achieve using the Weblogs dataset with an error threshold of $e = 20,000$. As shown, the size of the buffer can dramatically impact the write throughput of an A-Tree. More specifically, larger buffers result in fewer splitting operations, improving performance. However, a buffer that is too large will result in longer lookup latencies (modeled in the cost model in Section 5.5).

Therefore, the fill factor of an A-Tree can be effectively used by a DBA to tune an A-Tree to be more read or write optimized, depending on the workload.

5.6.10 Exp. 10: Correlation Maps

Correlation Maps [83] are designed to exploit correlations that exist between attributes. More specifically, they leverage an existing primary index to more efficiently access data items. To reduce the size of a secondary index created over continuous values, the authors also propose a bucketing strategy across both dimensions. In the following, we first show that CMs applied to primary indexing are no more efficiently to a B+ tree that uses fixed-sized pages using a simple dataset. Then, we show how CMs compare to A-Tree using the previously described Weblogs dataset.

To ensure a fair comparison and to adapt the techniques described in the Correlation Maps paper to be used as a clustered primary index, we assumed that each tuple has an implicit position (e.g., ROWID). In addition to the design in the original paper [83] (e.g., bucketing both along the clustered and unclustered dimension), we implemented additional optimizations not described in the paper since our use case has the additional knowledge that the unclustered attribute (i.e., timestamp) is sorted with respect to the clustered attribute (i.e., ROWID). For example, instead of storing several



(a) **Linear Distribution** (b) **Weblogs**

Figure 5.19: Correlation Maps

buckets for a given unclustered range (e.g., {100-200} → [b0,b1,b2,b3]) our implementation stores only the first and last bucket only (e.g., {100-200} → [b0,b3]). Additionally, when looking up a key, our implementation uses binary search within the entire region instead of searching each bucket individually. We found that these two optimizations improved lookup performance/reduced the size of a CM.

First, to show that CMs are no more efficient for primary indexes than an index that uses fixed-sized pages, we consider the simple case of indexing all integer values from 1 to 100M. Although simple, this is not unusual since users often index monotonically increasing identifiers (e.g., customer ID). Figure 5.19a shows the lookup latency for various index sizes (x-axis) for CMs, A-Tree, and fixed-sized paging. Again, as discussed before, we vary the size of A-Tree by selecting various error thresholds and use different page sizes/bucket sizes for CMs and B+ trees that use fixed-sized paging. As shown, CMs perform similar to B+ trees with fixed-sized pages, since they use fixed-sized buckets to partition the attribute domain. A-Tree, on the other hand, can use a single segment to represent this data and can locate to the exact position of any element using almost no space.

Next, we also used CMs also to build a primary key index on the Weblogs data set and compare it to our A-Tree. Figure 5.19b shows size (x-axis) vs. the lookup latency (y-axis) for both CMs and A-Tree using 400M timestamps from the Weblogs dataset. Since A-Tree creates variable-sized segments that better model the underlying data distribution (instead of the fixed-sized bucketing approach that CMs use), A-Tree is able to provide faster lookup performance using a smaller memory footprint.

5.6.11 Exp. 11: Index Construction

In the following, we quantify the cost to construct a A-Tree. More specifically, we measure the amount of time required to bulk load a A-Tree and compare it to the time required to construct a

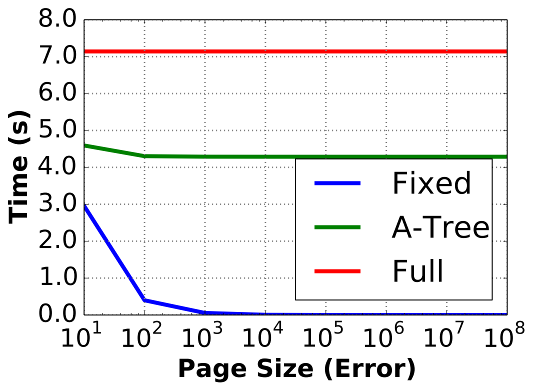


Figure 5.20: Index Construction Time

B+ tree that uses fixed-sized pages as well a full index. The results, shown in Figure 5.20 show the runtime for each approach, for various page sizes (Fixed B+ tree) or error thresholds (A-Tree).

