WRITE-INTENSIVE DATA MANAGEMENT IN LOG-STRUCTURED STORAGE

WANG SHENG

Bachelor of Engineering

Harbin Institute of Technology, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2016

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.
I have duly acknowledged all the sources of information which have been used in the thesis.
This thesis has also not been submitted for any degree in any university previously.

[illegible]

WANG Sheng 18 April 2016

ACKNOWLEDGMENT

The completion of this thesis has been the most significant academic challenge for me. Without the support and guidance of the following people, the study would not have been possible. It is now my great pleasure to thank them all.

First, I would like to express my most sincere gratitude to my supervisor Prof. Beng Chin Ooi. I am so deeply grateful for his continuous support through my entire PhD life, for his patience, wisdom and immense knowledge. Without his contributions of time and ideas, I would not be able to complete the tough PhD study. His guidance taught me not only how to solve a research problem, but more importantly how to keep a research brain. I also thank him for offering me the opportunities to visit research labs and collaborate with excellent researchers.

I would like to thank all my co-authors during my PhD study for their invaluable support to accomplish my work. A special thank to Prof. David Maier for his initial idea and continuous guidance to my research topic, and Prof. Divyakant Agrawal, Prof. Gang Chen and Dr. Hoang Tam Vo.

I would like to thank my thesis advisory committee, Prof. Stephane Bressan and Prof. Chee Yong Chan, for their constructive comments for this thesis.

I thank all labmates and friends during my PhD journey: Jinyang Gao, Feng Li, Qian Lin, Xuan Liu, Peng Lu, Meiyu Lu, Yanyan Shen, Wei Wang, Hao Zhang, Meihui Zhang and many others. I must acknowledge my girl friend, Yifang Yin, for her accompany on my hard times.

Finally, I would like to thank my parents for their love, support and encouragement throughout my entire life, without whom I would never caught so may opportunities to pursue my dreamed life.

CONTENTS

ABSTRACT

As the rapid development of information technologies, more and more data are generated everyday. Real-world workloads are becoming write-intensive and large-scale. On one hand, world-wide applications have large user bases acting simultaneously. On the other hand, large and cheap storage drives allow us to capture high-volume data, e.g., user activity logs and sensor readings. This write-heavy tend poses new challenges to data management solutions, where databases are required to provide high throughput for write operations while preserving read performance.

In this dissertation, we work towards designing solutions for managing write-intensive workloads with the adoption of log-structured techniques. More specifically, we first propose a distributed log-structured storage, providing high write-throughput for key-value operations. It removes bottleneck by unifying data and log repositories, and supports fast failure recovery. Second, we design a novel indexing method on top of the log-storage to support efficient range queries. This method works well for observational data, which is a common and important type of write-intensive source. It utilizes intrinsic clustering property in raw data source, and reduces index structure size by orders of magnitude. Lastly, we provide an extended solution for indexing multi-dimensional observational data. It overcomes the data sparseness in multi-dimensional spaces, and minimizes space "over-coverage" introduced by conventional spatial indexing methods. We evaluate our proposed approaches via extensive experiments using real and benchmark workloads, and observe that even our approaches are optimized for write throughput, they still preserve good read efficiency.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In this chapter, we first briefly present the write-intensive trends in our daily applications and services, and discuss consequent challenges faced by data storage systems. We overview our solution – a distributed log-structured storage system – that meets requirements from write-intensive applications. After that, we discuss the challenges for observational data which has much higher write-rate than other sources and is extremely hard to query at low cost. We introduce our novel index approach on top of log-structured storage, and extend it as a multi-dimensional access method. Finally, we summarize the contributions of this dissertation and outline the organization.

1.1 Data is Becoming Write-Heavy

Nowadays, data is of great importance to the whole human civilization. Activities in banks, companies and laboratories all involve processing and analyzing of data. Data management systems are therefore designed to efficiently store and query data. Relational database management systems (RDBMS) are most widely used in both industry and academia, and have been evolved over forty years. The idea of relational data model was first proposed in E.F.Codd's 1970 paper [31], and its advent significantly facilitated data modeling and application programming.

However, as the rapid development of information technologies, data in real-world applications are becoming write-intensive and large-scale. There are several facts inducing this situation. First, applications and services need to generate and collect high-frequency data, such as web activities and mobile signals. Many world-wide applications have huge user bases, in which millions of users act simultaneously, producing massive reading and updating messages. As reported by Facebook in August 2012 [1], with more than 950 million users: 300 million photos were uploaded per day; 105 terabytes of data were scanned in Facebook's clusters every 30 minutes; and more than 500 terabytes of new data were ingested into their databases every day. Such high-rate data production is unimaginable before but can be observed everywhere now. As for 2014, in every minute we could see [2]:

- Facebook users share nearly 2.5 million pieces of content.
- Twitter users tweet nearly 300,000 times.
- Instagram users post nearly 220,000 new photos.
- YouTube users upload 72 hours of new video content.
- Email users send over 200 million messages.
- Amazon generates over $80,000 in online sales.

As shown in Figure 1.1, this data production rate will keep accelerating, and is estimated to lead to 50-fold growth of the entire digital universe from 2010 to 2020 [45].

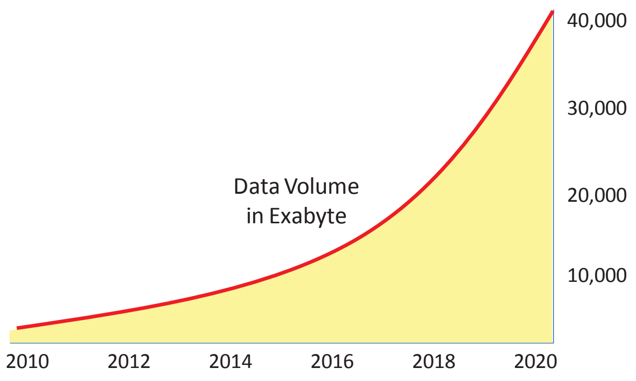


Figure 1.1: Size of total data in entire digital universe studied by IDC [45].

Second, data generation and access patterns have been shifting towards write-heavy. As been observed at Yahoo [78], the ratio of write operations in

[1]http://gigaom.com/2012/08/22/facebook-is-collecting-your-data-500-terabytes-a-day/
[2]http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/

their applications was just 10% $\sim$ 20% in 2010, but this figure had rapidly reached 50% in 2012. This shift is driven by different types of applications. For example, social applications, .e.g, Facebook and Twitter, accept many updates from different users and each user later operate a single read operation to retrieve all new posts. Besides, analytic applications ingest event logs (such as user clicks and mobile device sensor readings), and later mine the data by issuing long scans, or targeted range queries.

Third, larger and cheaper storage drives allow service providers to capture high-volume data, such as detailed user activity logs and high-resolution sensor device readings, without additional financial concern. For example, in 2015, Seagate Archive HDD with 8 terabytes capacity sold at lower than \$300 [3], which was just four cents per gigabyte. As shown in Figure 1.2, the composite cost for storing a gigabyte of data dropped from \$4 in 2010 to lower than \$1 in 2015.

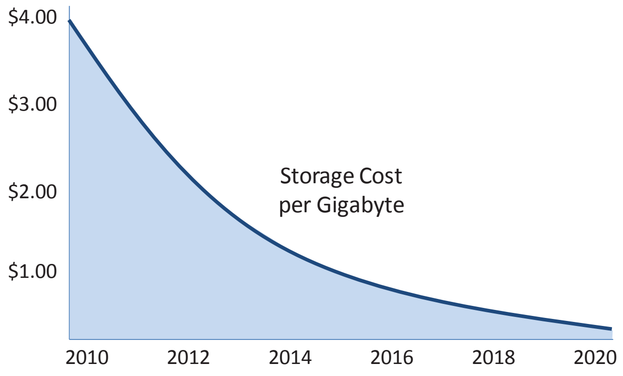


Figure 1.2: Falling costs for storing data studied by IDC [45].

These trends pose new challenges to modern data management systems. When facing large write-heavy data, the most important aspect is the capability to persist all inputs. Without strong ingestion power, we have to drop part of data or block user requests, which will interrupt the service of applications. This makes traditional RDBMs not an elegant solution, as large-volume data sources are not initially targeted. With complex built-in query processing engines, they are cumbersome to handle high-rate data ingestion, incurring prohibitive cost. This situation encourages the advent of NoSQL data stores, whose design principle is to easily scale to thousands of servers. They trade

[3]http://www.seagate.com/sg/en/products/enterprise-servers-storage/nearline-storage/archive-hdd/

complex query functionality and data consistency for better scalability and availability. Most of them are designed and used by commercial companies, such as Google BigTable [27], Facebook Cassandra [61], Amazon Dynamo [39] and MongoDB [8]. Though NoSQL data stores have been proved to be superior for high-volume data, the write-intensive trend has not been fully addressed by them. Typically, NoSQL data stores handle high-frequent write operations by distributing workloads among a sufficient number of servers. However, more servers handling a single workload introduces more coordination and communication overhead. Besides, it also means more financial cost and energy consumption. To fully address the write-intensive issue, we might need to think from a different angle: further improving the write-capacity of each individual server. It should be much more valuable if the same workload can be handled by a smaller cluster – hence lower budget and fewer energy.

In this dissertation, we address the problem of designing a storage system with high per-node write-throughput. Though our main focus is write-throughput, there are many other issues to consider, i.e., dynamic scalability, efficient data access and fast failure recovery, before we could have a practical system.

1.2 Write-Heavy Log-Structured Storage

Write-ahead-logging (WAL) [66] is widely adopted in both RDMBS and NoSQL systems. In this approach, in order to improve system performance while ensuring data durability, updates are first sequentially recorded into an external "stable storage" as a log, while application data are buffered in memory, which will be written back to their own disk-locations in the future. This method facilitates that similar data can be clustered physically on disk, optimizing for subsequent read requests. Although it can defer writing of disk data, sooner or later all data have to be persisted eventually, which would result in unexpected performance. As been discussed in Michael Stonebraker's blog [82], logging is one of the biggest bottlenecks with respect to storage performance: everything is written twice - once to the database and once to the log. The separation of log and application data nearly doubles I/O operations, which adversely affects the system severely in write-heavy applications, limiting the write throughput and recovery efficiency.

To remove this bottleneck, log-structured approaches can be adopted. New arrival application data are directly organized in an append-only manner, like writing into a log. Later on, out-of-date logs are cleaned and compacted to save space. This type of approaches has been widely used in many related areas, such as file systems [79] and memory management systems [77]. From database systems' point of view, log-structured approaches have two direct advantages. First, data redundancy can be completely eliminated. Application data and logging data can be managed in a unified repository, where a single copy of data serves both data access and failure recovery. Second, amortized I/O cost of write operations can be further reduced. Since both initial persisting and subsequent compaction all access disk sequentially, the number of expensive disk seek operations is minimized, compared to other approaches where each page write needs a disk seek.

Following this idea, we first propose a log-structured data store, named `LogBase`, which adopts log-only storage for removing the write bottleneck and supporting fast system recovery. It can be dynamically deployed on commodity clusters to take advantage of elastic scaling property of cloud environments.

1.3 Write-Heavy Observational Data

Among the growing write-intensive trends in all types of data source, there exists a major category of data source that is extremely write-heavy in nature. That is *observational data*, which is usually generated by sensing devices, recording the status of objects and the environments. Such data collection is now ubiquitous in many fields of scientific research and analytics.

There are mainly three facts contribute to the write-heavy nature of observational data. First, sensor devices appear for measuring a broader range of entities. Humankind has a rapidly growing ability to digitize the universe. The entities whose state can be continuously captured span from microscopic to macroscopic: molecules, cells, electronic devices, wild life, automobiles, dams, oceans and even distant stars. Second, sensors' capabilities keep improving in both sampling frequency and resolution. For example, a single sensor can capture the velocity of a moving object in units of μ/s at a frequency of $2000Hz$. Hence individual sensors can generate huge amount of data during a short period. Third, decreasing device prices and increasing power efficiency facilitate the deployment of large sensor networks. They may consist of thousands of sensors, producing simultaneous high-frequency observations. All these trends demand data storage with high write-throughput, to capture observations in a timely manner. Hence, log-structured storage, e.g., `LogBase`, should be amenable to handle such write-heavy ingestion.

Though log storage satisfies high ingestion rate, value-based data access is still a problem. Unlike most other write-heavy applications, where only key-based data lookup is involved, analysis on observational data usually requires filtering on attribute values. In order to support such complex queries, corresponding attributes should be externally indexed by secondary indexes. However, maintaining multiple index structures, e.g., B$^+$-trees, is too costly for write-heavy data sources, as index construction may take a few hours before they are ready to serve queries. Besides, too many system resources consumed by index construction inversely degrades the throughput for accepting new data. Consequently, low-cost construction is a critical requirement for indexes managing observational data.

To reduce index maintenance cost, various work has been proposed in the literature. Examples include bulk-insertion techniques [28, 30, 12] that update the indexes in a batch manner, which lowers the per-item cost; and log-structured-merge trees [71, 78, 14] that incur only sequential I/Os for updates; besides, database cracking [55, 48, 56] gradually constructs indexes during user queries, which reduces the downtime before the indexed data can be queried. Though all these methods addresses index-construction efficiency, they are still maintaining a large number of index entries and re-organizing them frequently, which still can not scale up well when observational data rapidly grow.

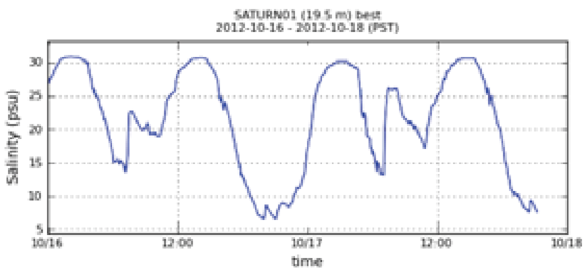


Figure 1.3: Value-continuity in estuary salinity status, collected by SATURN-01 [1].

We study the characteristics of observational data, and its natural compatibility with log-structured storage. We observe several facts from this data source, and the most interesting one is its *value-continuity*, i.e., observations from the same source tend to have similar readings during a short period, as can be seen in Figure 1.3. This inspires us a new idea called *intrinsic clustering*: instead of manually re-organizing all indexed data at record-level, we can somehow rely on local clustering patterns in original data source to provide reasonable access efficiency. With low-cost sequential scan, we can expect to retrieve multiple query results from a few consecutive disk pages.

Based on this idea, we propose a scheme for storing observational data in log-store while preserving data locality. We then design a novel indexing method, called `CR-index`, which is a lightweight structure that is fast to construct and often small enough to reside in RAM. In particular, it assigns one index entry for a collection of records and when a query arrives, it relies on partial scans to access potential results. As a consequence, it provides fast queries without compromising write throughput.

1.4 Observational Data in Multi-Dimension

In practice, an observational data source usually generates observations with multiple attributes (i.e., dimensions). For example, observations from underwater sensors may contain attributes like water temperature, salinity, oxygen saturation and pH. Sometimes, an analysis task may issue queries with several filter conditions on different attributes. Instead of accessing multiple `CR-index`es on corresponding attributes and joining partial results in an additional phase, it is much more effective to index all these attributes in a single structure and to answer multi-dimensional range queries in one step. This demand drives us to extend `CR-index` as a multi-dimensional structure.

However, extending one-dimensional structure to multi-dimension is not trivial. For example, R-tree is an anatomic extension to B$^+$-tree, but they have different re-balance strategies and operation complexity. Similarly, it is also challenging to exploit intrinsic clustering property, as used in `CR-index`, in multi-dimensional cases. First, data sparseness in multi-dimensional spaces hurts query efficiency. Bounding objects such as minimal bounding rectangles/boxes/spheres (MBR/MBB/MBS) are widely used in spatial access methods [50, 59] to represent a collection of data items. However, such representations will introduce "over-coverage", i.e., portions of indexed spaces that contain no actual points as shown in Figure 1.4, which causes accessing false-positive entries. This issue becomes even more severe as the dimensionality increases. Second, the write-intensive aspect limits resources and opportunity to derive bounding objects with the least over-coverage. The derivation of index entries should be extremely fast so that it does not affect the throughput for updates.

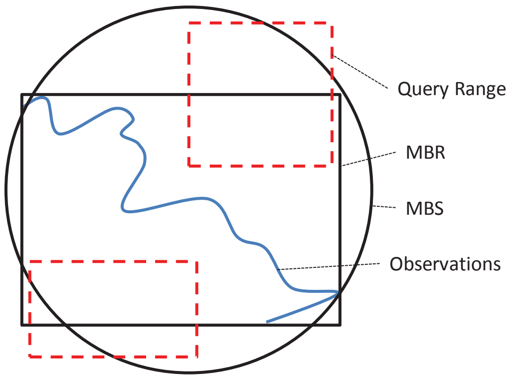


Figure 1.4: Over-coverage caused by bounding objects.

We address above challenges and propose a multi-dimensional indexing approach, called `SICC`, which is a general extension of `CR-index` for multi-dimension. Instead of representing collections of points using Minimal Bounding Boxes, we model sets of successive points as hyper-segments, which drastically reduces over-coverage. To ensure real-time access, we introduce algorithms for generating segments and data structures for building the index. Its fast construction guarantees extremely high throughput for data ingestion. We also design an adaptive refinement mechanism that can improve query performance over time in the background, based on query execution history.

1.5 Objectives and Contributions

In this dissertation, we work towards designing solutions to provide superior storage throughput for write-intensive applications, while preserving excellent data access performance especially for observational data. In particular, we summarize main contributions as follows:

- We design a distributed log-structured database system, called `LogBase`, that can be dynamically deployed in a cluster. It provides fast recovery capability while offering highly sustained throughput for write-heavy applications.

- We propose a light-weight indexing method, called `CR-index`, on top of `LogBase`, which indexes observational data without affecting system throughput. It is fast to construct, small to reside in RAM, and performs quite well for range queries.
- We extend the idea of `CR-index` and propose a multi-dimensional indexing framework, called `SICC`, which preserves low construction cost. It optimizes its data representation towards observational data, and conquers the data sparseness in multi-dimension.

Though our main focus for write-intensive applications is write throughput, we evaluate that at the same time our solutions also provide similar data access efficiency compared to read-optimized approaches.

1.6 Overview and Roadmap

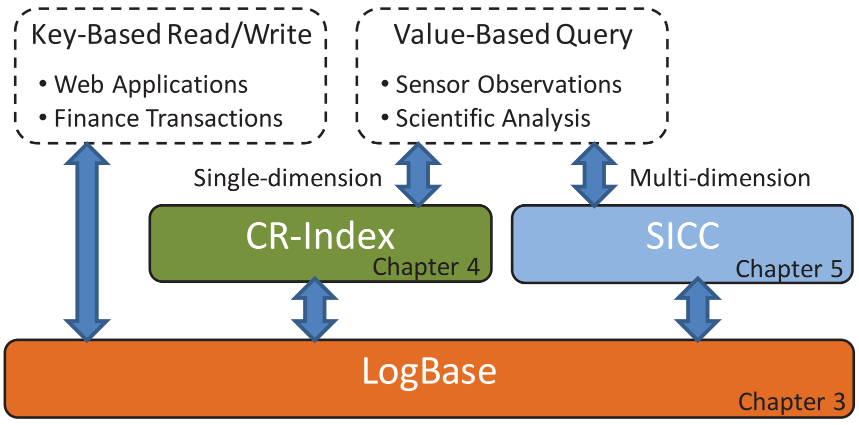


Figure 1.5: System overview.

The `LogBase` system targets applications under write-intensive environments, supporting various data access patterns, such as key-based access (e.g., web applications) and value-based access (e.g., analytic tasks). Figure 1.5 shows the system overview from applications' perspective, displaying how requirements are satisfied in different components.

The rest of the dissertation proceeds as follows:

Chapter 2 discusses the literature review.

Chapter 3 presents the design of `LogBase`, which supports high throughput for write-intensive applications.

Chapter 4 presents the design of `CR-index`, which improves query efficiency for observational data while not affecting `LogBase`'s throughput.

Chapter 5 presents the design of `SICC`, which is a general multi-dimensional extension for `CR-index`.

Chapter 6 concludes the dissertation and discusses possible future work.

CHAPTER 2

LITERATURE REVIEW

There have been a lot of research work in data management domain that improve write performance. In this chapter, we review those work that are closely related to this dissertation. From data storage perspective, we discuss current solutions to handle data updates in conventional storage; from data indexing perspective, we introduce existing techniques for reducing index maintenance overhead, for both single-key indexing and multi-dimentional indexing. We also analyze the limitation of prior methods when dealing with write-heavy applications. We also presents related work from observational data management perspective.