Since a full index must insert every element into the tree, it takes a constant amount of time. However, the time required to bulk load a B+ tree that uses fixed-sized pages decreases as the page size increases since only the page boundaries (e.g., every 100th element) must be accessed from the underlying data. As previously mentioned, building a A-Tree requires first segmenting the data, and then inserting each segment into the underlying tree structure. Unsurprisingly, since the segmentation algorithm must examine every element in the data, a A-Tree incurs extra overhead ($\approx$ 4.2 seconds in the shown experiment) compared to a B+ tree that uses fixed-sized pages.

Interestingly, however, this constant amount extra overhead can be avoided in some cases. For example, when initially loading the data into the system, it is possible to execute the segmentation algorithm in a streaming fashion. In this case, it is possible that building a A-Tree is actually less expensive than building a B+ tree with fixed-sized pages since the resulting segmentation algorithm leverages trends in the data to produce fewer lead-level entries.

5.7 Related Work

The presented techniques in this paper overlap with work in different areas including (1) index compression, (2) partial/adaptive indexes, and (3) function approximation.

5.7.1 Index Compression:

Since B+ trees often consume significant space, several index compression techniques have been proposed. These approaches reduce the size of keys in internal nodes by applying techniques such as prefix/suffix truncation, dictionary compression, and key normalization [52, 50, 105]. Importantly, these techniques can be applied within A-Tree to further reduce the size of the underlying tree.

Similar to B+ tree compression, several methods have been proposed in order to more compactly represent bitmap indexes [23, 120, 133, 13, 144, 74]. Many of these techniques are specific to bitmap indexes, which are primarily only useful for attributes with few distinct values and not the general workloads that A-Tree targets. Moreover, other techniques have been developed to better align the node layout with the requirements of modern hardware by also utilizing compression. For example, CSB+ trees [123] remove the need to store pointers by using offsets but target mostly read-heavy workload. FAST [82] is another more recent tree structure that organizes tree elements in order to store then index in a more compact representation and exploit modern hardware features (e.g., SIMD, cache line size) for read-heavy workloads. Similarly, an Adaptive Radix Tree (ART) [86] also tried to leverage CPU caches for in-memory indexing. Another idea discussed in [154] are hybrid indexes which separate the index in hot and cold regions where cold data is stored in a compressed format since it is updated less often. Lastly, Log-structured Merge-trees [109] are designed for mostly write intensive workloads and extensions including Monkey [36] balance performance and memory consumption. Although there are many other compression schemes, many of these techniques can be seen as orthogonal and thus also could be used by A-Tree to more efficiently store the underlying tree structure as well as optimize for read-heavy workloads or hot/cold data.

Correlation Maps [83] try to leverage correlations between an unclustered attribute and a clustered attribute when an existing primary key index exists. Our approach, on the other hand, does not assume an existing index already exists and uses variable sized paging (instead of fixed-sized buckets) that better model the underlying data.

Bloom filters are used extensively in systems to check if an element belong to a set, resulting in a great deal of research [100, 27, 113]. However, unlike A-Tree which targets general index operations, bloom filters are only useful for a small class of queries.

Other indexing techniques have been proposed to store information about a region of the dataset, instead of the indexing individual keys. For example, leaf nodes in a BF-Tree [11] are bloom filters. Unlike A-Tree, BF-Tree does not exploit properties about the data's distribution when segmenting a dataset. Another example are learned indexes [84], which aim to learn the underlying data distribution to index data items. Unlike learned indexes, A-Tree has strict error guarantees, supports insert operations, and provides a cost model to ensure predictable performance and storage consumption. Sparse indexes like Hippo [149], Block Range Indexes [121], and Small Materialized Aggregates (SMAs) [101] all store information about value ranges similar to the idea of segments that we store in A-Tree. However, these techniques do not consider the underlying data distribution or bound the latency for lookups/inserts.