2.1 Data Storage Systems

In traditional data storage systems, an update operation is mainly processed by fetching the page containing original data and then propagating the update back to this page, which is called *in-place update*. To ensure durability, even a single byte has been changed on that page, should the page be written back to external disk. Besides, seeking to the specific page position for each update is quite costly. Hence, a number of non-overwrite strategies were proposed.

2.1.1 Non-Overwrite Strategies

Early database systems such as System R [49] use shadow paging strategy to avoid the cost of in-place updates. In this approach, the database is treated as a collection of pages and accessed through a page table, which maps page ids into disk addresses. When a transaction updates a data page, it makes a copy, i.e., a shadow, of that page and operates on that. When the transaction commits, it updates the page table by mapping the page ids into the new addresses. Although this approach does not require logging for data durability, the overheads of page copying and updating are much higher for each transaction, and adversely affect the overall system performance.

Another no-overwrite strategy for updating records is employed in POSTGRES [83, 84]. In particular, each data page in POSTGRES maintains an anchor table which is used to retrieve records stored in that page. When a record is inserted for the first time, space is allocated for the record. For data modification, instead of performing updating in-place in the page, a delta record is added to store the changes from the previous version of the record. When reading a record the system has to traverse and process the whole chain from the first version of the record. POSTGRES is initially optimized for small records, and delta records should be on the same page with the initial record. Although POSTGRES introduces many novel ideas, the performance is lower than expected due to the way records are stored, which requires read operations to reconstruct records from the delta chains. Further, POSTGRES uses a force buffer policy, i.e., all pages modified by a transaction are written into disk at commit. Such high cost of write operations is inadequate for write-heavy applications.

2.1.2 WAL + Data

ARIES [66] is an algorithm designed for database recovery and enables no-force, steal buffer management, and thus improves system performance, as updates to data can be buffered in memory without incurring "update loss" issues. The main principle of ARIES is write-ahead-logging (WAL), i.e., any change to a record is first stored in the log which must be written to "stable storage" before being reflected into the data structure.

WAL is a common approach in most storage systems ranging from traditional DBMSes, including open source databases like MySQL and commercial databases like IBM DB2, to the emerging cloud storage systems, a.k.a distributed key-value stores such as BigTable [27], HBase [4] and Cassandra [61]. One of the main reasons why this approach is popular is that while the log cannot be re-ordered, the data can be sorted or clustered in any order to exploit data locality for better I/O performance (e.g., clustering access via clustered indexes). However, this feature is not necessary for all applications. In addition, the separation of log and application data in this approach might incur potential overheads that would reduce the write throughput and increase the time for system recovery.

In particular, although this design defers writing the application data to disks in order to guarantee system response time, sooner or later all the data buffered in memory have to be reflected into the physical storage, which could result in write bottlenecks. Therefore, the system might not be able to provide high write throughput for handling a large amount of incoming data in write-heavy applications. In addition, when recovering from machine failures the system needs to replay relevant log records and update the corresponding application data before it is ready for serving new user requests. As a consequence, the time to bring the failed machine back to usable state is delayed.

2.1.3 Log-Structured Systems

Log-structured file systems (LFS) pioneered by Ousterhout and Rosenblum [76] for write-heavy environments have been well studied in the OS community. More recently, BlueSky [79], a network file system that adopts log-structured design and stores data persistently in a cloud storage provider, has been proposed. However, to import similar idea into data storage systems, there are many challenges. That is, we need to provide database abstraction in stead of a file-system abstraction on top of the segmented log; we need to support fine-grained access to data records instead of data blocks as in LFS; and we need to take care of the range query performance after frequent updates.

Contemporary log-structured systems for database applications include Berkeley DB [70] and PrimeBase [10] (an open source log-structured storage engine for MySQL). Both systems are developed for single machine environment and use disk-resident indexes, which restricts system scalability and performance.

Storing data in a sequential log is used as a way to maintain historical versions of data in temporal databases [64]. The log-only principle is also employed in Vagabond [69], a temporal object database system. These log-structured temporal database systems are also designed for centralized environments.

Recent research systems for scalable log-structured data management include Hyder [22] and RAMCloud [72]. Hyder takes advantage of new advent of modern hardware such as solid-state drives and high-speed network for shared-flash environments. Similarly, RAMCloud is a DRAM-based storage system that requires servers with large memory and high-speed network to meet latency goals. It implements a unified log-structured mechanism [77] both for active data in memory and backup data on disk. OctopusDB [40] is another research proposal that uses the log as its primary data repository. Embracing the idea of "one size fits all". OctopusDB creates various storage views over the log to serve different applications. Developing an "one size fits all" system, though interesting, might reduce the needed performance for all applications. Unfortunately, none of existing designs considers a shared-nothing disk-based system that is inherently suitable for large-scale commodity clusters.

2.2 Data Indexing Methods

Index structures play an important role in supporting searches. In order to provide good performance for complex queries, multiple secondary indexes are required. To guarantee consistency, those indexes are updated whenever the data changes. However, conventional index methods such as B^+-tree [32] entail expensive maintenance overhead, due to frequent and expensive node splitting. Such overhead is a critical performance bottleneck, especially for write-intensive applications. Therefore, a number of methods are proposed to reduce index maintenance cost.

2.2.1 Bulk Insertion

One simple yet effective solution is bulk-insertion techniques [47, 12]. These methods buffer and group new insertions, and eventually update indexes in batch manner. Hence, a bunch of writes on the same page can be done via a single I/O which reduces per-item cost. To ensure correctness, query operations need to search the buffer structure in addition to original indexes. As this idea is generally applicable for a wide range of indexes, those structures with high update cost might benefit much from bulk insertions. For example, generalized bulk-insertion strategies have been proposed for R-trees [28, 30]. However, such approaches only alleviate the influence of in-place update, rather than replace in-place to completely eliminate potential bottleneck.

2.2.2 Adaptive Indexing

Constructing a complete index structure for huge incoming data introduces hours or days of downtime before those data can be queried. Recently, adaptive indexing methods, such as Adaptive Merge Trees [48] and Cracker Indexes [55, 52], have been proposed to gradually construct indexes during user queries. At first, there are no indexes on top of the raw data, and queries are executed by scanning the whole data. During those executions, adaptive indexes are automatically constructed by refining data physical representation in response to incoming queries. Hence, answering a query can also improve subsequent queries at the same time. Given enough queries that touch all data items, data representation of an adaptive index can eventually converge to a pre-built index, such as a B^+-tree. The subsequent updates will break the clustering property built by existing indexes. Hence, the indexes are dropped and rebuilt periodically. Adaptive indexes reduce the downtime before raw data can be queried. However, they do not effectively reduce total index construction cost, (and may incur more overhead sometimes). Therefore, they are mainly used in read-heavy data warehouses.

2.2.3 Log-Structured Merge Tree

Following no-overwrite strategies introduced by early database systems, log structured merge tree (LSM-tree [71]), which is a hierarchy of indexes spanning across memory and disk, is proposed for maintaining real-time indexes at low I/O cost. All incoming updates are maintained in an in-memory structure. When the in-memory buffer is full, all updates are sorted and dumped into the disk. Instead of directly apply updates to existing on-disk trees, these updates are stored as an independent tree sub-component. Hence, random writes are replaced by sequential I/Os. However, to answer a query, all those on-disk sub-components are forced to be accessed. Periodically, exponential-sized sub-components are merged to bound the read latency. The log-structured history data access method (LHAM [67]) is an adaptation of LSM-tree for hierarchical storage systems that store a large number of components on archival media. The bLSM-tree [78], an optimization of LSM-tree that guarantees excellent read performance against read-optimized B$^+$-trees in common workloads, has been proposed. It uses Bloom filters [23] to prune unnecessary sub-component lookups. Besides, it applies spring and gear scheduler and snow-shoveling merging to bound write latency without impacting throughput. Recently, LSM-trie [89] adopts LSM strategy to trie-trees to further reduce write amplification, with the expense of range query capability.

It is also noteworthy that LSM-trees are designed to be compatible with external write ahead logs. Therefore, although some cloud storage systems, such as HBase [4] and Cassandra [61], have adopted LSM-trees for maintaining their data, they have not totally removed potential write bottlenecks since the separation of log and application data still exists in these systems.

2.2.4 Multi-Dimensional Access Methods

In the area of multi-dimensional indexes, a large number of multi-dimensional access methods have been proposed over the past three decades. These methods can be broadly classified into two classes, point access methods (PAMs) and spatial access methods (SAMs), and are examined in several surveys [44, 13, 65]. In PAMs, the whole space is divided into subspaces, either at a single level, e.g., grid files [68], or hierarchically, e.g., kD-trees [18] and quadtrees [42]. Recently, a space-efficient PH-tree [90] was proposed based on binary PATRICIA-tries and hypercubes. In contrast, SAMs are designed for objects with geometric extent. The most well known SAM index is the R-tree [17, 50]. Variants of the R-tree include the PR-tree [15], which is asymptotically optimal, and the X-tree [19] which avoids splits that may cause severe overlaps. In SAMs, bounding objects, e.g., minimum bounding rectangles/boxes/spheres, are used to approximate the extent of objects and bound the space of subtrees. Such representations may incur over-coverage, which affects query efficiency. An earlier work [58] uses polygons to reduce over-coverage. However, its costly construction is not suitable for the write-intensive scenarios.

2.3 Observational Data Management

Powerful sensing devices make observational data write-intensive, demanding high write-throughput storage and efficient query processing. In this section, we review the existing solutions for managing observational data in large scale. We also discuss other research fields that entail similar data characters but with different focus, such as trajectory indexing and time-series analysis.

2.3.1 Streaming Warehouse

Data and streaming warehouses are a major group of storage systems, some of which collect observational data. DataDepot [46] is a tool for building and managing streaming warehouses in an RDBMS, providing fast data loading, automated view maintenance and data consistency control. SDAF [29], a data warehouse framework for sensor data, supports spatial queries over objects relating to location and time. A cloud-based sensor data warehouse method [60] was proposed on top of the distributed NoSQL database HBase [4]. It provides a simple key-value data model to manage sensor data in the column-oriented paradigm.

NoSQL systems [4, 27, 61] can be used as large-scale observational data storage. One advantage of NoSQL systems is their high write throughput. In contrast to RDBMS, the data are simply represented as a set of key-value pairs. Since the data models and schema are more flexible and impose fewer constraints, the writing cost is substantially reduced. However, a drawback of such systems is the simple key-based interface, which does not support range retrieval on values.

2.3.2 Trajectory Indexing

Trajectory indexing over the sequences of positions of moving objects is an area related to observational data management. A trajectory can be viewed as continuous observations in exact two or three dimensions. Trajectory indexes usually store positions from multiple trajectories in a single structure while still preserving trajectory-level properties [75]. They are typically tailored for data in 2 or 3-D spaces with special assumptions [74], e.g., constrained movement and in-network movement. This case is different from observational data where we have a single long-life entity with many dimensions or attributes. In general, trajectory indexing focuses on the time dimension, i.e., find all trajectories at a time-point, while observations focuses on space dimensions, i.e., find all subsequences within a value-range.

2.3.3 Time-Series Analysis

Data-series and time-series analyses [86, 91] index multiple points, i.e., points in a series, in one index entry similar to the methods proposed in this dissertation, but they are fundamentally different. In time-series analysis, each series is considered as a whole, thus indexing points together is to facilitate similarity or k-NN search. However, for observational data, indexing points together is just to reduce maintenance cost, and individual observations are still queried independently.

2.4 Summary

In this chapter, we have reviewed related work for write-intensive data management. Log-structured methods are the main direction of effort in literature, which replace costly in-place updates with sequential writes to minimize disk-seek operations. Most commercial large-scale storage systems [4, 27, 61] adopt log-structured methods to satisfy the demand of huge-volume data ingestion. However, the separation of data and WAL is still the common setting, which limits the overall system throughput under write-intensive scenarios. To reduce expensive index maintenance overhead, log-structured merge trees [71, 78, 89] are the most feasible solutions for write-intensice applications. However, all existing methods are processing data at the record-level, which is hard to optimize when total records are in large scale. Another interesting direction is to exploit partial order and local clustering in unclustered data [81].

CHAPTER 3

A SCALABLE LOG-STRUCTURED DATA STORE

Numerous applications such as financial transactions (e.g., stock trading) are write-heavy in nature. The shift from reads to writes in web applications has also been accelerating in recent years. Write-ahead-logging is a common approach for providing recovery capability while improving performance in most storage systems. However, the separation of log and application data incurs write overheads observed in write-heavy environments and hence adversely affects the write throughput and recovery time in the system.

In this chapter, we introduce `LogBase` – a scalable log-structured database system that adopts log-only storage for removing the write bottleneck and supporting fast system recovery. It is designed to be dynamically deployed on commodity clusters to take advantage of elastic scaling property of cloud environments. `LogBase` provides in-memory multiversion indexes for supporting efficient access to data maintained in the log. `LogBase` also supports transactions that bundle read and write operations spanning across multiple records. We implemented the proposed system and compared it with HBase and a disk-based log-structured record-oriented system modeled after RAMCloud. The experimental results show that `LogBase` is able to provide sustained write throughput, efficient data access out of the cache, and effective system recovery.

3.1 Introduction

There are several applications that motivate the design and implementation of `LogBase`, such as logging user activity (e.g., visit click or ad click from high volume web sites) and financial transactions (e.g., stock trading). The desiderata for the backend storage systems used in such write-heavy applications include:

- **High write throughput.** In these applications, a large number of events occur in a short period of time and need to be durably stored into the backend storage quickliest possible so that the system can handle a high rate of incoming data.

- **Dynamic scalability.** It is desirable that the storage systems are able to support dynamic scalability for the increasing workload, i.e., the ability to scale out and scale back on demand based on load characteristics.

- **Efficient multiversion data access.** The support of multiversion data access is useful since in these applications users often perform analytical queries on the historical data, e.g., finding the trend of stock trading or users' behaviors.

- **Transactional semantics.** In order to relieve application developers from the burden of handling inconsistent data, it is necessary for the storage system to support transactional semantics for bundled read and write operations that possibly access multiple data items within the transaction boundary.

- **Fast recovery from machine failures.** In large-scale systems, machine failures are not uncommon, and therefore it is important that the system is able to recover data and bring the machines back to usable state with minimal delays.

Storage systems for photos, blogs, and social networking communications in Web 2.0 applications also represent well-suited domains for `LogBase`. The shift from reads to writes has been accelerating in recent years as observed at Yahoo! [78]. Further, since such data are often written once, read often, and rarely modified, it is desirable that the storage system is optimized for high aggregate write throughput, low read response time, faut-tolerance and cost-effectiveness, i.e., less expensive than previous designs in storage usage while offering similar data recovery capability.

Previous designs for supporting data durability and improving system performance, do not totally fit the aforementioned requirements. Copy-on-write strategy used in System R [49] incurs much overhead of copying and updating data pages, and therefore affects the write throughput. In POSTGRES [83, 84], a delta record is added for each update, which would increase read latency since records have to be reconstructed from the delta chains. In write-ahead-logging (WAL) [66], in order to improve system performance while ensuring data durability, updates are first recorded into the log presumably stored in "stable storage", before being buffered into the memory, which can be flushed into data structures on disks at later time. We refer to this strategy as WAL+Data approach. Although this approach can defer writing data to disks, all the data have to be persisted into the physical storage eventually, which would result in the write bottleneck observed in write-heavy applications. In addition, the need to replay log records and update corresponding data structures when recovering from machine failures before the system becomes ready for serving new requests is another source of delay.

`LogBase` instead adopts log-only approach, in which the log serves as the unique data repository in the system, in order to remove the write bottleneck. The essence of the idea is that all write operations are appended at the end of the log file without the need of being reflected, i.e., updated in-place, into any data file. There are some immediate advantages from this simple design choice. First, the number of disk I/Os will be reduced since the data only need to be written once into the log file, instead of being written into both log and data files like the WAL+Data approach. Second, all data will be written to disk, i.e., the log file, with sequential I/Os, which is much less expensive than random I/Os when performing in-place updates in data files. As a consequence, the cost of write operations with log-only approach is reduced considerably, and therefore `LogBase` can provide the much needed high write throughput for write-heavy applications. Log-only approach also enables cost-effective storage usage since the system does not need to store two copies of data in both log and data files.

Given the large application and data size, it is desirable that the system can be dynamically deployed in a cluster environment so that it is capable of adapting to changes in the workload while leveraging commodity hardware. `LogBase` adopts an architecture similar to HBase [4] and BigTable [27] where a machine in the system, referred to as tablet server, is responsible for some tablets, i.e., partitions of a table. However, `LogBase` is different in that it leverages the log as its unique data repository. Specifically, each tablet server uses a single log instance to record the data of the tablets it maintains. `LogBase` stores the log in an underlying distributed file system (DFS) that replicates data blocks across nodes in the cluster to guarantee that the probability of data loss is extremely unlikely, except catastrophic failures of the whole cluster. Consequently, `LogBase`'s capability of recovering data from machine failures is similar to traditional WAL+Data approach.

Since data, which are sequentially written into the log, are not well-clustered, it is challenging to process read operations efficiently. To solve this problem, tablet servers in `LogBase` build an index per tablet for retrieving the data from the log. Each index entry is a $< key, ptr >$ pair where *key* is the primary key of the record and *ptr* is the offset that points to the location of that record in the log. The index of each tablet can be maintained in memory since the size of an index entry is much smaller than the record's size. The in-memory index is especially useful for handling long tail requests, i.e., queries that access data not available in the cache, as it reduces I/O cost of reading index blocks. The interference of reads and writes over the log is affordable since reads do not occur frequently in write-heavy applications. As machines in commodity environments are commonly not equipped with dedicated disks for logging purpose, most scalable cloud storage systems such as HBase [4] also store both log and application data in a shared DFS and hence observe similar interferences.

`LogBase` utilizes the log records to provide multiversion data access since all data are written into the log together with their version number, which is the commit timestamp of the transactions that write the data. To facilitate reads over multiversion data, the indexes are also multiversioned, i.e., the *key* of index entries now is composed of two parts: the primary key of the record as the prefix and the commit timestamp as the suffix. Furthermore, `LogBase` supports the ability to bundle a collection of read and write operations spanning across multiple records within transaction boundary, which is an important feature that is missing from most of cloud storage systems [26].

In summary, the contributions of this chapter are as follows.

- We propose `LogBase` – a scalable log-structured database system that can be dynamically deployed in a cluster. It provides similar recovery capability to traditional write-ahead-logging approach while offering highly sustained throughput for write-heavy applications.

- We design a multiversion index strategy in `LogBase` to provide efficient access to the multiversion data maintained in the log. The in-memory index can efficiently support long tail requests that access data not available in the cache.

- We further enhance `LogBase` to support transactional semantics for read-modify-write operations and provide snapshot isolation – a widely accepted correctness criterion.

- We conducted an extensive performance study on LogBase and used HBase [4] and LRS, a log-structured record-oriented system that is modeled after RAMCloud [72] but stores data on disks, as our baselines. The results confirm its efficiency and scalability in terms of write and read performance, as well as effective recovery time in the system.

This chapter proceeds as follows. In Section 3.2, we present the design and implementation of `LogBase`. We evaluate the performance of `LogBase` in Section 3.3 and conclude the paper in Section 3.4.

3.2 Design and Implementation

In this section, we present various issues of the design and implementation of `LogBase` including data model, partitioning strategy, log repository, multiversion index, basic data operations, transaction management, and system recovery method.

3.2.1 Data Model

Cloud storage systems, as surveyed in [26], represent a recent evolution in building infrastructure for maintaining large-scale data, which are typically extracted from Web 2.0 applications. Most systems such as Cassandra [61] and HBase [4] and Dynamo [39], employ key-value model or its variants (e.g., column-based storage) and make a trade-off between system scalability and functionality. Recently, some systems such as Megastore [16] adopt a variant of the abstracted tuples model of an RDBMS where the data model is represented by declarative schemas coupled with strongly typed attributes. Pnuts [33] is another large-scale distributed storage system that uses the tuple-oriented model.

Since `LogBase` aims to provide scalable storage service for database-centric applications in the cloud, its data model is also based on the widely-accepted relational data model where data are stored as tuples in relations, i.e., tables, and a tuple comprises of multiple attributes' values. However, `LogBase` further adapts this model to support column-oriented storage model in order to exploit the data locality property of queries that frequently access a subset of attributes in the table schema. This adaptation is accomplished by the partitioning strategy presented in the below section.

3.2.2 Data Partitioning

`LogBase` employs vertical partitioning to improve I/O performance by clustering columns of a table into column groups which comprise of columns that are frequently accessed together by a set of queries in the workload. Column groups are stored separately in different physical data partitions so that the system can exploit data locality when processing queries. Such vertical partitioning benefits queries that only access a subset of columns of the table, e.g., aggregate functions on some attributes, since it saves significant I/O cost compared to the approach that stores all columns in the schema into a single physical table.

This partitioning strategy is similar to data morphing technique [53] which also partitions the table schema into column groups. Nevertheless, the main difference is that data morphing aims at designing a CPU cache-efficient column layout while the partitioning strategy in `LogBase` focuses on exploiting data locality for minimizing I/O cost of a query workload. In particular, given a table schema with a set of columns, multiple ways of grouping these columns into different partitions are enumerated. The I/O cost of each assignment is computed based on the query workload trace and the best assignment is selected as the vertical partitions of the table schema. Since we have designed the vertical partitioning scheme based on the trace of query workload, tuple re-construction is only necessary in the worst case. Moreover, each column group still embeds the primary key of data records as one of its componential columns, and therefore to reconstruct the tuple, `LogBase` collects the data in all column groups using the primary key as selection predicate.

To facilitate parallel query processing while offering scale out capability, `LogBase` further splits the data in each column group into horizontal partitions, referred to as tablets. `LogBase` designs the horizontal partitioning scheme carefully in order to reduce the number of distributed transactions across machines. In large-scale applications, users commonly operate on their own data which form an entity group or a key group [16, 36, 87]. By cleverly designing the key of records, all data related to a user could have the same key prefix, e.g., the user's identity. As a consequence, data accessed by a transaction are usually clustered on a physical machine. In this case, executing transactions is not expensive since the costly two-phase commit can be avoided.

For scenarios where the application data cannot be naturally partitioned into entity groups, we can implement a group formation protocol that enables users to explicitly cluster data records into key groups [36]. Another alternative solution is workload-driven approach for data partitioning [35]. This approach models the transaction workload as a graph in which data records constitute vertices and transactions constitute edges. A graph partitioning algorithm is used to split the graph into sub partitions while reducing number of cross-partition transactions.

3.2.3 Architecture Overview

Figure 3.1 illustrates the overall architecture of `LogBase`. In this architecture, each machine – referred to as tablet server – is responsible to maintain several tablets, i.e., horizontal partitions of a table. The tablet server records the data, which might belong to the different tablets that it maintains, into its single log instance stored in the underlying distributed file system (DFS) shared by all servers. Overall, a tablet server in `LogBase` consists of three major functional layers, including transaction manager, data access manager, and log repository.

Log Repository. At the bottom layer is the repository for maintaining log data. Instead of storing the log in local disks, the tablet servers employ a shared distributed file system (DFS) to store log files and provide fault-tolerance in case of machine failures. The implementation of Log

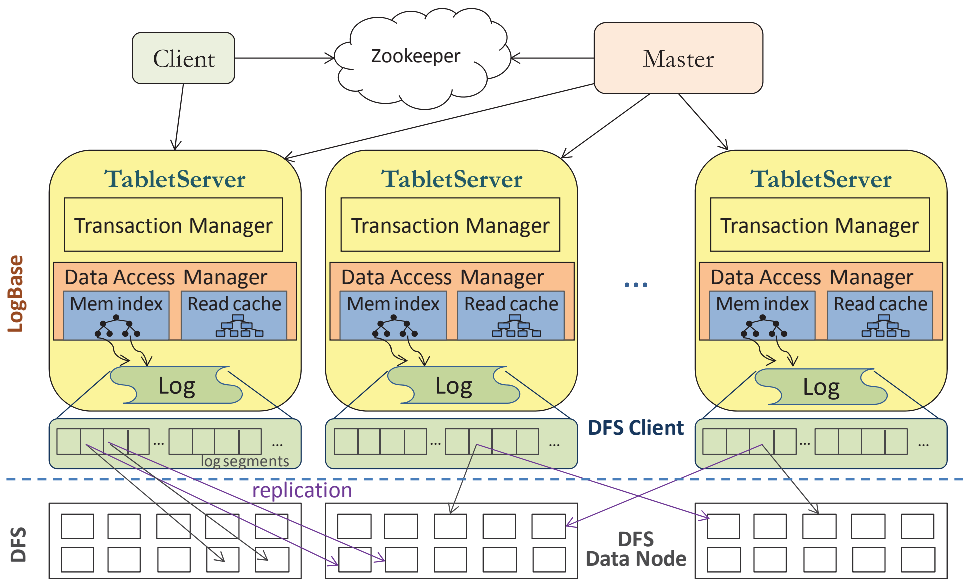


Figure 3.1: System architecture.

Repository is described in Section 3.2.4.

Data Access Manager. This middle layer is responsible to serve basic data operations including `Insert`, `Delete`, `Update`, and `Get` a specific data record. Data Access Manager also supports `Scan` operations for accessing records in batches, which is useful for analytical data processing such as programs run by MapReduce [3, 38]. In `LogBase` tablet severs employ in-memory multiversion indexes (cf. Section 3.2.5) for supporting efficient access to the data stored in the log. The processing of data operations is discussed in Section 3.2.6.

Transaction Manager. This top layer provides interface for applications to access the data maintained in `LogBase` via transactions that bundles read and write operations on multiple records possibly located on different machines. The boundary of a transaction starts with a `Begin` command and ends with a `Commit` or `Abort` command. Details of transaction management is presented in Section 3.2.7.

The master node is responsible for monitoring the status of other tablet servers in the cluster, and provides the interface for users to update the metadata of the database such as create a new table and add column groups into a table. To avoid critical point of failures, multiple instances of master node can be run in the cluster and the active master is elected via Zookeeper [24, 54], an efficient distributed coordination service. If the active master fails, one of the remaining masters will take over the master role. Note that the master node is not the bottleneck of the system since it does not lie on the general processing flow. Specifically, a new client first contacts the Zookeeper to retrieve the master node information. With that information it can query the master node to get the tablet server information and finally retrieve data from the tablet server that maintains the records of its interest. The information of both master node and tablet servers are cached for later user and hence only need to be looked up for the first time or when the cache is stale.

Although `LogBase` employs a similar architecture to HBase [4] and Bigtable [27], it introduces several major different designs. First, `LogBase` uses the log as data repository in order to remove the write bottleneck of the WAL+Data approach observed in write-heavy applications. Second, tablet servers in `LogBase` build an in-memory index for each column group in a tablet to support efficient data retrieval from the log. Finally, `LogBase` provides transactional semantics for bundled read and write operations accessing multiple records.

3.2.4 Log Repository

As discussed in Section 3.1, the approach that uses log as the unique data repository in the system benefits write-heavy applications in many ways, including high write throughput, fast system recovery and multiversion data access. Nevertheless, there could be questions about how this approach can guarantee the property of data durability in comparison to the traditional write-ahead-logging, i.e., WAL+Data approach.

Guarantee 1. ***Stable storage.*** *The log-only approach provides similar capability of recovering data from machine failures compared to the WAL+Data approach.*

Recall that in the WAL+Data approach, data durability is guaranteed with the “stable storage” assumption, i.e., the log file must be stored in a stable storage with zero probability of losing data. Unfortunately, implementing stable storage is theoretically impossible. Therefore, some methods such as RAID (Redundant Array of Independent Disks [73]) have been proposed and widely accepted to simulate stable storages. For example, a RAID-like erasure code is used to enable recovery from corrupted pages in the log repository of Hyder [22], which is a log-structured transactional record manager designed for shared flash.

To leverage commodity hardware and dynamic scalability designed for cluster environment, `LogBase` stores the log in HDFS [5] (Hadoop Distributed File System). HDFS employs n-way replication to provide data durability (n is configurable and set to 3-way replication as default since it has been a consensus that maintaining three replicas is enough for providing high data availability in distributed environments). The log can be considered as an infinite sequential repository which contains contiguous segments. Each segment is implemented as a sequential file in HDFS whose size is also configurable. We set the default size of segments to 64 MB as in HBase.

Replicas of a data block in HDFS are synchronously maintained. That is, a write operation to a file is consistently replicated to n machines before returning to users. This is equivalent to RAID-1 level or mirroring disks [73]. Further, the replication strategy in HDFS is rack-aware, i.e., it distributes replicas of a data block across the racks in the cluster, and consequently guarantees that the probability of data loss is extremely unlikely, except catastrophic failures of the whole cluster. Therefore, the use of log-only approach in `LogBase` does not reduce the capability of recovering data from machine failures compared to the other systems. Note that HBase also stores its log data (and its application data) in HDFS.

Each tablet server in `LogBase` maintains several tablets, i.e., partitions of a table, and record the log data of these tablets in HDFS. There are two design choices for the implementation of the log: (i) a single log instance per server that is used for all tablets maintained on that server and (ii) the tablet server maintains several log instances and each column group has one log instance. The advantages of the second approach include:

- **Data locality.** Since `LogBase` uses log as the unique data repository, it needs to access the log to retrieve the data. If a log instance contains only the data that are frequently access together, e.g., all rows of a column group, it's likely to improve the I/O performance for queries that only access that column group. On the contrary, in the first approach, the system needs to scan the entire log containing rows of all column groups.

- **Data recovery.** If a tablet server fails, its tablets will be assigned to other servers. In the second approach, one log represents one column group, and therefore, other servers only need to reload the corresponding index file and check the tail of that log (from the consistent point immediate after the latest checkpoint). Otherwise, in the first approach, the log has to be sorted and split by column group, and then scanned by the corresponding servers as in BigTable [27] and HBase [4].

However, the downside of the second approach is that, the underlying distributed file system has to handle many read/write connections that are used for multiple log instances. In addition, it also consumes more disk seeks to perform writes to different logs in the physically storage. Since `LogBase` aims at write-heavy applications that require sustained write throughput, we choose the first approach, i.e., each tablet server uses a single log instance for storing the data from multiple tablets that it maintains. Moreover, this approach still can support data locality after the log compaction process (cf. Section 3.2.6) which periodically scans the log, removes out-of-date data and sorts the log entries based on column group, primary key of the record, and timestamp of the write. That is, all data related to a specific column group will be clustered together after the log compaction.

Note that if each server has only one attached disk, even using a single log instance will not guarantee minimal disk seeks for writing, due to the mixture of primary replica of own writes and backup replicas from other servers (by n-way replication of HDFS). This can be solved by attaching two disks, one for primary replica and the other for backup replicas.

A log record comprises of two components $< LogKey, Data >$. The first component, $LogKey$, stores the information of a write operation, which includes log sequence number (LSN), table name, and tablet information. LSN is used to keep track of updates to the system, and is useful for checkpointing and recovery process (cf. Section 3.2.8). LSN either starts at zero or at the last known LSN persisted in the previous consistent checkpoint block. The second component, $Data$, is a pair of $< RowKey, Value >$ where $RowKey$ represents the id of the record and $Value$ stores the content of the write operation. $RowKey$ is the concatenation of the record's primary key and the column group updated by the write operation, along with the timestamp of the write. Log records are to be persisted into the log repository before write operations can return to users.

3.2.5 In-Memory Multiversion Index

Since `LogBase` records all writes sequentially in the log repository, there is no clustering property of data records stored on disks. As a result, access to data records based on their primary keys is inefficient as it is costly to scan the whole log repository only for retrieving some specific records. Therefore, `LogBase` builds indexes over the data in the log to provide efficient access to the data.

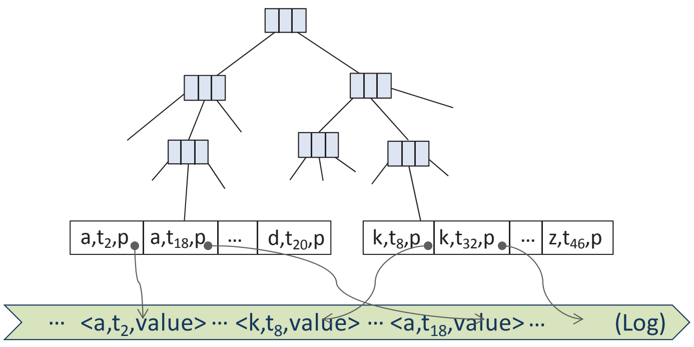


Figure 3.2: Multiversion index over the log repository.

In particular, tablet servers build a multiversion index, as illustrated in Figure 3.2, for each column group in a tablet. `LogBase` utilizes the log entries to provide multiversion data access since all data are written into the log together with their version numbers, i.e., the timestamp of the write. To facilitate reads over multiversion data, the indexes are also multiversioned. The indexes resemble B^{link}-trees [62] to provide efficient key range search and concurrency support. However, the content of index entries is adapted to support multiversion data. In our indexes, each index entry is a pair of $< IdxKey, Ptr >$. The $IdxKey$ is composed of two parts: the primary key of the record as the prefix and the timestamp as the suffix. Ptr is the offset that points to the location of a data record in the log, which includes three information: the file number, the offset in the file, the record's size.

We design an index entry as a composite value of record id and timestamp so that the search for current as well as historical versions of particular data records, which is the major access pattern in our applications, can be done efficiently. Historical index entries of a given record id, e.g., key a in Figure 3.2, are clustered in the index and can be found by performing an index search with the data key a as the prefix. Among the found entries, the one that has the latest timestamp contains the pointer to the current version of the data record in the log.

The ability to search for current and historical versions efficiently is useful for developing the multiversion concurrency control in `LogBase` (cf. Section 3.2.7). Although multiversion indexes can be implemented with other general multi-version access methods, e.g., Time-Split B-tree (TSB-tree) [63], these methods are mainly optimized for temporal queries by partitioning the index along time and attribute value dimensions, which increases the storage space and insert cost considerably.

The indexes in `LogBase` can be stored in memory since they only contain the $< IdxKey, Ptr >$ pairs whose size are much smaller than the record's size. For example, while the size of records, e.g., blogs' content or social communications, could easily exceed 1 KB, the $IdxKey$ only consumes about 16 bytes (including the record id and timestamp of long data type) and Ptr consumes about 8 bytes (including the file number and record size as short data type, and the file offset as integer data type), which makes a total size of 24 bytes each index entry. Assuming that the tablet server can reserve 40% of its 1 GB heap memory for in-memory indexes (HBase [4] uses a similar default setting for its memtables), the indexes of that server can maintain approximately 17 million entries.

There are several methods to scale out `LogBase`'s index capability. A straight-forward way is to increase either the heap memory for the tablet server process or the percentage of memory usage for indexes (or both). Another solution is to launch more tablet server processes on other physical machines to share the workload. Finally, `LogBase` can employ a similar method to log-structured merge-tree (LSM-tree) [71] for merging out part of the in-memory indexes into disks, which we shall investigate in the experiments.

A major advantage of the indexes in `LogBase` is the ability to efficiently process long tail requests, i.e., queries that access data not available in read cache. `LogBase` uses in-memory indexes for directly locating and retrieving data records from the log with only one disk seek, while in the WAL+Data approach (e.g., in HBase [4]) both application data and index blocks need to be fetched from disk-resident files, which incurs more disk I/Os.

The downside of in-memory indexes is that their content are totally lost when machines crash. To recover the indexes from machine failures, the restarted server just scans its log and reconstructs the in-memory index for the tablets it maintains. In order to reduce the cost of recovery, LogBase performs checkpoint operation at regular times. In general, tablet servers periodically flush the in-memory indexes into the underlying DFS for persistence. Consequently, at restart time the tablet server can reload the indexes quickly from the persisted index files back into memory. We describe the details of LogBase's recovery technique in Section 3.2.8.

3.2.6 Tablet Serving

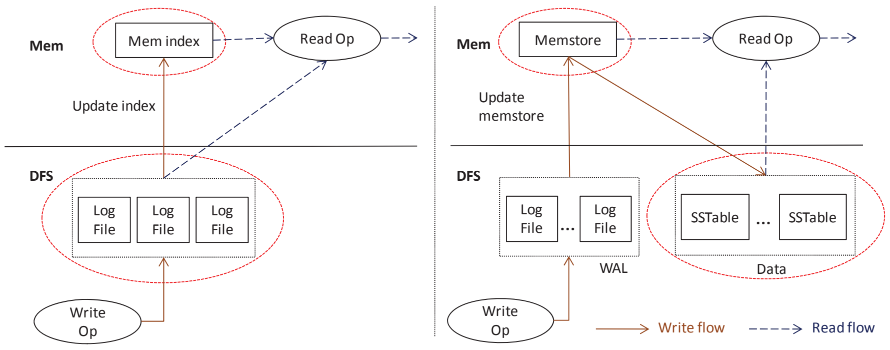


Figure 3.3: Tablet serving of LogBase (left) vs. HBase (right).

Now we present the details of a tablet server in LogBase, which uses only log files to facilitate both data access and recovery. As illustrated in Figure 3.3, each tablet server manages two major components, including (i) the single log instance (consisting of sequential log segments) which stores data of multiple tablets maintained by the server, and (ii) the memory index for each column group which map the primary key of data records to their location in the log. Another major component (not shown) is the transaction manager whose details will be described in the next section.

LogBase differs from HBase [4] on every aforementioned component. More specifically, HBase stores data in data files which are separate with the log and uses memtables to buffer recently updated data, in addition to the fact that it does not support transactional semantics for bundled read and write operations. The benefits of log-only approach compared to WAL+Data approach when serving write-heavy applications have been briefly discussed in Section 3.1. In the following, we shall describe how LogBase performs basic data operations such as write, read, delete, and scan over the tablets as well as tablet compaction operation.

Write

When a write request (`Insert` or `Update`) arrives, the request is first transformed into a log record of $< LogKey, Data >$ format, where $LogKey$ contains meta information of the write such as log sequence number, table name, and tablet information while $Data$ stores the content of the write, including the record's primary key, the updated column group, the timestamp of the write, and the new value of data. Then the tablet server writes this log record into the log repository.

After the log record has been persisted, its starting offset in the log along with the timestamp are returned so that the tablet server subsequently updates the in-memory index of the corresponding updated column group. This guarantees that the index are able to keep track of historical versions of the data records. The indexes are used to retrieve the data records in the log at later time.

In addition, the new version of data can also be cached in a read buffer (not shown in Figure 3.3) so that `LogBase` can efficiently serve read requests on recently updated data. While the in-memory index is a major component and is necessary for efficient data retrieval from the log, read buffer is only an optional component whose existence and size are configurable parameters. The read buffer in `LogBase` is different from the memtable in HBase [4] in that the read buffer is only for improving read performance while the memtable stores data and needs to be flushed into disks whenever the memtable is full, which incurs write bottlenecks in write-intensive applications.

A counter is maintained to record the number of updates that have occurred to the column group of a tablet. If the number of updates reaches a threshold, the index can be merged out into an index file stored in the underlying DFS and the counter is reset to zero. Persisting indexes into index files helps to provide a faster recovery from failures, since the tablet servers do not need to scan the entire log repository in order to rebuild the indexes. Note that the DFS with 3-way synchronous replication is sufficient to serve as a stable storage for index files (as the case of log files and discussed in Section 3.2.4).

Read

To process a `Get` request, which retrieves data of a specific record given its primary key, the tablet server first checks whether the corresponding record exists in the read buffer. If the value is found, it is returned and the request is completed. Otherwise, the server obtains the log offset of the requested record from the in-memory index. With this information, the data record is retrieved from the log repository, and finally returned to clients. By default, the system will return the latest version of the data of interest. To access historical versions of data, users can attach a timestamp t_q with the retrieval request. In this case, `LogBase` fetches all index entries with the requested key as the prefix and follows the pointer of the index entry that has the latest timestamp before t_q to retrieve the data from the log.

Meanwhile, the read buffer also caches the recent fetched record for serving possible future requests. Since there is only one read buffer per tablet server and the size of the read buffer is limited, an effective replacement strategy is needed to guarantee the read buffer is fully exploited while reducing the number of cache misses. In our implementation, we employ the LRU strategy which discards the least recently used records first. However, we also design the replacement strategy as an abstracted interface so that users can plug in new strategies that fit their application access patterns. With the use of read buffer, `LogBase` can quickly answer queries for data that have recently been updated or read, in addition to the ability to process long tail requests efficiently via in-memory indexes.

Note that the vertical partitioning scheme in `LogBase`, as discussed in Section 3.2.2, is designed based on the workload trace, and therefore most queries and updates will access data within a column group. In the case where tuple reconstruction is necessary, `LogBase` collects componential data of a record from all corresponding column groups.

Delete

A tablet server in `LogBase` performs a *Delete* operation given a record primary key in two steps. First, it remove all index entries associated with this record key from the in-memory index. By doing this all incoming queries at later time cannot find any pointer from the index in order to access the data record in the log repository. However, in the event of tablet server's restart after failures, the index is typically reloaded from the previous consistent checkpoint file, which still contains the index entries that we have attempted to remove in the first step.