Finally, several approximation techniques have been proposed in order to improve the performance of similarity search [60, 124, 46] (for string or multimedia data), unlike A-Tree which uses approximation for compressing indexes optimized for traditional point and range queries.

5.7.2 Partial and Adaptive Indexes:

Partial indexes [134] aim to reduce the storage footprint of an index since they index only a subset of the data that is of interest to the user. For example, Tail Indexes [47, 34] store only rare data items in order to reduce the storage footprint of the overall index. A-Tree, on the other hand, supports queries over all attribute values but could be extended to index only "important" data ranges as well. Furthermore, database cracking [62] is a technique that physically reorders values in a column in order to more efficiently support selection queries without needing to store secondary indexes. Since database cracking reorganizes values based on past queries, it does not efficiently support ad-hoc queries, like A-Tree can.

5.7.3 Function Approximation:

The main idea of an A-Tree is to approximate the data distribution using piece-wise linear functions. The problem of approximating curves using piece-wise functions is not new [45, 21, 40, 91]. The error metrics E_2 (integral square error) and E_∞ (maximal error) for these approximations have been discussed as well as different segmentation algorithms [115, 44]. Unlike prior work, we consider only monotonic increasing functions and E_∞. Moreover, none of these techniques have been applied to indexing.

More recent work [129, 80, 26, 24, 146] for time series data also leverages piece-wise linear approximations to store patterns for similarity search. While these works also trade-off the number of segments with the accuracy of the approximated representation, they do not aim to provide the lookup and space consumption guarantees that A-Tree does, and do not have the analysis related to these guarantees.

Finally, other work [7] leverages piece-wise linear functions to compress inverted lists by storing functions and the distances of elements from the extrapolated functions. However, these approximations use linear regression (which minimizes E_2 per segment), and there are no bounds on the error (neither E_2 nor E_∞).

Chapter 6

Conclusion and Future Work

In this thesis, we presented novel techniques that all aim to improve the interactivity of visual data exploration tools for common data exploration tasks. First, we presented the first Interactive Data Exploration Accelerator (IDEA), a new middleware that sits between a visual data exploration tool and a backend data source (e.g., legacy DBMS, flat file, analytics framework). Then, to better support the conversational paradigm that visual data exploration tools promote and to enable exploration for larger datasets, in Chapter 3, we described novel approximate query processing techniques specifically for IDEAs. Next, since time is often a critical component of many interesting datasets, we presented an in-depth study of backend systems for use in our IDE stack specifically in the context of time series data. Based on the results from our study of existing time series data management solutions, we outline our ideas for a new type of index structure in Chapter 5.

Importantly, several of the core ideas of this worked motivated many new research directions and projects [25, 153, 18, 41, 152, 20, 55, 155, 156, 42]. Such projects span everything from ensuring safe visual data exploration and automatic machine learning to new visually balanced index structures and benchmarks for interactive data exploration.

Looking ahead, we believe that there are several exciting new research directions that remain to be explored. First, we believe that several interesting optimizations are possible in our proposed interactive data exploration stack. For example, an IDEA could leverage any of the internal components inside a backend data source (e.g., use indexes that exist in a DBMS to push down predicates to a DBMS). Additionally, we believe that there are huge opportunities to improve the performance of machine learning in the context of visual data exploration tools like Vizdom. For example, to support interactive machine learning over large datasets, machine learning algorithms which provide approximate results with a bounded error have the potential to offer huge benefits to end users. Lastly, we believe that incorporating trends that exist in the underlying data has the potential to offer significant performance improvements for a wide variety of core data management problems. For example, similar to how A-Tree leverages patterns that exist in the data to compress indexes, we believe that storage and compression can be dramatically improved using similar techniques.