Therefore, in order to guarantee durable effect of the *Delete* operation, `LogBase` performs a second step which persists a special log entry, referred to as invalidated log entry, into the log repository to record the information about this *Delete* operation. While this invalidated log entry also contains $LogKey$ similar to normal log entries, its *Data* component is set to *null* value in order to represent the fact that the corresponding data record has been deleted. As a consequence, during the restart of the tablet server, this invalidated log entry will be scanned over and its deletion effect will be reflected into the in-memory index again.

Scan

`LogBase` supports two types of scan operations, including range scan and full table scan. A range scan request takes a start key and an end key as its input. If the query range spans across tablet servers, it will be divided into subranges which are executed in parallel on multiple servers. Each tablet server process a range scan as follows. First, it traverses the in-memory index to enumerate all index entries that satisfies the query range. Then, it follows the pointers in the qualified index entries to retrieve the data from the log repository. Since the data in the log are not clustered based on the search key, it is not efficient when handling with large range scan queries. However, `LogBase` periodically performs log compaction operation which will be discussed below. After this compaction, data in the log are typically sorted and clustered based on the data key. Therefore, `LogBase` can support efficient range scan queries, i.e., clustering access on the primary key of data records, if the log compaction operation is performed at regular times.

In contrast to range scan queries, full table scans can be performed efficiently in `LogBase` without much optimization. Since full table scans do not require any specific order of access to data records, multiple log segments, i.e., log files, in the log repository of tablet servers are scanned in parallel. For each scanned record, the system checks its stored version with the current version maintained in the in-memory index to determine whether the record contains latest data.

Compaction

In the log-only approach, updates (and even deletes) are sequentially appended as a new log entry at the end of the log repository. After a period of time, there could be obsolete versions of data that are not useful for any query, but they still consume storage capacity in the log repository. Therefore, it is important to perform a vacuuming process, referred to as compaction, in order to discard out-of-date data and uncommitted updates from the log repository and reclaim the storage resources.

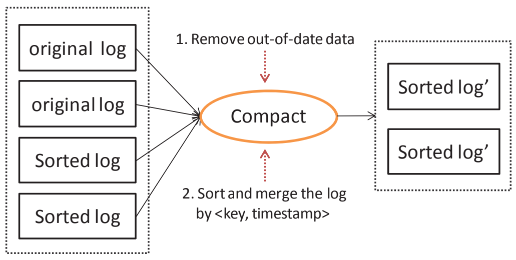


Figure 3.4: Log compaction.

Compaction could be done periodically as background process or more frequently when the system has spare CPU and I/O bandwidth. Figure 3.4 illustrates the compaction process performed by a tablet server in `LogBase`. In particular, `LogBase` performs a MapReduce-like job which takes the current log segments (some of them are sorted log segments, resulted from the previous compaction) as its input, removes all obsolete versions of data and invalidated records, and finally sorts the remaining data based on the following criteria (listed from the highest to lowest priority): table name, column group, record id, and timestamp. The result of this job is a set of sorted log segments in which data are well-clustered. Then, each tablet server builds the in-memory indexes over these new log segments. After the indexes have been built, the tablet server now can efficiently answer clients' queries on the clustered data in the sorted log segments.

Note that until this time point, old log segments and in-memory indexes are still in use and all clients' update requests from the start of the running compaction process are stored in new log segments which will be used as inputs in the next round of compaction. That is, `LogBase` can serve clients' queries and updates as per normal during the compaction process. After the compaction process has finished, i.e., the resulted sorted segments and in-memory indexes are ready, the old log segments and in-memory indexes can be safely discarded.

An additional optimization is adopted during the compaction process to decrease the storage consumption of log segments and further improve I/O performance for queries. Specifically, since the data in the resulting log segments are clustered by table name and column group already, it is not necessary to store this information in every log entries any more. Instead, the tablet server only needs to maintain a metadata which maps the table name and column group information to a list of log segments that store its data.

3.2.7 Transaction Management and Correctness Guarantees

In the previous section, we have presented `LogBase`'s basic data operations, which only guarantee single row ACID properties similar to other cloud storage systems such as Pnuts [33], Cassandra [61] and HBase [4]. We now present how `LogBase` ensures ACID [51] semantics for bundled read and write operations spanning across multiple records.

Concurrency Control and Isolation

The Rationale of MVOCC. Recall that `LogBase` is designed with a built-in function of maintaining multiversion data. In addition, the careful design of the data partitioning scheme in `LogBase`, which is based on application semantics and query workload, clusters data related to a user together, and thus reduces the contention between transactions as well as the number of distributed transactions. Consequently, we employ a combination of multiversion and optimistic concurrency control (MVOCC) to implement isolation and consistency for transactions in `LogBase`.

A major advantage of MVOCC [21] is the separation of read-only and update transactions so that they will not block each other. In particular, read-only transactions access a recent consistent snapshot of the database while update transactions perform on the latest version of the data. Therefore, read-only transactions always commit successfully, whereas an update transaction after finishing its read phase has to validate its possible conflicts with other concurrently executing update transactions before being allowed to enter the write phase.

While traditional OCC needs to maintain old write-sets of committed transactions in order to verify data conflicts, the MVOCC in `LogBase` provides another advantage that in the validation phase of update transactions, the transaction manager can use the version numbers of data records to check for conflicts with other update transactions. In particular, to commit an update transaction T, the transaction manager checks whether T's write set are updated by other concurrent transactions that have just committed by comparing the versions of the records in T's write set that T has read before (there is no blind write) with the current version of the records maintained in the in-memory indexes. If there is any change in the record versions, then the validation fails and T is restarted. Otherwise, the validation return success and T is allowed to enter the write phase and commit.

Validation with Write Locks. To avoid possible conflicts of concurrent writes, `LogBase` embeds write locks into the validation phase of MVOCC. In particular, an update transaction first executes its read phase as per normal; however, at the beginning of validation phase, the transaction manager will request write locks over the data records for its intention writes. If all the locks can be obtained and the validation succeeds, the transaction can execute its write phase, and finally release the locks. Otherwise, if the transaction manager fails to acquire all necessary write locks, it will still hold the existing locks while re-executing the read phase and trying to request again the locks that it could not get in the first time. This means that the transaction keeps pre-claiming the locks until it obtains all the necessary locks, so that it can enter the validation phase and write phase safely. Deadlock can be avoided by enforcing each transaction to request its locks in the same sequence, e.g., based on the record key's order, so that no transaction waits for locks on new items while still locking other transactions' desired items.

`LogBase` delegates the task of managing distributed locks to a separate service, Zookeeper [54, 11], which is widely used in distributed storage systems, such as Cassandra [61] and HBase [4], for providing efficient distributed synchronization. In addition, `LogBase` employs Zookeeper as a timestamp authority to establish a global counter for generating transaction's commit timestamps and therefore ensuring a global order for committed update transactions.

Snapshot Isolation in `LogBase`. The locking method during validation ensures "first-committer-wins" rule [20]. Therefore, the MVOCC in `LogBase` provides similar consistency and isolation level to standard snapshot isolation [20].

Guarantee 2. *Isolation.* *The hybrid scheme of multiversion optimistic concurrency control (MVOCC) in* `LogBase` *guarantees snapshot isolation.*

Proof Sketch: The MVOCC in LogBase is able to eliminate inconsistent reads, including "Dirty read", "Fuzzy read", "Read skew" and "Phantom", and inconsistent writes, including "Dirty write" and "Lost update", while still suffers from "Write skew" anomaly, thereby follows strictly the properties of Snapshot Isolation. □

The multiversion histories representing these phenomena when executing transactions in `LogBase` are listed below. In our notation, subscripts are used to denote different versions of a record, e.g., x_i refers to a version of x produced by transaction T_i. By convention, T_0 is an originator transaction which installs initial values of all records in the system.
Dirty read: $w_1[x_1]...r_2[x_0]...((c_1$ or $a_1)$ and $(c_2$ or $a_2)$ in any order)
Fuzzy read: $r_1[x_0]...w_2[x_2]...((c_1$ or $a_1)$ and $(c_2$ or $a_2)$– any order)
Read skew: $r_1[x_0]...w_2[x_2]...w_2[y_2]...c_2...r_1[y_0]...(c_1$ or $a_1)$
Phantom: $r_1[P]...w_2[y_2$ in $P]...c_2...r_1[P]...c_1$
Dirty write: $w_1[x_1]...w_2[x_2]...((c_1$ or $a_1)$ and $(c_2$ or $a_2)$ in any order)
Lost update: $r_1[x_0]...w_2[x_2]...w_1[x_1]...c_1$
Write skew: $r_1[x_0]...r_2[y_0]...w_1[y_1]...w_2[x_2]...(c_1$ and $c_2)$

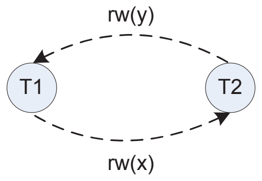


Figure 3.5: Multiversion serialization graph for write skew.

Under dependency theory [41], an edge from transaction T_1 to transaction T_2 is added into the multiversion serialization graph (MVSG) to represent their data conflicts in three scenarios: (1) ww-dependency where T_1 installs a version of x and T_2 installs a later version of x, (2) wr-dependency where T_1 installs a version of x and T_2 reads this (or a later) version of x, and (3) rw-dependency where T_1 reads a version of x and T_2 installs a later version of x.

The MVSG of "Write skew", as depicted in Figure 3.5, contains a cycle between T_1 and T_2, showing that the MVOCC in `LogBase` suffers from this anomaly. On the contrary, the MVSG of the remaining phenomena (not shown) is acyclic, which means that `LogBase` is able to prevent those inconsistent reads and inconsistent writes. Therefore, `LogBase` provides snapshot isolation semantics for read-modify-write transactions.

Since snapshot isolation is a widely accepted correctness criterion and adopted by many database systems such as PostgreSQL, Oracle and SQL Server, we hypothesize that it is also useful for large-scale storages such as `LogBase`. If strict serializability is required, read locks also need to be acquired by transactions [85], but that will affect transaction performance as read locks block the writes and void the advantage of snapshot isolation. Another method which prevents cyclic "read-write" dependency at runtime is conservative and may abort transactions unnecessarily [25].

Commit Protocol and Atomicity

Guarantee 3. ***Atomicity.*** *The* `LogBase`*'s commit protocol guarantees similar atomicity property to the WAL+Data approach.*

The commit procedure for an update transaction T proceeds as follows. After executing T's read phase, the transaction manager runs the validation algorithm to determine if T conflicts with other committed transactions or not. If the validation fails, then T is restarted. Otherwise, the transaction manager gets a commit timestamp from the timestamp authority and persists T's writes along with the commit record into the log repository. In addition, relevant in-memory index entries are updated accordingly to reflect the changes, and all the write locks held by T are released.

Note that if the transaction manager fails to persist the final commit record into the log repository (due to errors of the log), T is still not completed as in the WAL+Data approach. Although uncommitted writes could have been written to the log, they are not reflected in the index and thus cannot be accessed by users. Scan operations also check and only return data whose corresponding commit record exists. The uncommitted writes will be totally removed out of the log when the system performs log compaction. In summary, all or none of the updates of a transaction are recorded into the system, i.e., `LogBase` guarantees similar atomicity property to the WAL+Data approach.

Since the number of distributed transactions has been reduced at most by the use of smart data partitioning, the costly two-phase-commit protocol only happens in the worst case. `LogBase` further embeds an optimization technique that processes commit and log records in batches, instead of individual log writes, in order to reduce the log persistence cost and therefore improve write throughput.

3.2.8 Failures and Recovery

We have shown how `LogBase` ensures atomicity, consistency and isolation property. In the following, we present the data durability property of `LogBase`, which guarantees all modifications that have been confirmed with users are persistent in the storage.

Guarantee 4. *Durability.* *The* `LogBase`*'s recovery protocol guarantees similar data durability property to the WAL+Data approach.*

When a crash occurs, the recovery is simple in `LogBase` since it does not need to restore the data files as in the WAL+Data approach. Instead, the only instance in `LogBase` that needs to be recovered is the in-memory indexes. As a straightforward way, the restarted server can scan its entire log and rebuild the in-memory indexes accordingly. However, this approach is costly and infeasible in practice. In order to reduce the cost of recovery, `LogBase` performs checkpoint operation at regular times or when the number of updates has reached a threshold.

In the checkpoint operation, tablet servers persist two important information into the underlying DFS to enable fast recovery. First, the current in-memory indexes are flushed into index files stored in DFS for persistence. Second, necessary information, including the current position in the log and the log sequence number (LSN) of the latest write operation whose effects have been recorded in the indexes and their persisted files in the first step, are written into checkpoint blocks in DFS so that `LogBase` can use this position as a consistent starting point for recovery.

With the checkpoint information, recovery from machine failures in `LogBase` can be performed fast since it only needs to do an analysis pass from the last known consistent checkpoint towards the end of the log where the failures occurred. At restart time the tablet server can reload the indexes quickly from the persisted index files back into the memory. Then a redo strategy is employed to bring the indexes up-to-date, i.e., the tablet server analyzes the log entries from the recovery starting point and updates the in-memory indexes accordingly. If the LSN of the log entry is greater than the corresponding index entry in the index, then the pointer in the index entry is updated to this log address. Performing redo is sufficient for system recovery since `LogBase` adopts optimistic concurrency control method, which defers all modifications until commit time. All uncommitted log entries are ignored during the redo process and will be discarded when the system performs log compaction. In addition, in the event of repeated restart when a crash occurs during the recovery, the system only needs to redo the process.

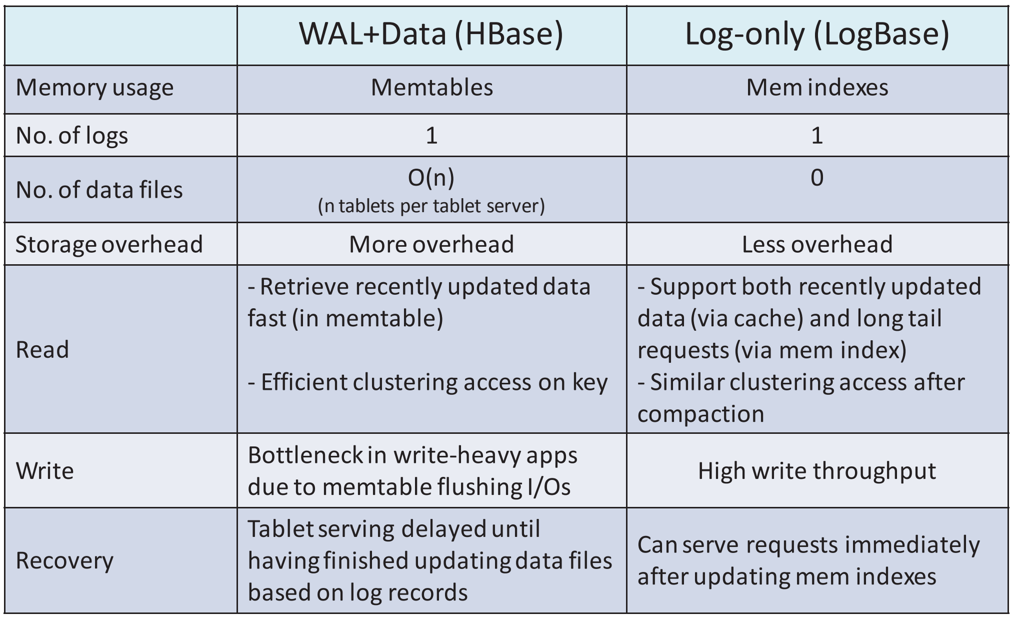
Note that if a tablet server fails to restart within a predefined period after its crash, the master node will consider this as permanent failures and re-assign the tablets maintained by this failed server to other healthy tablet servers in the system. The log of the failed servers, which is stored in the shared DFS, is scanned (from the consistent recovery starting point) and split into separate files for each tablet according to the tablet information in the log entries. Then the healthy tablet servers scan these additional assigned log files to perform the recovery process as discussed above.

3.2.9 Discussion

We now discuss the advantages of log-only approach for write-heavy applications. In particular, we look especially at the WAL+Data approach used in HBase [4] and compare it with the log-only used in `LogBase`. Figure 3.6 summarizes the comparison in term of various aspects such as storage overhead, write and read performance, and recovery time.

Both approaches need similar number of I/Os and storage consumption for logging operation, and similar memory usage (memtables for buffering data in HBase, and in-memory indexes for locating data in `LogBase`). Nevertheless, the WAL+Data approach in HBase eventually incurs more storage overhead due to the use of additional storage for application data. More importantly, its frequent flushing of memtables into physical storage, which is the norm in write-heavy applications, results in write bottlenecks. On the contrary, the log-only approach in `LogBase` avoids such cost and therefore is able to provide the desired high write throughput.

Retrieving data that have recently been updated is fast in both approaches



	WAL+Data (HBase)	Log-only (LogBase)
Memory usage	Memtables	Mem indexes
No. of logs	1	1
No. of data files	O(n) (n tablets per tablet server)	0
Storage overhead	More overhead	Less overhead
Read	- Retrieve recently updated data fast (in memtable) - Efficient clustering access on key	- Support both recently updated data (via cache) and long tail requests (via mem index) - Similar clustering access after compaction
Write	Bottleneck in write-heavy apps due to memtable flushing I/Os	High write throughput
Recovery	Tablet serving delayed until having finished updating data files based on log records	Can serve requests immediately after updating mem indexes

Figure 3.6: WAL+Data vs. Log-only.

since they are cached in memory. However, for long tail requests that do not query recently updated data, the log-only approach can be more efficient since it uses in-memory indexes for directly locating and retrieving data from the log while in the other approach, application data and its index blocks need to be fetched from data files, which incurs more disk I/Os. Note that after a log compaction process (cf. Section 3.2.6), data of a table in the log are clustered based on its primary key, and hence the log-only approach can support similar data access to the WAL+Data approach.

3.3 Performance Evaluation

3.3.1 Experimental Setup

Experiments were performed on an in-house cluster including 24 machines, each with a quad core processor, 8 GB of physical memory, 500 GB of disk capacity and 1 gigabit ethernet. `LogBase` is implemented in Java, inherits basic infrastructures from HBase open source, and adds new features for log-structured storages including access to log files, in-memory indexes, log compaction, transaction management and system recovery. We compare the performance of `LogBase` with HBase (version 0.90.3). All settings of HBase are kept as its default configuration, and `LogBase` is configured to similar settings. Particularly, both systems use 40% of 4 GB heap memory for maintaining in-memory data structures (the memtables in HBase and in-memory indexes in `LogBase`), and 20% of heap memory for caching data blocks. Both systems run on top of Hadoop platform (version 0.20.2) and store data into HDFS. We keep all settings of HDFS as default, specifically the chunk size is set to 64 MB and the replication factor is set to 3.

Each machine runs both a data node and a tablet server process. The size of datasets is proportional to the system size, and for every experiment we bulkload 1 million of 1KB records for each node (the key of each record takes its value from $2 * 10^9$ which is the max key in YCSB benchmark [34]). For scalability experiments, we run multiple instances of benchmark clients, one for each node in the system. Each benchmark client submits a constant workload into the system, i.e., a completed operation will be immediately followed by a new operation. The benchmark client reports the system throughput and response time after finishing a workload of 5,000 operations. Before running every experiments, we execute about 15,000 operations on each node to warm up the cache. The default distribution for the selection of accessed keys follows Zipfian distribution with the co-efficient set to 1.0.

3.3.2 Micro-Benchmarks

In this part, we study the performance of basic data operations including sequential write, random read, sequential scan and range scan of `LogBase` with a single tablet server storing data on a 3-node HDFS. We shall study the performance of `LogBase` with mixed workloads and bigger system sizes in the next section.

Write Performance

Figure 3.7 plots the write overhead of inserting 1 million records into the system. The results show that `LogBase` outperforms HBase by 50%. For each insert operation, `LogBase` flushes it to the log and then update the memory index. It thus only writes the data to HDFS once. On the contrary, besides persisting the log information (which includes the record itself) into HDFS, HBase has to insert the record into a memtable, which will be written to the data file in

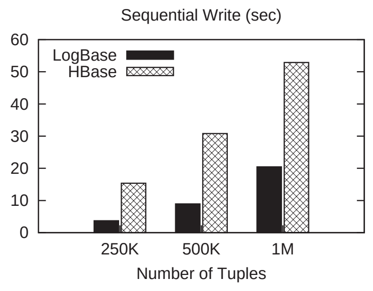


Figure 3.7: Write performance.

HDFS when the memtable is full (64 MB as default setting). As a result, HBase incurs more write overhead than `LogBase`.

Random Access Performance

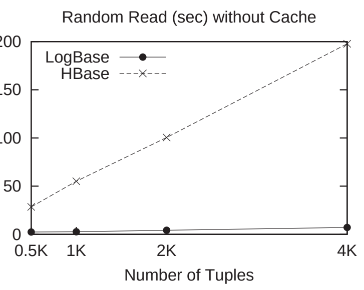


Figure 3.8: Random access (without cache).

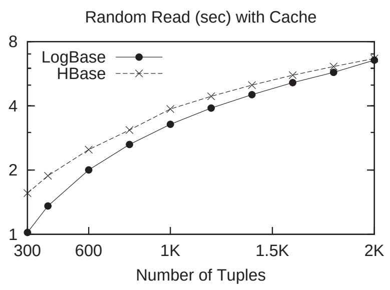


Figure 3.9: Random access (with cache).

Figure 3.8 shows the performance of random access without any cache used in both systems. The performance of `LogBase` is superior to HBase, because `LogBase` maintains a dense in-memory index and each record has a corresponding index entry containing its location in the log. With this information, `LogBase` is able to seek directly to the appropriate position in the log and retrieve the record. In contrast, HBase stores separate sparse block indexes in different data files, and hence after seeking to the corresponding block in one data file, it loads that block into memory and scan the block to get the record of interest. Further, the tablet server in HBase has to check its multiple data files in order to get the proper data record. Therefore, `LogBase` can efficiently support long tail requests that access data not available in the cache.

As shown in Figure 3.9, the performance gap between `LogBase` and HBase reduces when the block cache is adopted in the system. The main reason is that, if the block containing the record to be accessed is cached from previous requests, HBase does not need to seek and read the entire block from HDFS. Instead, it only reads the proper record from the cached block. Note that with larger data domain size in distributed YCSB benchmark as will be discussed in the next section, the cache has less effect and `LogBase` provides better read latency for the support of in-memory indexes.

Scan Performance

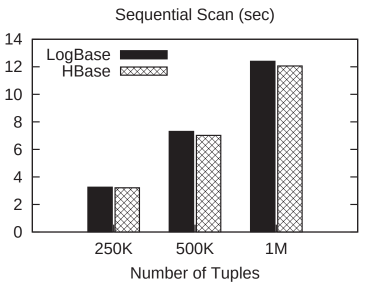


Figure 3.10: Sequential scan.

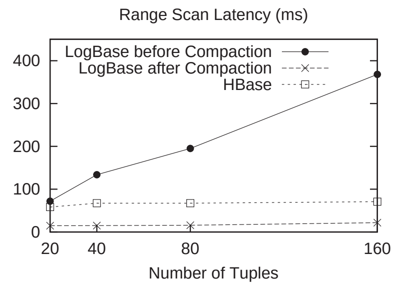


Figure 3.11: Range scan.

Sequential scan. Figure 3.10 illustrates the result of sequential scan the entire data. The performance of `LogBase` is slightly slower than HBase. `LogBase` scans the log files instead of the data files as HBase, and each log entry contains additional log information besides the data record such as the table name and column group. As such, a log file has larger size than a data file and `LogBase` has to spend slightly more time to scan the log file.

Range scan. The downside of `LogBase` is that it is not as efficient as HBase when processing range scan query as shown in Figure 3.11. In HBase, data in memtables are kept sorted by key order and persisted into data files, and hence facilitates fast range scan query. `LogBase`, on the contrary, sequentially writes data into the log without any clustering property and might need to perform multiple random access to process a single range scan query. However, it is notable that after the compaction process, data in the log are well-clustered and `LogBase` is able to provide even better range scan performance than HBase for its ability to load the correct block quickly with the support of dense in-memory indexes.

3.3.3 YCSB Benchmark

In the following, we examine the efficiency and scalability of `LogBase` with mixed workloads and varying system sizes using YCSB benchmark [34]. The system size scales from 3 to 24 nodes and two write-heavy mix workloads (95% and 75% of update in the workload) are tested.

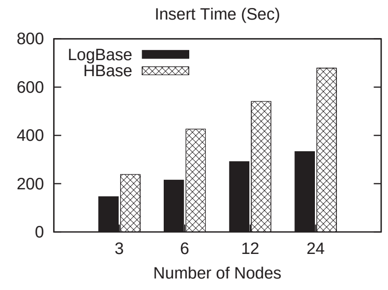


Figure 3.12: Data loading time.

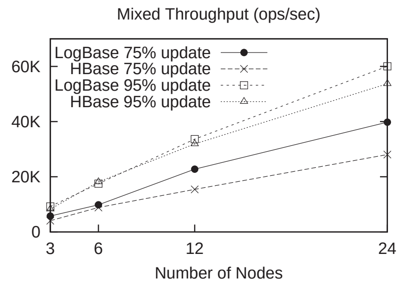


Figure 3.13: Mixed throughput.

In the loading phase of the benchmark, multiple instances of clients are launched to insert benchmark data in parallel. Similar to the result of sequential write in the micro-benchmark, Figure 3.12 shows that `LogBase` outperforms HBase when parallel loading data and only spends about half of the time to insert data. This confirms that `LogBase` can provide highly sustained throughput for write-heavy environments.

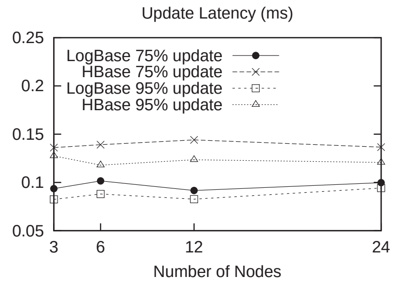


Figure 3.14: Update latency.

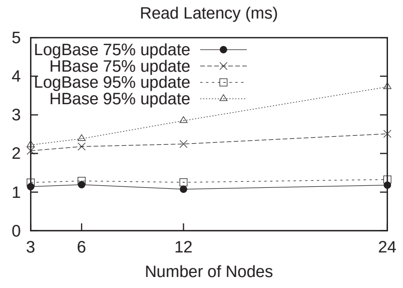


Figure 3.15: Read latency.

In the experiment phase, the benchmark client at each node will continuously submit a mixed workload into the system. An operation in this workload either reads or updates a certain record that has been inserted in the loading phase. The system overall throughput with different mixes is plotted in Figure 3.13 and the corresponding latency of update and read operations is shown in Figure 3.14 and Figure 3.15 respectively. The results show that both `LogBase` and HBase achieve higher throughput with the mix that has higher percentage of update since both systems perform write operations more efficient than read operations.

In addition, for each mix, LogBase achieves higher throughput than HBase for its ability to support both write and read efficiently. In HBase, if the memtable is full and a minor compaction is required, the write has to wait until the memtable is persisted successfully into HDFS before returning to users and hence the write response time is delayed. LogBase provides better read latency for the support of in-memory indexes as we have shown in the micro-benchmarks. Although HBase employs cache to improve read performance, the cache has less effect in this distributed experiment since both data domain size and experimental data size are large, which affects read performance.

Figure 3.14 and Figure 3.15 also illustrate the elastic scaling property of LogBase where the system scales well with flat latency. That is, the more workload can be served by adding more nodes into the system.

3.3.4 TPC-W Benchmark

In this experiment, we examine the performance of LogBase when accessing multiple data records possibly from different tables within the transaction boundary. In particular, we experiment LogBase with TPC-W benchmark which models a webshop application workload. The benchmark characterizes three typical mixes including browsing mix, shopping mix and ordering mix that have 5%, 20% and 50% update transactions respectively.

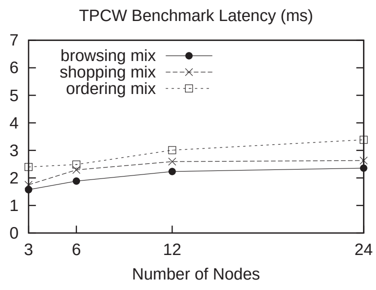


Figure 3.16: Transaction latency.

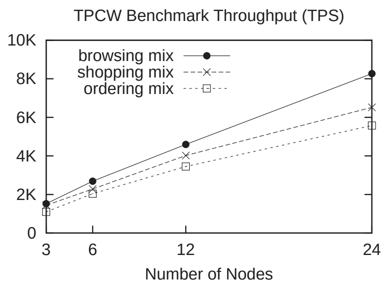


Figure 3.17: Transaction throughput.

A read-only transaction performs one read operation to query the details of a product in the item table while an update transaction executes an order request which bundles one read operation to retrieve the user's shopping cart and one write operation into the `orders` table. Each node in the system is bulk loaded with 1 million products and customers before the experiment. We stress test the system by using a client thread at each node to continuously submit transactions to the system and then benchmark the transaction throughput and latency.

As can be seen in Figure 3.16, under browsing mix and shopping mix, `LogBase` scales well with nearly flat transaction latency when the system size increases and as a result, the transaction throughput (shown in Figure 3.17) scales linearly under these two workloads. The low overhead of transaction commit is attributed to this result since in these two workloads, most of the transactions are read-only and always commit successfully without the need of checking conflicts with other transactions for the use of MVOCC.

3.3.5 Checkpoint and Recovery

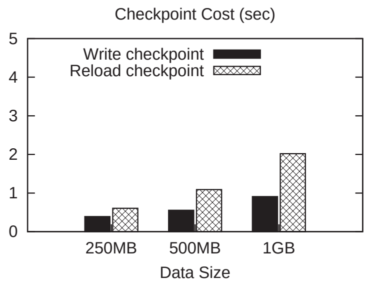


Figure 3.18: Checkpoint cost.

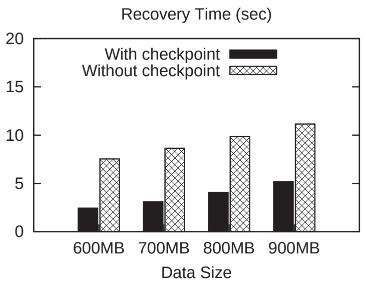


Figure 3.19: Recovery time.

We now study the cost of checkpoint operation and the recovery time in a system of 3 nodes. Figure 3.18 plots the time to write a checkpoint and reload a checkpoint with varying thresholds at which a tablet server performs the checkpoint operation. `LogBase` takes less time to write a checkpoint (persist in-memory indexes) than to reload a checkpoint (reload the persisted index files into memory) because HDFS is optimized for high write throughput. This is useful because checkpoint writing is to be performed more frequently in `LogBase`, whereas checkpoint loading only happens when the system recovers from tablet servers' failures.

The time to recover varying amount of data maintained by a failed tablet server is shown in Figure 3.19. The checkpoint was taken at a threshold of 500 MB before we purposely killed the tablet server when its amount of data reached 600 MB to 900 MB. The results show that the recovery time in the system with checkpoint is significantly faster than without checkpoint. In the former approach, the system only needs to reload the checkpoint and scan a little additional log segments after the checkpoint time to rebuild the in-memory indexes, whereas in the latter approach the system has to scan the entire log segments.

`LogBase` does not support as efficient recovery time as RAMCloud [72] because the two systems make different design choices for targeting at different environments. In RAMCloud, both indexes and data are entirely stored in memory while disks only serve as data backup for recovery purpose. Therefore, RAMCloud backups log segments of a tablet dispersedly to hundreds of machines (and disks) in order to exploit parallelism for recovery. In contrast, `LogBase` stores data on disks and hence cannot scatter log segments of a tablet to such scale in order to favor recovery as it would adversely affect the write and read performance of the system.

3.3.6 Comparison with Log-Structured Systems

Recent scalable log-structured record-oriented systems (LRS) such as RAMCloud [72] and Hyder [22] target at different environments with `LogBase`. Specifically, RAMCloud stores its data and indexes entirely in memory while Hyder scales its database in shared-flash environments without data partitioning. Therefore, we cannot compare their performance directly with `LogBase`. Here, for comparison purpose as well as exploring the opportunity of scaling the indexes beyond memory, we examine a system, referred to as LRS, which has a distributed architecture and data partitioning strategy similar to RAMCloud and `LogBase` but stores data on disks and indexes them with log-structured merge trees (LSM-tree) [71] to deal with scenarios where the memory of tablet servers is scarce. Particularly, in this experiment we use LevelDB [1], a variant LSM-tree open source by Google, with all settings kept as default.

The results of comparison between `LogBase` and LRS in a system of 3 nodes are shown in Figure 3.20, Figure 3.21, and Figure 3.22 respectively for sequential write, random access, and sequential scan. The comparison results with varying system sizes are also plotted in Figure 3.23. Overall, the sequential write and

[1] http://code.google.com/p/leveldb/

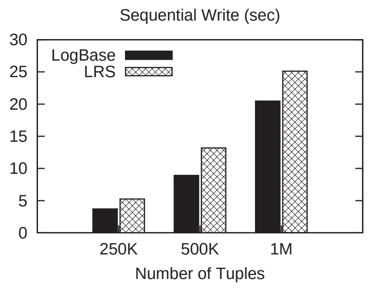


Figure 3.20: Sequential write.

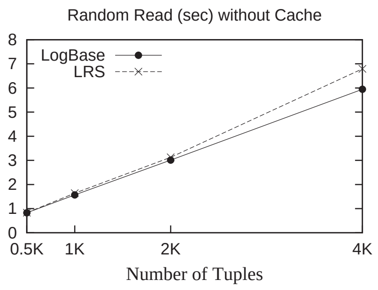


Figure 3.21: Random access.

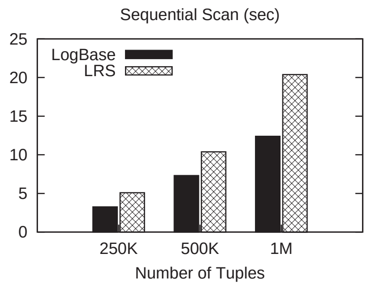


Figure 3.22: Sequential scan.

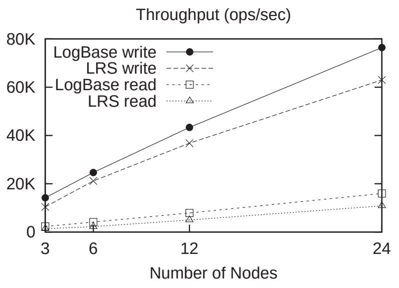


Figure 3.23: Throughput.

random access performance of LRS are only slightly lower than that of `LogBase` because LevelDB is highly optimized for a variety of workloads and can provide efficient write and read performance with moderate write and read buffer (4 MB and 8 MB respectively in the experiment). This leads us to conclude that it is possible for `LogBase` to scale its indexes beyond memory (by the use of LSM-trees) without paying much cost of reduction in the system throughput.

`LogBase` also achieves higher sequential scan performance than LRS. Recall that for each scanned record, the system needs to check its stored version against the current version maintained in the indexes to determine whether the record contains the latest data. Such cost of accessing indexes is attributed to the difference in the scan performance of the two systems. Note that after log compaction, historical versions of a record are clustered together and hence the number of version checking with indexes is minimized, which would reduce the scan performance gap.

3.4 Summary

We have introduced a scalable log-structured database system called `LogBase`, which can be elastically deployed in the cloud and provide sustained write throughput and effective recovery time in the system. The in-memory indexes in `LogBase` support efficient data retrieval from the log and are especially useful for handling long tail requests. `LogBase` provides the widely accepted snapshot isolation for bundled read-modify-write transactions. Extensive experiments on an in-house cluster verifies the efficiency and scalability of the system. Our future works include the design and implementation of efficient secondary indexes and query processing for `LogBase`.

CHAPTER 4

INDEXING OBSERVATIONAL DATA IN LOG STORE

Huge amounts of data are being generated by sensing devices every day, recording the status of objects and the environment. Such observational data is widely used in scientific research. As the capabilities of sensors keep improving, the data produced are drastically expanding in precision and quantity, making it a write-intensive domain. Log-structured storage is capable of providing high write throughput, and hence is a natural choice for managing large-scale observational data.

In this chapter, we propose an approach to indexing and querying observational data in log-structured storage. Based on key traits of observational data, we design a novel index approach called the `CR-index` (Continuous Range Index), which provides fast query performance without compromising write throughput. It is a lightweight structure that is fast to construct and often small enough to reside in RAM. Our experimental results show that the `CR-index` is superior in handling observational data compared to other indexing techniques. While our focus is scientific data, we believe our index will be effective for other applications with similar properties, such as process monitoring in manufacturing.

4.1 Introduction

Humankind has a rapidly growing ability to digitize the real-world. The variety of entities whose state can be monitored continuously is ever increasing, and spans from microscopic to macroscopic scales: individual molecules, single cells, electronic devices, wild life, automobiles, dams, oceans and even distant stars. More and more sensors are gathering continuous observations of physical variables such as temperature, humidity and velocity. Such data collection is now ubiquitous in many fields of scientific research.

Multiple trends contribute to increases in sensor data rates. Sensors are increasing in resolution temporally, spatially and the bits of precision captured. Hence individual sensors generate data at higher rates. Further, instrument packages are carrying more kinds of sensors, as devices appear for measuring a broader range of physical quantities. Finally, decreasing price and increasing power efficiency means more sensors can be deployed in more places for longer periods of time. These trends make observational data management write-intensive, demanding data storage with high write-throughput, to capture these records in a timely manner. An additional challenge is indexing newly arrived data quickly while providing efficient querying.

Log-structured storage (log-store) is amenable to handling such write-intensive scenarios. A log-store appends newly arrived data to the end of a log file, rather than seeking specific positions on disk for each record. Compared with in-place-update storage, log-store provides higher write throughput by avoiding random I/Os.

This chapter focuses on storing observational data in log-store and indexing it efficiently by exploiting its traits, including:

- **No update.** An observational record is inherently immutable. Each record has an unique *observation time* attribute. Complete historical data are required for diverse analysis tasks.
- **Continuous change.** Most physical variables have values that change continuously at some maximal rate. If frequent observations are taken, we expect successive readings to be bounded by some maximal change.
- **Potential discontinuities.** Though ideal data should be continuous, gaps could arise from noise, data loss, or combining readings from multiple sensors.

Index structures play an important role in supporting queries. Traditional record-level indexes, such as B$^+$-trees [32] and LSM-trees [71], incur significant index maintenance cost. The random I/Os due to updates render B$^+$-trees impractical for write-intensive workloads. Although LSM-trees avoid random I/Os, the cost of maintaining a large number of index entries is still considerable. Since these structures have not been designed to exploit the characteristics of observational data and its applications, they may not scale up well.

Current state-of-the-art techniques for storing observational data do not take high-throughput workloads into account. Some real data observation systems such as CMOP [1] utilize a combination of RDBMS and netCDF [9] data files to manage data. Our approach stores the observational data in log-store, which provides superior performance for write-heavy workloads. In log-store, records are ordered by arrival time, which correlates strongly with observation time. Thus, for queries on observation time, access methods based on physical order perform well (and can be further improved through off-line reorganization as discussed in Chapter 3). However, queries on observational data will often include conditions on the measured variables. Because of the data continuity, their values are *locally correlated* with observation time (and hence with arrival time). Our approach exploits this correlation to provide lightweight indexing on observational data as it is stored. We group successive records into blocks. Each block is summarized by a value range, which is compact and can be computed quickly. We accommodate the inevitable gaps by detecting them during query processing and avoiding them on subsequent queries.

Another trait of observational data that we can exploit is *spatial correlation* of two readings. The same physical variable sensed in two nearby locations is likely to be similar. For example, two temperature sensors at the same point in a river, but at different depths, are likely to report similar readings (or at least increase and decrease together). Given the large number of sensors in some deployments, it could negate some of the benefits of log-store if readings for each are stored in a separate file. Our method gives reasonable performance if readings from correlated sensors are merged, with gap-detection methods handling periods of divergence.

Our contributions include:

- We proposed a scheme for storing observational data in log-store that preserves data locality to facilitate indexing. The data organization provides optimization opportunities for reducing both write and read I/O costs.

- We designed a novel, lightweight pruning-based index structure for range queries, tailored for log-store, supporting efficient sequential I/Os. It lowers maintenance costs by taking full advantage of observational-data traits.
- We conducted an extensive experimental evaluation on two real-world observational datasets that compares our solution to traditional record-level indexes. The results confirm both low write overhead and query efficiency.

The rest of this chapter is organized as follows. In Section 4.2, we provide background on observational data and storage choices. Section 4.3 presents a scheme for storing data. In Section 4.4, we present the design of our indexing structure. We evaluate the performance in Section 4.5. Summaries are given in Sections 4.6.

4.2 Preliminaries

This section presents some characteristics and applications of observational data and common query types. We recall our log-structured storage system, `LogBase`, which is the choice of storage in our implementation.