Bibliography

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak R. Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into Data at Facebook. In *VLDB*, pages 1057–1067, 2013.

[2] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional Samples for Approximate Answering of Group-By Queries. In *SIGMOD*, pages 487–498, 2000.

[3] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The Aqua Approximate Query Answering System. In *SIGMOD*, pages 574–576, 1999.

[4] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, pages 29–42, 2013.

[5] Akumuli. `https://akumuli.org/`, 2018.

[6] Michael P Andersen and David E. Culler. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *FAST*, pages 39–52, 2016.

[7] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *VLDB*, pages 470–481, 2011.

[8] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB*, pages 480–491, 2004.

[9] Martin Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. IoTAbench: An Internet of Things Analytics Benchmark. In *ICPE*, pages 133–144, 2015.

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[11] Manos Athanassoulis and Anastasia Ailamaki. BF-tree: Approximate Tree Indexing. In *VLDB*, pages 1881–1892, 2014.

[12] Manos Athanassoulis and Stratos Idreos. Design Tradeoffs of Data Access Methods. In *SIGMOD*, pages 2195–2200, 2016.

[13] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*, pages 1319–1332, 2016.

[14] Atlas. `https://github.com/Netflix/atlas`, 2018.

[15] Andreas Bader, Oliver Kopp, and Michael Falkenthal. Survey and Comparison of Open Source Time Series Databases. In *BTW*, pages 249–268, 2017.

[16] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *SIGMOD*, pages 1363–1375, 2016.

[17] Rudolf Bayer and Karl Unterauer. Prefix B-trees. *ACM Trans. Database Syst.*, pages 11–26, 1977.

[18] Carsten Binnig, Benedetto Buratti, Yeounoh Chung, Cyrus Cousins, Tim Kraska, Zeyuan Shang, Eli Upfal, Robert Zeleznik, and Emanuel Zgraggen. Towards Interactive Curation & Automatic Tuning of ML Pipelines. In *DEEM@SIGMOD*, pages 1:1–1:4, 2018.

[19] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. In *VLDB*, pages 528–539, 2016.

[20] Carsten Binnig, Lorenzo De Stefani, Tim Kraska, Eli Upfal, Emanuel Zgraggen, and Zheguang Zhao. Toward Sustainable Insights, or Why Polygamy is Bad for You. In *CIDR*, 2017.

[21] Dietrich Braess. Chebyshev Approximation by Spline Functions with Free Knots. *Numerische Mathematik*, pages 357–366, 1971.

[22] United States Census Bureau. U.S. and World Population Clock. `https://www.census.gov/popclock/`, 2018.

[23] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap Index Design and Evaluation. In *SIGMOD*, pages 355–366, 1998.

[24] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. Efficient Metric Indexing for Similarity Search and Similarity Joins. In *KDE*, pages 556–571, 2017.

[25] Yeounoh Chung, Sacha Servan-Schreiber, Emanuel Zgraggen, and Tim Kraska. Towards Quantifying Uncertainty in Data Analysis & Exploration. *IEEE Data Eng. Bull.*, pages 15–27, 2018.

[26] Tak chung Fu. A Review on Time Series Data Mining. *Engineering Applications of Artificial Intelligence*, pages 164 – 181, 2011.

[27] Saar Cohen and Yossi Matias. Spectral Bloom Filters. In *SIGMOD*, pages 241–252, 2003.

[28] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *NSDI*, pages 313–328, 2010.

[29] Couchbase. `https://www.couchbase.com/`, 2018.

[30] CrateDB. Benchmarking Complex Query Performance in CrateDB and PostgreSQL. `https://crate.io/a/benchmarking-complex-query-performance-cratedb-postgresql/`, 2018.

[31] CrateDB. `https://crate.io`, 2018.

[32] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An Architecture for Compiling UDF-centric Workflows. *VLDB*, pages 1466–1477, 2015.