4.2.1 Scientific Data Analysis

Many scientific analysis applications entail monitoring of correlations among multiple physical variables using diverse sensors. For example, coastal-margin observation deploys multiple underwater sensors at different sites and depths, gathering information such as water temperature, salinity and oxygen saturation. Scientific data captured in this manner, which we call *observational data*, have special traits mentioned previously. Its most distinctive characteristic is that records' values are changing continuously. The inherent continuity can be captured by two key concepts: *continuous variable* and *continuous measurement*.

Continuous variable. An observational variable can be expressed as a function $f(x)$ with respect to time x. If the function $f(x)$ is continuous, for any value v where $f(x_1) \leq v \leq f(x_2)$, there exists an x', $x_1 \leq x' \leq x_2$, where $f(x') = v$, by the Intermediate Value Theorem.

Continuous measurement. *Continuous measurements* are a series of frequent observations on a continuous variable. If the change rate of the variable is bounded by R and samples are taken every U time units, consecutive measurements will differ by no more than $m_x = U \cdot R$. Thus, if we have measurements m_1 and m_2, we expect to have at least $\frac{|m_1 - m_2|}{m_x}$ intermediate values measured between them.

These two conditions often hold for observational data from the natural world, though our index method does not depend on these assumptions for correctness. As long as jumps and gaps are not too frequent, we maintain efficiency.

Basic Query Formats

We provide SQL expressions for the basic query formats we address. A *time-range query* specifies a time period in which some attributes are requested, for example:

SELECT $T.\mathcal{A}$ FROM $Table\ T$
WHERE $T.t \geq startTime$ and $T.t \leq endTime$
ORDER BY $T.t$

Here $\mathcal{A}$ is the set of requested attributes and T is the logical table. The result set provides the trends for observed variables, such as the salinity versus time plot in Figure 4.1 (from observation station SATURN-01 in the CMOP [1] observatory). Data might be used for other kinds of analysis, e.g., correlation tests between two variables.

A *value-range query* specifies a value range on an attribute, for example:

SELECT $T.\mathcal{A}$ FROM $Table\ T$
WHERE $(T.a \geq minV$ and $T.a \leq maxV)$
ORDER BY $T.t$

Here a is the constrained attribute in $\mathcal{A}$. Such a query can be used to monitor a variable for abnormal ranges, then collect other values from the same periods. For instance, we can monitor the sensor in Figure 4.1 for salinity above 32.0, then analyze how such periods of high salinity correlate with oxygen saturation and acidity.

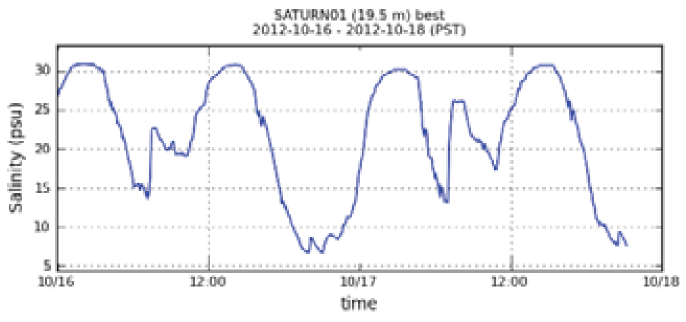


Figure 4.1: CMOP SATURN-01 salinity trend.

Our work focuses on supporting basic query types. Most complicated multi-attribute queries are extensions and combinations of basic queries. We will discuss them in Section 4.4.6.

Secondary Indexes on Observational Data

A typical observational record has a set of attributes representing different physical variables, in addition to an observation timestamp. In order to provide good performance for a variety of complicated queries, most attributes should be indexed. However, maintaining multiple conventional secondary indexes is costly. We expect our lightweight indexing mechanism to be a superior choice for keeping a large number of secondary indexes, as it will not consume much time nor space, while providing significant query acceleration. We also accommodate joint indexing of correlated sources.

4.2.2 Log-Structured Storage

In a typical log-store, log files are the only repository for upper-layer application data. Arriving data are simply appended to log files, rather than written to specific locations, thus improving write throughput. There are two types of log-store: *ordered* log-store and *unordered* log-store.

Storage Types

In *ordered* log-store, such as HBase [4], data are first written to buffers in RAM. Periodically, they are sorted by key and flushed to disk. Therefore, the keys of the records stored on disk are batch-increasing, which facilitates subsequent search for a requested key.

In contrast, *unordered* log-store never sorts the data, appending them immediately. No separate log is required for recovery, since the application-data file itself is a log. The improvement of write throughput compared to ordered log-store can hurt read performance, as the results for a query may be scattered through files. It is challenging to support efficient range queries on unordered log files. However, the characteristics of observational data mean that unordered log-store could provide good read performance.

LogBase

For implementing data storage and indexing, we briefly recall the basics of `LogBase`. Each machine in the system is a tablet server, responsible for one or more partitions of a table. Its data model is basically relational, where each record has a primary key and several attributes. Physically, each record is decomposed as a set of *cells*. A cell is the basic writable unit, structured as:

(KEY, ATTRIBUTE, VALUE, TIMESTAMP)

The *key*, *attribute* and *value* fields describe one attribute of a record. Once a record arrives, its attributes are divided into separate cells and appended to the log. When part of a record is requested, `LogBase` fetches relevant cells via an in-memory primary index on the *key* field and combines them. The *timestamp* field is hidden and set by the system for recovery and multi-version control.

In addition, `LogBase` is column-oriented by providing a logical field *group*. Attributes in different groups will be stored in different tablet servers.

4.3 Storing Observational Data

We first present the logical view of observational records in storage, then their physical organization in files and the benefits of that organization.

4.3.1 Logical View

Observational data in different scenarios might vary in many aspects, such as the number of variables and active sensors. Figure 4.2 shows an instance of a generic schema, describing coastal data for a fixed station with sensors at several depths. The whole data set is viewed as a flat table in which all records

TIME	**SENSOR ID**	**GROUP** Water		**GROUP** Air
		ATTRIBUTE Salinity	**ATTRIBUTE** Oxygen	**ATTRIBUTE** Air temperature
9:01	depth 0m	16.32	3.36	7.05
9:01	depth 2.4m	22.38	3.28	
9:02	depth 0m	16.14		6.98
9:02	depth 8m	29.01	2.97	

Figure 4.2: Schema logical view.

are ordered by observation time. The primary key is the combination of *sensor ID* and *time*. The *sensor ID* indicates the device from which the record is collected, distinguishing records from different sensors. In the example, we identify sensors by depth. Sensors are free to join or leave the system without affecting the schema.

Records with same *sensor ID* are identified by the *time*, which indicates when they were collected. Thus, a record is the ensemble of all observed variables for a sensor at a time. In Figure 4.2, some cells are empty. Empty cells are common, as values could be missing due to environmental conditions or device failures. For example, a sensor under water cannot detect air temperature.

The *group* is optional for column-oriented storage and reflects a column partition. For example, *Salinity* and *Oxygen* might be in the same group, since they are often queried together. In fact, our index is not limited to such storage. In record-oriented storage, if only a column subset is involved, the system can materialize part of the data to optimize access cost. The only worry is that if record size keeps growing, the access cost might be high. Our goal is to reduce index-maintenance cost compared to conventional methods. In the case that the record size is extremely large, the index cost will be relatively lower, and hence conventional record-level indexes are efficient enough. This situation is not the application scenario we target.

4.3.2 Physical View

Recall that `LogBase` splits records into attribute cells before appending them to the log, with different groups in different tablets. Figure 4.3 shows the physical organization of the records in Figure 4.2, for the *Water* group.

The four fields in a cell make it self-contained, allowing multiple sources

KEY	ATTRIBUTE	VALUE	TIMESTAMP
depth 0m	Salinity	16.32	9:01
	Oxygen	3.36	
depth 2.4m	Salinity	22.38	9:01
	Oxygen	3.28	
depth 8m	Salinity	29.01	9:02
	Oxygen	2.97	
depth 0m	Salinity	16.14	9:02

Figure 4.3: Schema physical view.

stored in one file. All cells of a record are stored contiguously in one atomic operation. Thus, it is simple to reassemble a record from its cells. When there are multiple attributes in a record, only the first cell requires a non-empty *key* and *timestamp*.

Since observational data has a *time* field and the storage system provides a similar *timestamp* component, we extend this component to keep two versions for each cell: a physical version and a logical version. The physical version keeps the system time for failure recovery, while the logical version keeps the observation time from the sensor side. They have different meanings, but are closely correlated. Assuming records from the same senor always arrive in order, for two records r_1,r_2 that have $r_1.logicalTime < r_2.logicalTime$, we have $r_1.physicalTime < r_2.physicalTime$.

Data from different sources might not strictly adhere to this property. However, we can still expect them to be roughly ordered. Data disorder will be discussed in Section 4.4.3.

4.3.3 Observational Data Locality

In general, the append-only strategy hurts read performance, as no data locality exists. In observational data analysis, however, the data-access pattern has inherent properties that provide considerable data locality in log-store.

The *time-ordered property* says that when a record is accessed, the succeeding records are likely to be requested in (logical) time order. It is implicit in time-range queries. In log-store, since records are in insertion order, once the first record is located, the following results will be in subsequent physical disk blocks. A sequential scan is sufficient to access the entire result set. Sequential scan is an efficient process, as it eliminates disk-seek and exploits high bandwidth.

The *value-correlated property* states that when a record is accessed, the records whose values are close to this record's might also be requested. As observational data is seldom retrieved by exact equality (because they are floating-point numbers), we expect values to be returned by range, as in value-range queries. Due to the continuity trait, once a record is inside the range, surrounding records will also lie in the range with high probability. Therefore, a log-store provides partial data locality for such range queries. Although the results are not entirely located together, they are likely clustered into sequences on the disk.

4.4 Indexing Observational Data

This section presents our indexing method for range queries on attributes of observational data in a log-store. First, we introduce the idea of a pruning-based indexing structure, which locates data blocks that may contain data of interest. After that, we propose optimization on the basic structure. At last, we discuss the extensions for processing multi-attribute queries.

4.4.1 The `CR-index` Structure

The advantage of log-store is its excellent write performance. Therefore, heavy index maintenance works against the goal of supporting write-intensive workloads. To reduce the index cost, we propose a pruning-based index method, called the *Continuous Range Index* (`CR-index`). This light-weight index exploits the traits in observational data.

The value-correlated property implies that a seek in the log can potentially yield many results. Therefore, we do not need to locate qualifying records individually, as long as we can identify regions containing results. Our main idea is to group successive records into blocks, which are the atomic units for indexing and retrieval. Each block is summarized by a value range using a *boundary pair*. When the value range of a block intersects with a query range, there is high likelihood of results in the block based on the continuous-change trait of observational data. In that case, the whole block will be fetched and scanned.

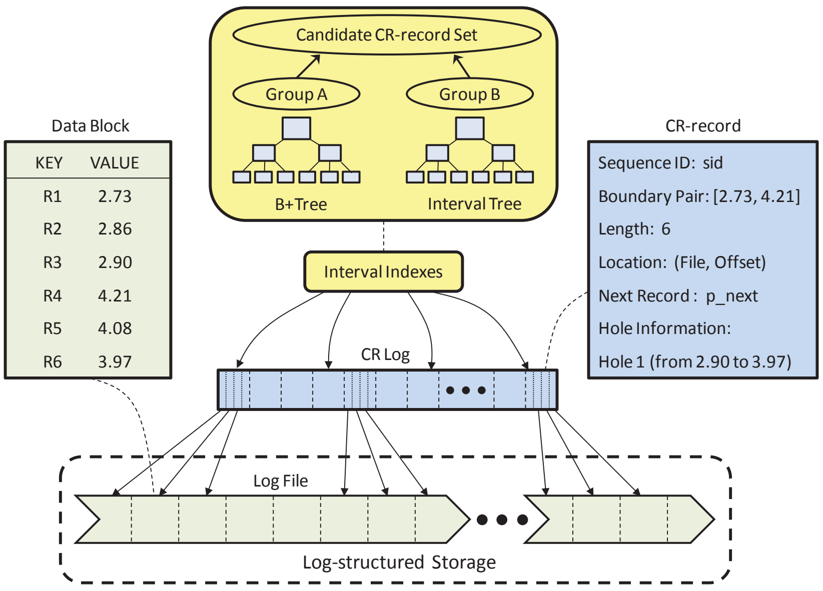


Figure 4.4: The `CR-index` structure.

Figure 4.4 shows the `CR-index` structure for indexing a single attribute. The lowest level is the abstraction of data blocks in the log file. In the middle level, we generate a record, called a *CR-record*, containing brief description for each block, which we use to prune blocks. CR-records are appended to the *CR-log*. The CR-log is much smaller than original data file and may fit in main memory in most cases. The upper level is optional and provides interval indexes to improve the retrieval in disk-based CR-log.

Determining Data Block Size

Data blocks are disjoint groups of successive records in the file. The abstraction of blocks reduces the number of disk-seeks and utilizes high disk bandwidth. The `CR-index` only captures whether some records in a block might be in a query range, but not the location or identity of such records. Thus, even if only one record satisfies the query range, the entire block will be fetched and scanned. Consequently, block length – the number of records in the block – has important influence on index performance. Intuitively, a larger block length will make the CR-log smaller (and fit it in memory), but raises the cost of fetching and scanning a block. Our analysis in Section 4.4.5 will show that query performance degrades sub-linearly with increasing block length.

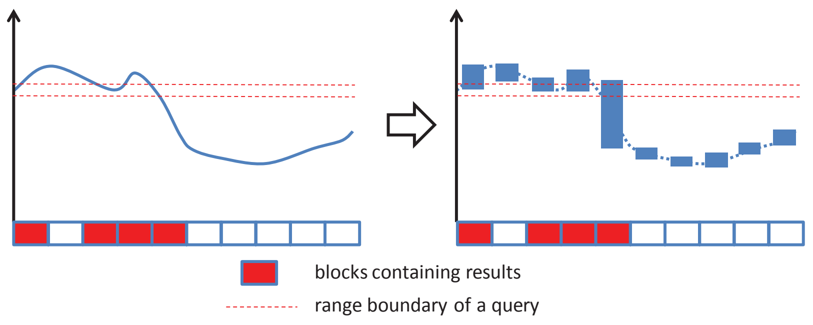


Figure 4.5: Abstraction of continuous data in blocks: original data on the left and block representation on the right.

Describing Data Blocks

In the CR-log, one CR-record describes a block containing possibly hundreds of records. It is challenging to describe the contents of so many records using a small descriptor. Hash-based approaches, such as Bloom Filters [23], provide a compact means for membership testing. However, hash-based approaches do not support range conditions naturally.

Our approach exploits the continuous nature of observational data. Referring back to Section 4.2.1, we expect to find records at a certain maximum spacing between two distinct values. Therefore, a pair of bounding values is reasonable to represent block content. Figure 4.5 shows such abstraction at the block level. In this figure, each block is abstracted as a range of values from minimum to maximum, represented as a *boundary pair* $[\min, \max]$. Although we cannot have every value between the pair, it is highly likely that we will find values in a range that overlaps $[\min, \max]$. Conversely, if the query range is disjoint from $[\min, \max]$, that block will have no valid records. The boundary pair can be computed quickly during insertion. Note that if the data source is not strictly continuous or the query range very small, the boundary pair can cause false positives.

A CR-record contains several fields, as shown in Figure 4.4:

- **Block ID** indicates the sequence order of data block.
- **Boundary pair** abstracts the content of the indexed attribute for records in the block.
- **Block length** is the number of records in the block.
- **File position** indicates the offset of the block location.

- **Hole information** is maintained for discontinuity optimization. The details are discussed in Section 4.4.2.

Indexing Data Blocks

Each boundary pair can be treated as an interval. A range query can therefore be transformed to an intersection-checking problem, i.e., finding all CR-records that overlap a given interval, then fetching and scanning corresponding data blocks. The efficiency of intersection-checking is important. If the CR-log fits in memory, a scan of the entire CR-log may give reasonable performance. However, if it requires disk storage, we need to index it.

Intersection queries are well studied in the literature and diverse index structures have been proposed, such as interval trees and segment trees [37]. However, the query cost using such structures depends heavily on the size of the query ranges: The larger the range is, the more branches in the tree need to be traversed, hurting performance.

To solve this problem, we partition the result set into two disjoint groups, which we retrieve separately but combine before data-block access:

- Group **A**: CR-records that have at least one endpoint inside the query range $[a, b]$.
- Group **B**: CR-records that completely contain the query range.

We retrieve Group **A** using a point-tree structure, such as a B$^+$-tree. For each CR-record, two entries are inserted into the B$^+$-tree, one for each endpoint of its boundary pair. The endpoint serves as a key, while the associated value is the CR-record's reference. For a CR-record in **A**, at least one endpoint can be found by a range query on the B$^+$-tree: find the node containing the query's left endpoint and traverse the successive nodes up to the one with the right endpoint. The number of entries in the B$^+$-tree is equal to twice the number of CR-records in the CR-log.

For retrieving Group **B**, we need a completely different structure. Recall that CR-records in **B** entirely contain the query range. We can simply pick any point d in the query range to represent the whole query. Hence, we have transformed an intersection query into a stabbing query, i.e., the queried range is just a point. (The transformed query might also find CR-records in Group

A.) A stabbing query is fast in interval structures, as it only involves one path in the tree. Though both segment trees and interval trees are suitable for stabbing queries, we prefer latter because of the low space demand, enabling us to cache a large part of the structure in memory.

In summary, a range query will be transformed into two sub-queries: a range query on the B$^+$-tree and a stabbing query on the interval tree. Each sub-query traverses only one tree-path, thus minimizing the number of accessed internal nodes. Sub-query results are combined to remove duplicates. Sfakianakis et al. apply a similar idea to index intervals using a key-value cloud store [80].

4.4.2 Index Optimization

There are several critical issues when using `CR-index` in real applications.

- How to make interval indexes small to cache them.
- How to handle occasional discontinuities in the data.

This section presents optimization mechanisms to handle these issues while preserving index performance.

Index with Delta Intervals

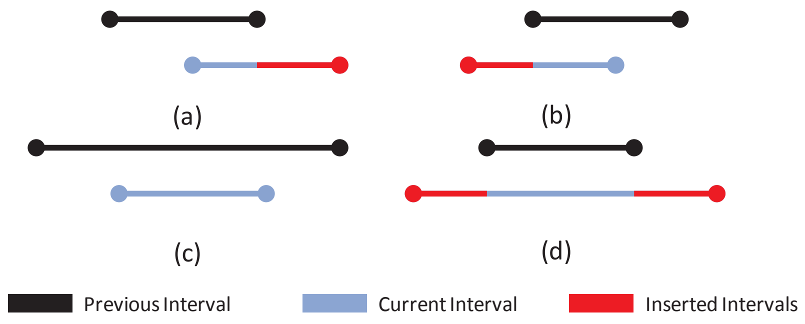


Figure 4.6: Cases of Delta Intervals.

The value of an observational record is expected to be close to that of the previous one. Therefore, boundary intervals of consecutive blocks might well overlap. If a query range intersects a block, there is a high chance that it will intersect following blocks. (We verify this statement using real-world datasets in Section 4.5.) This observation suggests we need not insert the entire interval of each CR-record into the interval indexes. We can instead index only non-overlapping parts of CR-records' intervals.

We define the *delta interval* of a block to be its non-overlapping part with previous one's interval. Only delta intervals are inserted in interval indexes. Figure 4.6 shows four cases of delta intervals, indicated using red segments. The use of delta intervals can significantly reduce the space consumption of interval indexes (B$^+$-tree and interval tree). Instead of inserting two endpoints, only the uncovered endpoints are inserted in the B$^+$-tree. In particular, in Case (c) no entries are needed. For the interval tree, the lengths of inserted intervals are reduced. Smaller intervals will be pushed closer to the leaf nodes, thereby reducing the size of upper-level nodes cached in memory. Although in case (d) two intervals are inserted, the total length becomes smaller.

Algorithm 1: resolveDelta(List entries, Integer K, Range range)

```
Input: a list of entries $entries$, a threshold integer $K$, a query range
       $range$
Output: a list of qualified entries $result$
1  Set $result$;
2  Scanner $CRlog$;
3  for each entry e from entries do
4      Int $counter = 0$;
5      $CRlog$.seek($e$.position);
6      while $counter < K$ do
7          $counter$++;
8          Record $next = CRlog$.next();
9          if $next$ overlaps $range$ then
10             $result$.add($next$);
11             $counter = 0$;
12 return result;
```

To further reduce index size, we extend the delta interval to length k: the portion of the interval not covered by the previous k blocks. This reduction on index size comes at the cost of accessing at most k additional CR-records after a qualifying CR-record. The CR-log is organized sequentially on disk, therefore accessing additional records is fast. Algorithm 1 shows how to resolve the complete CR-record set for a query, using the length-k delta intervals. The list *entries* identify all CR-records found in interval indexes. For each such entry, we locate it in the CR-log and set up a counter (lines 3-5). We keep reading the CR-records until the counter reaches the threshold (lines 6-8), reset the counter if we find a qualifying record (lines 9-11).

Hole Skipper

`CR-indexes` exploit data continuity. There are reasons that the continuity assumption might be violated:

Data loss because of sensor failures or network breakdowns, giving a jump for the missing period.

Abnormal values arising from a natural or man-made disturbance in the environment, such as sensor fouling or a passing vessel.

Multiple data sources in a single file. Figure 4.7 shows a multi-sensor data source[1], the temperatures at different locations in a beehive. As can be seen, there are gaps or "holes" between boundary pairs.

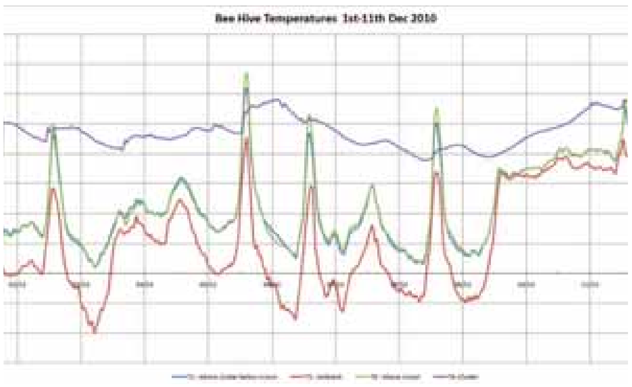


Figure 4.7: Multiple temperatures in a beehive.

Due to such issues, a boundary pair might not accurately describe block's content: sub-ranges with no data may exist. Any query on those sub-ranges will fetch false-positive blocks.

A *hole* is a sub-range that contains no actual values. We are concerned with holes whose *width*s are larger than query ranges. Depending on the cause of the hole, there might be similar holes in adjacent blocks. Thus, we allow a hole to have a *length*, measured in blocks. (Note that if we extend the length of a hole, its width may shrink.) Figure 4.8 shows three holes over a sequence of 11 data blocks. If a query falls in a hole, we can skip blocks for the length of the hole. *Hole skipper* (HS) is a mechanism that tracks a number of holes inside

[1]http://openenergymonitor.org/emon/buildingblocks/sd-card-logging

each CR-record. To limit the space for hole information, HS only keeps the k largest holes in each CR-record. The *size* of a hole is defined as $width \cdot length$. The larger the hole size is, the higher probability of skipping blocks it will have.

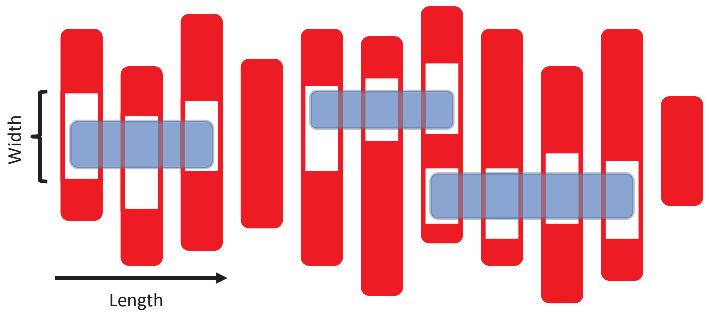


Figure 4.8: Holes in continuous ranges.

We concern the cost of finding largest holes. While there is little overhead in creating the boundary pair when initially writing a block, detecting holes incurs more cost. Therefore, HS applies an adaptive strategy to detect holes during query processing and keep them for future queries. Scanning a false-positive block means a new hole is detected. This hole is a candidate added to the corresponding CR-record.

Algorithm 2 shows the details of hole detection. Data blocks are scanned in order (lines 4-6). If a false positive block is detected (line 7), we extend the length of current hole (lines 8-13).An extension could make the hole smaller because the width might decrease[2]. Once a hole is completed and in top-k, it will be attached into the first CR-record that contains it (lines 8-10, 14-16).

The major advantage of applying the adaptive strategy is that it does not affect throughput in the write phase. In addition, the maintained holes only involve queried data. It avoids capturing holes that are not of interest to the users.

4.4.3 Dealing with Disordered Records

We expect that arriving records are ordered on timestamp, but disordered records can arise due to the network delays. We have to append such records as they arrive. It could pose problems with respect to data-continuity assumption.

Our approach tolerates a certain amount of disorder. The `CR-index` does not care about orders inside a block. On the other hand, disorder between

[2]We actually find the largest empty interval around the query range in each scanned block.

Algorithm 2: detectHole(List CRrecords, Range range)

```
Input: a list of entries $CRentries$, a query range $range$
Output: a list of qualified entries $result$
1  Hole $currentHole$;
2  CRrecord $firstR$;
3  Scanner $logFile$;
4  for each record r in $CRrecords$ do
5      $logFile$.seek($r$.position);
6      $logFile$.readNextBlock();
7      if no results inside $range$ then
8          if $r$ not next to $currentHole$ then
9              $firstR$.addHole($currentHole$);
10             $currentHole$ = new Hole();
11             $firstR = r$;
12         else
13             $currentHole$.extendLength();
14     else
15         $firstR$.addHole($currentHole$);
16         $currentHole$.clear();
17 return result;
```

blocks can extend the scope of boundary pairs, if the delayed record has a value beyond the range of the block. This will be managed by HS.

For time-range query, the existence of disorder extends the number of blocks to cover that range. We maintain a *checkpoint list* to help determine which parts of the file are involved. The system periodically adds time points to the list. For each checkpoint, it maintains: (1) the smallest block id that contains records later than that time; (2) the largest block id that contains records earlier than that time. The CR-records will be filtered by the block id range before actually fetching the corresponding blocks. The checkpoint list is a memory-based structure, therefore it can be easily updated when a delayed record arrives.

4.4.4 Evaluating Range Queries

Consider executing a query with conditions on both time and value. The value condition is used at both the interval-index level and the CR-log level, while the time condition is used at the CR-log via the checkpoint list. In addition, hole information will both be consulted and updated. The following are the main steps in evaluating a query:

1. Access the interval indexes to get CR-records ids: Group **A** from B$^+$-trees and Group **B** from interval trees.
2. Locate each identified record in the CR-log. Scan the log for additional CR-records if using delta intervals.
3. Filter CR-records using checkpoint list and hole information.
4. Fetch and scan the data blocks for remaining CR-records. Extract and return all qualifying results.
5. For any detected false-positive blocks, track the holes and update the hole information in CR-records.

4.4.5 Analysis of Index Behavior

The effectiveness of the `CR-index` depends on the data-continuity. To analyze index behavior, we first introduce metrics to quantify the continuity of a dataset. With these metrics, we can derive mathematical estimates of index performance. Finally, the tradeoff between index-maintenance cost and query cost is discussed.

Continuity of Observational Data

Consider an observational dataset D with N_D records, arranged in temporal order. For each record r_i ($1 \leq i \leq N_D$), v_i denotes the value of the indexed attribute. We define the *continuity distance* (dis) for D as:

$$dis(D) = \frac{1}{N_D - 1} \sum_{i=2}^{N_D} dis_i$$

$$dis_i = |v_i - v_{i-1}|$$

The dis_i represent the numerical distance between two adjacent records.

The more continuous D is, the lower the $dis(D)$ will be. To calibrate the continuity distance, the expected range size of queries should also be considered. For example, suppose the $dis(D)$ is 0.1. D has good continuity when the query range is $[23.2, 25.8]$, but not if the range is $[23.256, 23.259]$. Therefore, for any query Q with range $[a, b]$, we define the *degree of continuity* (doc) as:

$$doc(Q, D) = \frac{|b - a|}{dis(D)}$$

The larger the $doc(Q, D)$ is, the better continuity quality the dataset possesses for the given query.

Query Cost

Now we analyze the cost of executing a query Q with range $[a, b]$ and degree of continuity $doc(Q, D)$. Suppose the result set is R. Since the data values are continuous, we can expect R to be consist of sequences of contiguously stored records, whose values in the range $[a, b]$.

Using doc we can estimate the number of sequences in R. We start by randomly choosing a record in R and estimating the number of records in the sequence it belongs to. Suppose the chosen record has value $x \in [a, b]$. The shortest path for a continuous source to enter the query range, reach this value and then leave the range is when it both enters and exits from the nearer side, e.g., from point a if $|x - a| \leq |x - b|$. Therefore, the shortest path for reaching x in the range is:

$$path(x) = 2 \cdot \min\left(|x - a|, |x - b|\right)$$

We obtain the expected distance of the path by considering all values for x:

$$path = \int_a^b path(x)\mathrm{d}x = \frac{|b - a|}{2}$$

Thus we can expect $path/dis(D) = doc(Q, D)/2$ points in the same sequence as x. Therefore, the number of disjoint sequences N_{seq} in the result set can be estimated as:

$$N_{seq} = \lceil \frac{2|R|}{doc(Q, D)} \rceil$$

Note that this estimate is likely to be on the high side as paths through x are always larger than the minimum. Let the *block length* (L_{block}) be the number of records in each data block. For each sequence, at most $\lceil \frac{doc(Q,D)}{2L_{block}} + 1 \rceil$ blocks are needed to cover it. The total number of records accessed is bounded by:

$$B \leq N_{seq} \cdot L_{block} \cdot \lceil \frac{doc(Q, D)}{2L_{block}} + 1 \rceil$$

Only N_{seq} disk seeks are executed while accessing these records. The overall disk cost of executing query Q is:

$$\text{COST}_Q = T_{seek} \cdot N_{seq} + T_{trans} \cdot B$$

Here T_{seek} is the time of executing a disk seek and T_{trans} is the time of transferring a single record.

Unlike other indexing methods whose costs are defined in terms of the number of I/Os and how close it is to the optimal I/O $O(\log n + \#results)$, our index separate disk-seek cost and data-transfer cost. The `CR-index` tries to reduce costly seeks for a better utilization of disk bandwidth.

Storage Cost versus Query Performance

The key parameter that tunes the trade-off between `CR-index` storage cost and query performance is the block length L_{block}. The main contribution to the storage cost is the space for the CR-log. The size of CR-log varies inversely with L_{block}. With the query cost model in Section 4.4.5, disk seek cost is not affected much by L_{block}. By increasing L_{block} to $L_{block} + \Delta L$, only $N_{seq} \cdot \Delta L$ additional records are accessed. Thus we can trade a reduction in index size for a marginal increase in query time. Suppose L_{block} is 100 and N_{seq} for a given query is 5. If we increase L_{block} to 200, CR-log will consume half the space, at a cost of possibly reading 500 additional records. The amortized index size for each record can be just $1 - 5$ bytes. Hopefully, tens of MB of space are adequate for handling observational data on the scale of GB.

4.4.6 Multi-Attribute Queries

In previous sections, we present the details of executing single-attribute range queries using `CR-index`es. Here we discuss the feasibility of utilizing `CR-index`es on multi-attribute range queries as might arise in applications.

Multiple Continuous Attributes

We have argued that `CR-index` is lightweight in terms of both time and space cost. The overhead of maintaining secondary indexes on many attributes should be acceptable. As a result, the set of available indexes can facilitate the processing of complicated queries involving multiple attributes. Conceptually, the idea of the `CR-index` is easy to extend to multi-attribute cases. On the other hand, the conventional indexes, such as B$^+$-tree, cannot efficiently handle queries with range conditions on more than one attributes.

In detail, the strategy is to break a multi-attribute query into several single-attribute sub-queries. Each sub-query accesses a `CR-index` instance and the returned entries indicate the scope of sub-query results. It is possible to merge multiple scopes, depending on the OR and AND connectives in query expressions, before we fetch data blocks. For example, we have two blocks from different `CR-indexes`: one involving file offsets from 10 to 30 and the other from 20 to 40. After examining the query, we can directly extract results from [10,40] or [20,30] for OR or AND respectively. The system thus avoid accessing redundant and non-satisfying items.

In order to coordinate indexes and provide better efficiency, global data partitions can be applied for all indexes in the same table, i.e. the block partitions are common among all index instances and using global block ids. Compared to local data partitions in each index, a global partition could significantly reduce the index space and computations. The merge of results in multi-attribute queries could be processed at the level of block id, making merge operations much more efficient.

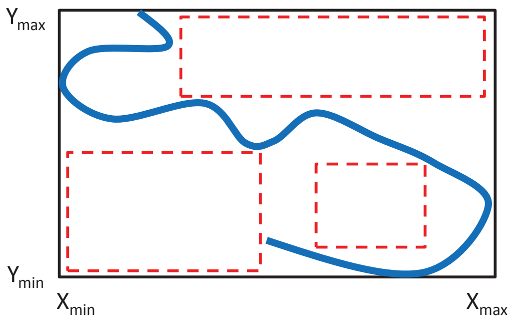


Figure 4.9: Inevitable holes (dashed-rectangles) in 2-dimensional continuous data sources.

We note that there are inherent holes in multi-dimensional data, even when each dimension is ideally continuous. In Figure 4.9, we show two sources that are ideally continuous over time, but where there exist large holes. An advanced method that addresses this issue will be discussed in Chapter 5.

Primary-Key Attributes

There is a second type of multi-attribute query, which includes a constraint on primary-key attributes, e.g., the retrieval of data from a specific sensor for a time period besides a range of salinity. If the number of distinct keys that will be retrieved is limited, an additional boundary-pair can be added for each such key. They further filter CR-records before fetching blocks. In the worst case where there are an excessive number of distinct keys, we still have two choices in execution: (1) get records using the primary index and filter them by value-conditions; (2) fetch data blocks by secondary `CR-index`es and extract the results using key constraints. Since the index-lookup cost of `CR-index` is extremely cheap, we can get the CR-records before actually making the decision. Choice (2) is preferred when the number of returned CR-records is small, which means only a few blocks need to scan. We test such queries in Section 4.5.6.

4.5 Experimental Results

This section presents an experimental study on indexing and querying observational data with the `CR-index`. Our objective is to demonstrate the feasibility of using this lightweight index to provide good query performance, compared to that of conventional record-level indexes. In addition, we will demonstrate its high write throughput, which makes it an excellent choice for write-intensive applications.

We compare the `CR-index` with two conventional index structures: B^+-trees and LSM-trees. We use open-source implementations for these alternatives, namely JDBM3 for B^+-tree [6] and LevelDB [7] for LSM-tree. In order to show the effect of design choices, we also compare variants of the `CR-index`. The variants consider the choices of CR-log storage types (disk-based or memory-based) and access types (interval indexes or brute-force scan).

4.5.1 Data Sets

We use two real sensor datasets for our test, one from scientific observations, the other from an instrumented sports game. The first dataset is strongly continuous, while the second one has numerous holes.

CMOP Coastal Margin Data

This dataset contains coastal margin data collected from the CMOP [1] SATURN Observing System. The data were collected between April 2011 and August 2012 from an observation station in SATURN. It contains diverse physical variables reflecting ocean and river status, including salinity, temperature and oxygen saturation. We transform the raw data files into records, each of which contains values collected at the same time.

Real-time Soccer Game Data

The second dataset is from the DEBS 2013 Grand Challenge [2]. This high-resolution data was collected from sensors embedded in balls during a soccer game. Each sensor produces records at 2000Hz. Each record contains sensor id, timestamp, position, speed as well as velocity and acceleration in 3 dimensions. The data is the combination of the readings from four balls, used alternately during the game. The original stream also contained lower-frequency readings for the players, which we removed in our test.

4.5.2 Experimental Setup

All experiments were performed on an in-house cluster, where each machine has a quad-core processor, 8 GB physical memory and 500GB disk capacity. All indexes (including `CR-index`, *B^+-tree* and *LSM-tree*) are implemented in JAVA and embedded into the `LogBase`.

We use default settings for both B^+-tree and LSM-tree as given in the open source code. In the `CR-index` configuration: the data-block length is 64 records; the CR-log is on disk and indexed by in-memory interval indexes; the delta-interval length is 1; each CR-record holds up to 5 holes.

The secondary index is built on single attributes in both datasets: *salinity* in the ocean data (CMOP) and *speed* in the soccer data (GAME). The CMOP data has better continuity, since the salinity of water changes slowly while the speed of balls can change suddenly. The query set contains queries that retrieve records whose indexed attribute lies in specified value ranges, with no restriction on time.

In each test, the client uploads a number of records into the system: 13 million for CMOP and 25 million for GAME. Records are managed by a single tablet server. The length of raw records are around 200 bytes and 100 bytes, respectively. After each fifth of the records is inserted, 10 queries are issued from the query set. The average result selectivity of queries is 8.4% for CMOP and 6.3% for GAME.

4.5.3 Write Performance

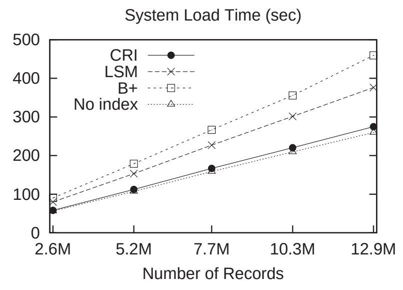
This subsection focuses on the data-insertion performance. We compare different index approaches on both time and space consumption.

System-Load Time

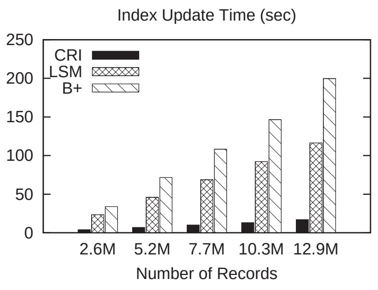
Figure 4.10 illustrates the write time in loading data, excluding the time of executing queries. As can be seen, the `CR-index` (CRI) is extremely lightweight and raises system time only slightly, by no more than 8%. This low overhead is suitable for write-intensive scenarios and allows maintaining many secondary indexes on a table. In contrast, both LSM-tree (LSM) and B^+-tree (B+) cause significant performance reductions. The write-optimized LSM-tree has 45-77% extra system cost, while the read-optimized B^+-tree's extra cost is 78-124%. Since the B^+-tree is update-in-place, its split operations bring random I/Os and thereby make the maintenance not scalable. Note that the GAME data was collected from an 1-hour game and our system is capable of processing the data in real-time.

Index-Update Time

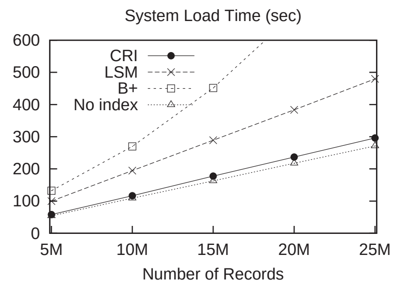
Figure 4.11 presents the index-only cost. We observe that the index update cost of the `CR-index` is about an order of magnitude lower than conventional index structures. The total cost is only 15% of LSM-tree and 9% of B^+-tree. The reduction in index update time comes from the boundary-pair abstraction. Each block generates only one index entry, much less than in other approaches.



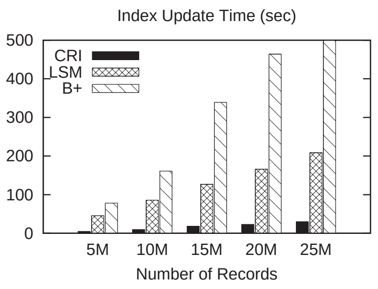
(a) CMOP



(a) CMOP



(b) GAME



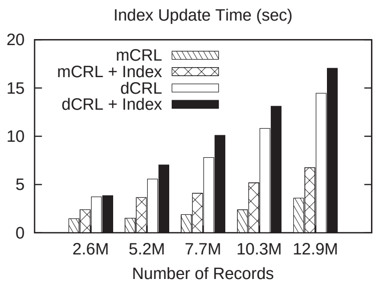
(b) GAME

Figure 4.10: Overall system load time. Figure 4.11: Index maintenance cost.