[33] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Vizdom: Interactive Analytics through Pen and Touch. In *VLDB*, pages 2024–2027, 2015.

[34] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. The Case for Interactive Data Exploration Accelerators (IDEAs). In *HILDA@SIGMOD*, 2016.

[35] Cube. `http://square.github.io/cube/`, 2018.

[36] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*, pages 79–94, 2017.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, pages 205–220, 2007.

[38] Druid. Benchmarking Druid. `http://druid.io/blog/2014/03/17/benchmarking-druid.html`, 2018.

[39] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*, pages 1275–1289, 2017.

[40] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A Time-series Compression Technique and Its Application to the Smart Grid. *The VLDB Journal*, pages 193–218, 2015.

[41] Philipp Eichmann, Carsten Binnig, Tim Kraska, and Emanuel Zgraggen. IDEBench: A Benchmark for Interactive Data Exploration. *CoRR*, abs/1804.02593, 2018.

[42] Muhammad El-Hindi, Zheguang Zhao, Carsten Binnig, and Tim Kraska. VisTrees: Fast Indexes for Interactive Data Exploration. In *HILDA@SIGMOD*, pages 5:1–5:6, 2016.

[43] Elasticsearch. `https://www.elastic.co/products/elasticsearch`, 2018.

[44] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *VLDB*, pages 145–156, 2009.

[45] R.E Esch and W.L Eastman. Computational Methods for Best Spline Function Approximation. *Journal of Approximation Theory*, pages 85 – 96, 1969.

[46] Andrea Esuli. Use of Permutation Prefixes for Efficient and Scalable Approximate Similarity Search. *Inf. Process. Manage.*, pages 889–902, 2012.

[47] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting Reuse for Approximate Query Processing. In *VLDB*, pages 1142–1153, 2017.

[48] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. A-Tree: A Bounded Approximate Index Structure. *CoRR*, abs/1801.10207, 2018.

[49] Google Cloud Machine Types. `https://cloud.google.com/compute/docs/machine-types`, 2018.

[50] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing Relations and Indexes. In *ICDE*, pages 370–379, 1998.

[51] Goetz Graefe. B-tree Indexes, Interpolation Search, and Skew. In *DaMon@SIGMOD*, 2006.

[52] Goetz Graefe and Per-Åke Larson. B-Tree Indexes and CPU Caches. In *ICDE*, pages 349–358, 2001.

[53] Graphite. Graphite. `https://graphiteapp.org/`, 2018.

[54] 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.*, pages 29–53, 1997.

[55] Yue Guo, Carsten Binnig, and Tim Kraska. What You See is Not What You Get!: Detecting Simpson's Paradoxes During Data Exploration. In *HILDA@SIGMOD*, pages 2:1–2:5, 2017.

[56] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD*, pages 287–298, 1999.

[57] Apache HBase. `https://hbase.apache.org/`, 2018.

[58] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, pages 51–59, 1999.

[59] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *SIGMOD*, pages 171–182, 1997.

[60] Michael E. Houle and Jun Sakuma. Fast Approximate Similarity Search in Extremely High-Dimensional Data Sets. In *ICDE*, pages 619–630, 2005.

[61] IDC. Rich Data and the Increasing Value of the Internet of Things. `https://www.emc.com/collateral/analyst-reports/idc-digital-universe-2014.pdf`, 2014.

[62] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, pages 68–78, 2007.

[63] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-memory Column-stores. *VLDB*, pages 586–597, 2011.

[64] InfluxData. InfluxDB Markedly Outperforms OpenTSDB in Time Series Data & Metrics Benchmark. `https://www.influxdata.com/blog/influxdb-markedly-outperforms-opentsdb-in-time-series-data-metrics-benchmark/`, 2018.

[65] InfluxData. InfluxDB Tops Cassandra in Time Series Data & Metrics Benchmark. `https://www.influxdata.com/blog/influxdb-vs-cassandra-time-series/`, 2018.

[66] InfluxData. InfluxDB vs. Elasticsearch for Time Series Data & Metrics Benchmark. `https://www.influxdata.com/blog/influxdb-markedly-elasticsearch-in-time-series-data-metrics-benchmark/`, 2018.

[67] InfluxDB. `https://www.influxdata.com/`, 2018.

[68] Yannis E. Ioannidis and Stratis Viglas. Conversational Querying. *Inf. Syst.*, pages 33–56, 2006.

[69] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An Architecture for Recycling Intermediates in a Column-Store. In *SIGMOD*, pages 309–320, 2009.

[70] Kaippallimalil J. Jacob and Dennis Shasha. A Financial Time Series Benchmark. `http://cs.nyu.edu/cs/faculty/shasha/fintime.html`, 2000.

[71] Chris Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. A Disk-Based Join With Probabilistic Guarantees. In *SIGMOD*, pages 563–574, 2005.

[72] Christopher M. Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable Approximate Query Processing with the DBO Engine. In *SIGMOD*, pages 725–736, 2007.

[73] Feng Jiang, Wen Tao, Shaohui Liu, Jie Ren, Xun Guo, and Debin Zhao. An End-to-End Compression Framework Based on Convolutional Neural Networks. *CoRR*, abs/1708.00838, 2017.

[74] Theodore Johnson. Performance Measurements of Compressed Bitmap Indices. In *VLDB*, pages 278–289, 1999.

[75] Kaggle. Young People Survey. `https://www.kaggle.com/miroslavsabo/young-people-survey`, 2018.

[76] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. Distributed and Interactive Cube Exploration. In *ICDE*, pages 472–483, 2014.

[77] Asterios Katsifodimos and Sebastian Schelter. Apache Flink: Stream Analytics at Scale. In *IC2E*, page 193, 2016.

[78] kdb+. `https://kx.com/discover`, 2018.

[79] Eamonn Keogh and Shruti Kasetty. On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. *Data Min. Knowl. Discov.*, pages 349–371, 2003.

[80] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An Online Algorithm for Segmenting Time Series. In *ICDM*, pages 289–296. IEEE, 2001.

[81] Albert Kim, Eric Blais, Aditya G. Parameswaran, Piotr Indyk, Samuel Madden, and Ronitt Rubinfeld. Rapid Sampling for Visualizations with Ordering Guarantees. In *VLDB*, pages 521–532, 2015.

[82] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[83] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. In *VLDB*, pages 1222–1233, 2009.

[84] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, pages 489–504, 2018.

[85] Domine M. W. Leenaerts and Wim M. Van Bokhoven. *Piecewise Linear Modeling and Analysis.* Kluwer Academic Publishers, 1998.

[86] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*, pages 38–49, 2013.

[87] Amaury Lendasse, Erkki Oja, Olli Simula, and Michel Verleysen. Time Series Prediction Competition: The CATS Benchmark. *Neurocomputing*, pages 2325–2329, 2007.

[88] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*, pages 615–629, 2016.

[89] M. Lichman. UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`, 2013.

[90] Lauro Didier Lins, James T. Klosowski, and Carlos Eduardo Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *TVCG*, pages 2456–2465, 2013.

[91] Xiaoyan Liu, Zhenjiang Lin, and Huaiqing Wang. Novel online methods for time series segmentation. *IEEE Trans. on Knowl. and Data Eng.*, pages 1616–1626, 2008.

[92] Xiufeng Liu, Lukasz Golab, Wojciech M. Golab, and Ihab F. Ilyas. Benchmarking Smart Meter Data Analytics. In *EDBT*, pages 385–396, 2015.

[93] Zhicheng Liu and Jeffrey Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *TVCG*, pages 2122–2131, 2014.

[94] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. *imMens*: Real-time Visual Querying of Big Data. *Comput. Graph. Forum*, pages 421–430, 2013.

[95] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In *UCC*, pages 69–78, 2014.

[96] Lucene. `https://lucene.apache.org/`, 2018.

[97] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. Data Ingestion for the Connected World. In *CIDR*, 2017.

[98] MgBench: A Benchmark for Machine-generated Data Management. `https://github.com/BrownBigData/MgBench`, 2018.

[99] Fabio Miranda, Marcos Lage, Harish Doraiswamy, Charlie Mydlarz, Justin Salamon, Yitzchak Lockerman, Juliana Freire, and Claudio T. Silva. Time Lattice: A Data Structure for the Interactive Visual Analysis of Large Time Series. *Computer Graphics Forum*, 2018.

[100] Michael Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Trans. Netw.*, pages 604–612, 2002.

[101] Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, pages 476–487, 1998.

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

[103] MongoDB. `https://www.mongodb.com/`, 2018.

[104] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. Recycling in Pipelined Query Evaluation. In *ICDE*, pages 338–349, 2013.

[105] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-style Engine for RDF. *VLDB*, pages 647–659, 2008.

[106] NYC Taxi & Limousine Commission Trip Record Data. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`.

[107] International Organisation of Vine and Wine. OIV Statistical Report on World Vitiviniculture. `http://www.oiv.int/public/medias/5336/infographie-focus-oiv-2017-new.pdf`, 2017.

[108] Frank Olken. *Random Sampling from Databases.* PhD thesis, University of California at Berkeley, 1993.

[109] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, pages 351–385, 1996.

[110] OpenTSDB. `http://opentsdb.net/`, 2018.

[111] OpenStreetMap database. `https://aws.amazon.com/public-datasets/osm`.

[112] Alvitta Ottley, Huahai Yang, and Remco Chang. Personality as a Predictor of User Strategy: How Locus of Control Affects Search Strategies on Tree Visualizations. In *CHI*, pages 3251–3254, 2015.

[113] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An Optimal Bloom Filter Replacement. In *SODA*, pages 823–829, 2005.

[114] Yongjoo Park, Ahmad Shahab Tajik, Michael J. Cafarella, and Barzan Mozafari. Database Learning: Toward a Database that Becomes Smarter Every Time. In *SIGMOD*, pages 587–602, 2017.

[115] T. Pavlidis and S. L. Horowitz. Segmentation of Plane Curves. *IEEE Trans. Comput.*, pages 860–870, 1974.

[116] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A Fast, Scalable, In-memory Time Series Database. In *VLDB*, pages 1816–1827, 2015.

[117] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic Indexing in Main-memory Column-stores. In *SIGMOD*, pages 1153–1166, 2015.

[118] Tom Petty and the Heartbreakers. The Waiting, 1981.

[119] PGTune. `https://pgtune.leopard.in.ua/`, 2018.

[120] Ali Pinar, Tao Tao, and Hakan Ferhatosmanoglu. Compressing Bitmap Indices by Data Reorganization. In *ICDE*, pages 310–321, 2005.

[121] PostgreSQL. `https://www.postgresql.org/`, 2018.

[122] QuasarDB. QuasarDB. `https://www.quasardb.net/`, 2018.

[123] Jun Rao and Kenneth A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.

[124] K. V. Ravi Kanth, Divyakant Agrawal, and Ambuj Singh. Dimensionality Reduction for Similarity Searching in Dynamic Databases. In *SIGMOD*, pages 166–176, 1998.

[125] Galen Reeves, Jie Liu, Suman Nath, and Feng Zhao. Managing Massive Time Series Streams with Multi-scale Compressed Trickles. *VLDB*, pages 97–108, 2009.

[126] RRDtool. RRDtool. `https://oss.oetiker.ch/rrdtool/`, 2018.

[127] Shibani Santurkar, David M. Budden, and Nir Shavit. Generative Compression. *CoRR*, abs/1703.01467, 2017.

[128] Arie Segev and Arie Shoshani. Logical Modeling of Temporal Data. In *SIGMOD*, pages 454–466, 1987.

[129] Hagit Shatkay and Stanley B Zdonik. Approximate Queries and Representations for Large Data Sequences. In *ICDE*, pages 536–545, 1996.

[130] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. *CoRR*, abs/1701.08530, 2017.

[131] Anshu Shukla and Yogesh Simmhan. Benchmarking Distributed Stream Processing Platforms for IoT Applications. *CoRR*, abs/1606.07621, 2016.

[132] Lefteris Sidirourgos and Martin Kersten. Column Imprints: A Secondary Index Structure. In *SIGMOD*, pages 893–904, 2013.

[133] Michał Stabno and Robert Wrembel. RLH: Bitmap Compression Technique Based on Run-length and Huffman Encoding. *Inf. Syst.*, pages 400–414, 2009.

[134] Michael Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, pages 4–11, 1989.

[135] Michael Stonebraker and Ugur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, pages 2–11, 2005.

[136] STX B+ Tree. `https://panthema.net/2007/stx-btree/`.

[137] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. Cache-on-Demand: Recycling with Certainty. In *ICDE*, pages 633–640, 2001.

[138] TimescaleDB. Eye or the Tiger: Benchmarking Cassandra vs. TimescaleDB for time-series data. `https://blog.timescale.com/time-series-data-cassandra-vs-timescaledb-postgresql-7c2cc50a89ce`, 2018.

[139] Timescale. `https://www.timescale.com/`, 2018.

[140] TimescaleDB. TimescaleDB vs. Postgres for time-series. `https://blog.timescale.com/timescaledb-vs-6a696248104e`, 2018.

[141] TPCx-IoT. `http://www.tpc.org/tpcx-iot/default.asp`, 2018.

[142] TPC-DS. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.8.0.pdf`, 2018.

[143] TPC-H. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf`, 2018.

[144] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing Bitmap Indices with Efficient Compression. *ACM Trans. Database Syst.*, pages 1–38, 2006.

[145] Yifan Wu, Joseph M. Hellerstein, and Eugene Wu. A DeVIL-ish Approach to Inconsistency in Interactive Visualizations. In *HILDA@SIGMOD*, 2016.

[146] Zhenghua Xu, Rui Zhang, Ramamohanarao Kotagiri, and Udaya Parampalli. An Adaptive Algorithm for Online Time Series Segmentation with Error Bound Guarantee. In *EDBT '12*, 2012.

[147] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-time Analytical Data Store. In *SIGMOD*, pages 157–168, 2014.

[148] YCSB-TS. `https://github.com/TSDBBench/YCSB-TS`, 2018.

[149] Jia Yu and Mohamed Sarwat. Two Birds, One Stone: A Fast, Yet Lightweight, Indexing Scheme for Modern Database Systems. In *VLDB*, pages 385–396, 2016.

[150] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[151] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data. In *SIGMOD*, pages 913–918, 2015.

[152] Emanuel Zgraggen, Alex Galakatos, Andrew Crotty, Jean-Daniel Fekete, and Tim Kraska. How Progressive Visualizations Affect Exploratory Analysis. *TVCG*, pages 1977–1987, 2017.

[153] Emanuel Zgraggen, Zheguang Zhao, Robert Zeleznik, and Tim Kraska. Investigating the Effect of the Multiple Comparisons Problem in Visual Analysis. In *CHI*, pages 479:1–479:12, 2018.

[154] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*, pages 1567–1581, 2016.

[155] Zheguang Zhao, Lorenzo De Stefani, Emanuel Zgraggen, Carsten Binnig, Eli Upfal, and Tim Kraska. Controlling False Discoveries During Interactive Data Exploration. In *SIGMOD*, pages 527–540, 2017.

[156] Zheguang Zhao, Emanuel Zgraggen, Lorenzo De Stefani, Carsten Binnig, Eli Upfal, and Tim Kraska. Safe visual data exploration. In *SIGMOD*, pages 1671–1674, 2017.

[157] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, pages 59–, 2006.