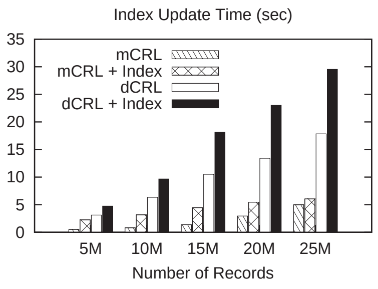
Figure 4.12 provides the detailed index-update overhead of `CR-index` variants. The most lightweight variant uses a memory-based CR-log without interval indexes (mCRL), which incurs no I/O. The interval indexes can still be constructed on a memory-based CR-log (mCRL + index), but updating indexes add cost. The disk-based CR-log (dCRL) has minimal memory consumption, at the cost of sequential I/Os. Although the maintenance cost is much higher than for in-memory variants, it is still an order of magnitude smaller than data-load time. The most versatile variant is the disk-based CR-log with memory-based interval indexes (dCRL + index), which is the default variant in other tests. The interval indexes raise the index cost by up 20-70% than disk-based approach but consume much less memory than memory-only variants.

Index-Space Consumption

Figure 4.13 summarizes the disk-space consumption of different indexes. We only sum up the disk space, ignoring any memory usage. In the figure, we can see that the size of the `CR-index` is only 10-12% of the LSM-tree and 4-6% of the B$^+$-tree. We expect that the B$^+$-tree uses more space than the LSM-tree, since its disk pages are often partially full. With default data-block length,



(a) CMOP



(b) GAME

Figure 4.12: CR-index variants maintenance cost.

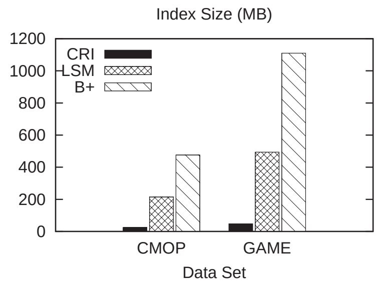


Figure 4.13: Index space consumption.

the number of entries in the CR-index is only 1/64 of that for the record-level indexes. However, since each entry (CR-record) keeps several fields, such as hole information, the entry size is larger.

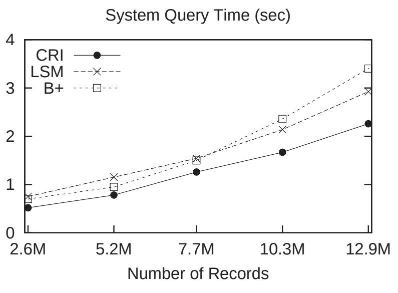
4.5.4 Query Performance

This subsection focuses on the response time of range queries. We consider both overall response time and sub-phase execution time.

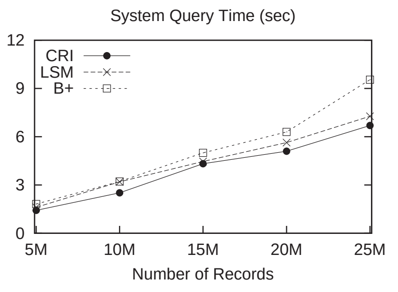
Query-Response Time

Figure 4.14 shows the overall query response time with different approaches. As can be seen, the response time of the CR-index is comparable to that for the LSM-tree and B$^+$-tree. It performs better on CMOP data, since the ocean's salinity provides stronger continuity than the mixture of four balls' speeds. The results from both datasets show that the CR-index can replace conventional indexes on observational data while preserving similar query performance.

Since all these are secondary indexes, they all employ two steps to process a query: the *index-lookup* phase accesses the index to get record references (in

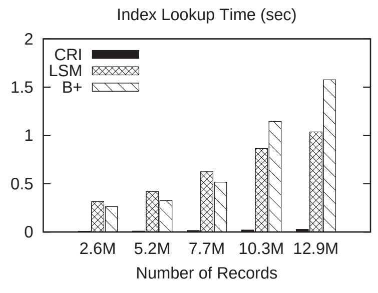


(a) CMOP

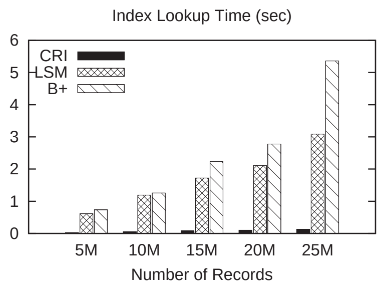


(b) GAME

Figure 4.14: Overall system query response time.



(a) CMOP



(b) GAME

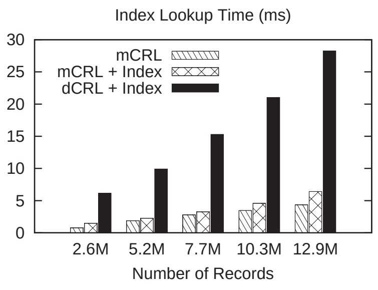
Figure 4.15: Index lookup phase cost.

B$^+$-tree and LSM-tree) or block references (in `CR-index`); the *data-access* phase reads records or blocks from data files. For both LSM-trees and B$^+$-trees, the lookup cost is significant. After a record is identified, they access it efficiently using accurate positional information. On the other hand, the lookup cost on small-sized `CR-index` is negligible. Most of the cost is incurred in fetching and scanning data blocks.

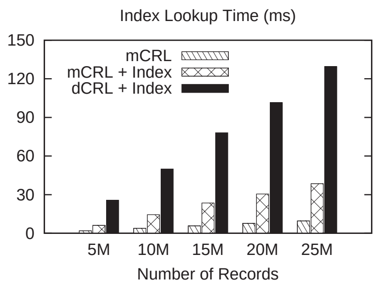
Index-Lookup Cost versus Data-Access Cost

Figure 4.15 examines the index-lookup cost. As can be observed, the `CR-index` spends much less time than other approaches in this phase. The total cost is only 3-7% of that of the LSM-tree and 4-9% of the B$^+$-tree. These values are not surprising, since the number of entries in the `CR-index` is only 1/64 that of the other two. Although the efficient lookup comes at the cost of increasing data-access time, the overall cost of these two phases is still low.

Figure 4.16 shows the lookup cost of different `CR-index` variants. Since scanning a disk-based CR-log takes about five times longer than using interval indexes, we discard that variant from the figure. Memory-based variants

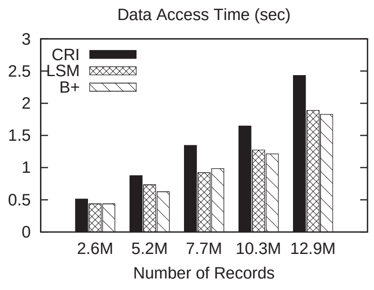


(a) CMOP

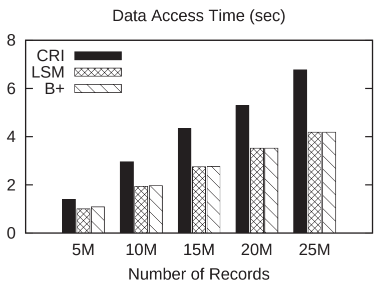


(b) GAME

Figure 4.16: `CR-index` variants index lookup cost.



(a) CMOP



(b) GAME

Figure 4.17: Data access phase cost.

(mCRL and mCRL+Index) have excellent lookup performance. Note that using the interval indexes with a memory-based CR-log actually incurs a performance penalty. However, for a disk-based CR-log (dCRL+Index), the interval indexes are necessary to reduce I/Os.

Figure 4.17 shows the data-access cost. The LSM-tree and B$^+$-tree have the identical set of record references. The time of accessing records in data files is therefore similar. However, for the `CR-index`, the data-access cost is higher, because data blocks are fetched. The block length used in our test is 64, but the data-access time is not 64 times longer. The block-scan only increases the time by 26-34%, as an accessed block always contains many results and most blocks are read as part of sequences. Table 4.1 shows the statistics for accessing blocks in different queries. In GAME data, since the records are from four balls, the results are diluted by noises from other readings. Hence, the number of blocks in a sequence is larger while the number of results in a block decreases. N_{seq} is the estimated number of sequences, using the analysis in Section 4.4.5, which is pessimistic. In real datasets, the number of seeks performed is much less than the theoretical bound.

Table 4.1: Result Sequences in Datasets

Query	CMOP1	CMOP2	GAME1	GAME2
# Results	503K	1217K	1075K	1595K
# Blocks	13815	29524	60754	95142
# Sequences	2991	6176	2249	5821
N_{seq}	28367	62380	6096	18087
Res/Blk	36.4	41.2	17.7	16.8
Blk/Seq	4.62	4.78	27.01	16.34

4.5.5 Influencing Factors

This subsection covers several factors that influence index performance and allow tuning the trade-off between write and query cost. The data size in following tests used the full-size configuration (12.9M for CMOP and 25M for GAME).

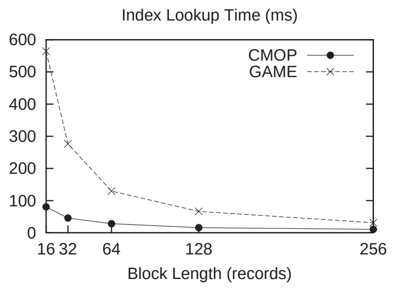
Block Length

Block length dominates both the number of generated CR-records and the block-scan cost. Figure 4.18(a) shows the index-lookup cost with different block lengths. The lookup time appears proportional to $O(n \log n)$, where n refers to the number of index entries. This pattern is expected due to the retrieval cost on tree-structured interval indexes.

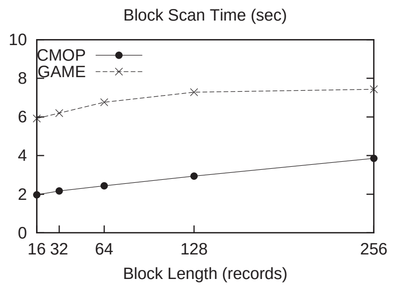
The data-access time intuitively rises with increasing block length. Figure 4.18(b) presents this trend. From this figure, we observe that scan cost increases linearly but slowly with block length. This phenomenon coincides with the mathematical analysis in Section 4.4.5 and supports our point that index size and update cost can be reduced significantly with only a moderate effect on query performance.

Effect of Hole Skipper

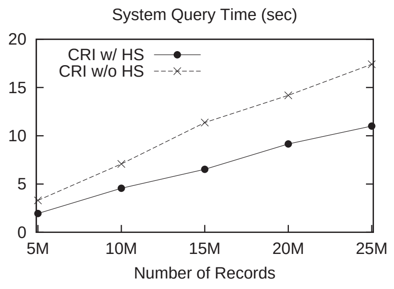
Potential discontinuities occur for many reasons. In the GAME dataset, the gaps in ball speed are produced by nature of the game: only one ball is in play alternately. Therefore, there are inherent gaps between the speed of the active ball and that of the other balls. Figure 4.18(c) shows the improvement in query response time that Hole Skipper provides for GAME data. As can be observed, it improves the performance by about 40%. HS only helps slightly for CMOP



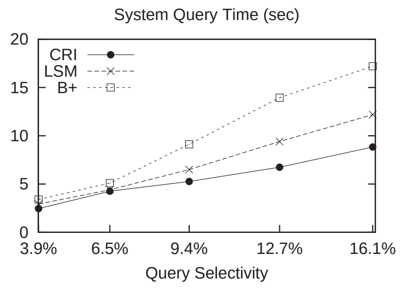
(a) Index lookup time relative to block length.



(b) Block scan time relative to block length.



(c) Query time improved by Hole Skipper.



(d) Query time relative to selectivity.

Figure 4.18: `CR-index` performance affected by different factors.

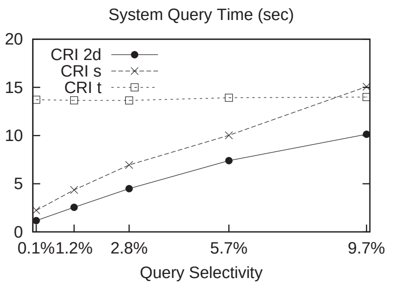
data because of its good continuity, and we omit the comparison here.

Query Selectivity

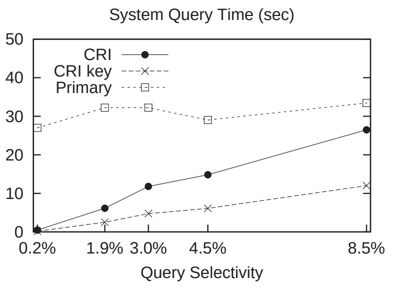
Figure 4.18(d) shows the query response time with different query selectivities on the CMOP dataset. As can be seen, when the selectivity is low, e.g. 3.9%, all indexes performs well. However, as the range increases, the `CR-index` scales well. In B$^+$-trees, split pages might not be physically contiguous. Therefore, for large-range queries, randomly located pages are accessed, hurting performance. In contrast, the `CR-index` only execute sequential I/Os in both index-lookup and data-access phases, accessing the CR-log and data files. Therefore, the `CR-index` is more scalable than record-level indexes, both in terms of dataset and result size.

4.5.6 Multi-Attribute Queries

This subsection verifies the feasibility of extending the `CR-index` to handle complex multi-attribute queries. The data size is still the full-size configuration.



(a) 2D



(b) 1D+Key

Figure 4.19: Multi-dimensional queries response time: (a) 2-dimensional range; (b) 1-dimensional range for a specific key.

Queries on Multiple Observational Attributes

We first consider a 2-attribute range query: retrieval of CMOP records whose *salinity* and *temperature* are in specific ranges. Having an index on only one attribute results in further filtering of returned records. The query selectivity is varied by changing the salinity-range while the temperature-range is fixed. As can been seen in Figure 4.19(a), the response time of indexing *salinity*(CRI s) is influenced severely by query selectivity as the number of candidate blocks increase, compared to indexing *temperature*(CRI t). When both `CR-index`es are available, the pre-filtering of CR-records prevents fetching most of the non-satisfying data blocks and the consequent improvement is significant (CRI 2d). The proposed technique is extensible for more than 2 attributes.

Queries with Equality on Primary Key

Second, we consider queries with select-conditions on both key and attributes: retrieval of GAME records whose *velocity* is in a specified range and from a specified sensor. The query selectivity is varied by changing the range while fixing the key. `LogBase` provides the key-based primary index, which we can use to fetch records and then filter by range-conditions. As can be seen in Figure 4.19(b), this plan (Primary) is not affected by query selectivity. On the other hand, the `CR-index` (CRI) on *velocity* is sensitive to query-range and could be outperformed by the primary index at some point of selectivity. This point could be considered as the watershed for the choice of query plan. Following our discussion in Section 4.4.6, with the improvement where the key has its own boundary-pair (CRI key), the overall response time could be much lower.

4.6 Summary

Log-structured storage is a natural choice for storing observational data that arrives as streams. We designed a novel lightweight index structure called the `CR-index` which is small enough to reside in main memory and is fast to construct. It avoids indexing each item, as in conventional indexes, and therefore achieves high write throughput in write-heavy applications. The index supports fast location of potential results, followed by a data-scan. The index exploits several key properties of observational data, most importantly, continuity. The experimental analysis verifies the feasibility of the `CR-index` and confirms that it can provide good query performance compared to existing indexing strategies, while achieving high write throughput. For other application areas where data is not strictly continuous but values are correlated between successive records, for example stock prices, our index might also be effective.

CHAPTER 5

MULTI-DIMENSIONAL OBSERVATIONAL DATA IN LOG STORE

Tremendous amounts of data are being generated by sensing devices each day, which include large quantities of multi-dimensional measurements. These data are expected to be immediately available for real-time analytics as they are streamed into storage. Such scenarios pose challenges to state-of-the-art indexing methods, as they must not only support efficient queries but also frequent updates. In this chapter, we propose a novel indexing method that ingests multi-dimensional observational data in real time. This method primarily guarantees extremely high throughput for data ingestion, while it can be continuously refined in the background to improve query efficiency. Instead of representing collections of points using Minimal Bounding Boxes as in conventional indexes, we model sets of successive points as line segments in hyperspaces, by exploiting the intrinsic value continuity in observational data. Such a representation reduces the number of index entries and drastically reduces "over-coverage" by entries. Our experimental results show that our proposed approach handles real-world workloads gracefully, providing both low-overhead indexing and excellent query efficiency.

5.1 Introduction

Rapid advances in sensing technologies and devices are creating a new norm in digitizing our physical world and daily life. The types of entities whose state can be continuously captured are increasing in tandem, from microscopic molecules to macroscopic celestial bodies. The range of properties that can be sensed from monitored entities is growing as well. As a result, the collected measurements, called *observational data*, are exploding both in volume and velocity. On one hand, sensors' capabilities keep improving in both sampling frequency and resolution. For example, a single sensor can capture the velocity of a moving object in units of $\mu m/s$ at a frequency of 2000Hz. On the other hand, decreasing device prices and increasing power efficiency facilitate the deployment of large sensor networks. They may consist of thousands of sensors, producing simultaneous high-frequency observations. All these trends make observational data management write-intensive.

To make the collected data ready for querying as soon as they are ingested into storage, indexing structures must be efficient for frequent updates. Various indexing methods have been proposed to address this problem. Examples include bulk-insertion techniques [28, 30] that update the indexes in a batch manner, which lowers the per-item cost, and log-structured-merge trees [14, 71, 78] that incur only sequential I/Os for updates. In the previous chapter, we proposed an index that exploits the intrinsic *value-continuity* of observations. Compared with other indexes where individual records are indexed, this method assigns one index entry for a collection of records (represented as a bounding-value pair) to reduce index-construction cost. When a query arrives, it relies on partial scans to access record collections. However, that method only addresses data in single dimension.

In this chapter, we study the indexing problem for multi-dimensional observations. In practice, an observational data flow is usually a continuous collection of observations with multiple attributes (i.e., dimensions). For example, observations from underwater sensors may contain water temperature, salinity, oxygen saturation and pH. Though all these data could be perfectly value-continuous, it is still challenging to exploit this feature in practice. First, data sparseness in multi-dimensional spaces hurts query efficiency. Bounding objects such as minimal bounding rectangles/boxes/spheres (MBR/MBB/MBS)

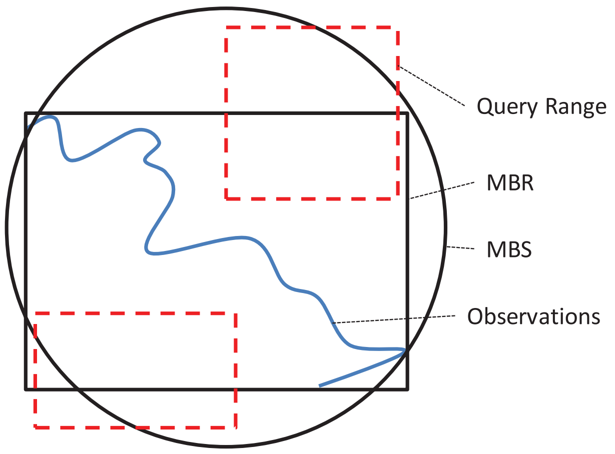


Figure 5.1: False positives from data sparseness.

are widely used in conventional indexes [13, 50, 59] to represent a collection of data items. However, such representations will cause "over-coverage", i.e., portions of indexed spaces that contain no actual points, as shown in Figure 5.1, where queries overlap with bounding objects, but not with observations. These structures force us to access false-positive entries. This issue becomes even more severe as the dimensionality increases. Second, the write-intensive aspect limits resources and possibilities to derive bounding objects with the least over-coverage. There is prior work [58] that investigates flexible bounding objects to reduce over-coverage, but it incurs high construction cost and hence is not affordable in our case. The derivation of index entries should be extremely fast so that it does not affect the throughput for updates.

To address challenges above, we propose a novel indexing method called SICC (Segment-oriented Indexing for Continuously-Changing data), which exploits value-continuity on observations to support fast and adaptive indexes for real-time workloads. The index is extremely lightweight for new data ingestion, and at the same time ensures query efficiency. Its graceful performance is derived from the following three important considerations. First, we note that while data are scattered all over the space in a global view, points collected during a short time period nonetheless can be estimated as a segment in the hyperspace. This inspires us to represent points concisely with a *bounding segment* that minimizes over-coverage. Second, to ensure real-time access, the index must be constructed as data arrive. The cost for deriving and indexing bounding segments is expected to be small and constant. Third, the initial construction targets massive data ingestion, hence might lead to unsatisfactory performance on some queries. Index-structures should be able to improve over time in the background.

The contributions of this chapter include:

- A novel bounding object called a *bounding segment* that represents points as a hyperline-segment, exploiting the value-continuity in observational data. Related operations, such as deriving segments and calculating volumes are provided as well.

- A framework that constructs segment-oriented indexes for continuously arriving observations. Under this framework, online segmentation algorithms derive bounding segments effectively and efficiently.

- An R-tree variant for indexing bounding segments with low overhead, while ensuring query efficiency. The structure is adaptive: it can be continuously improved based on query execution statistics.

- An extensive experimental evaluation on two real-world datasets, comparing with baseline approaches. The results confirm that our approach significantly reduces insertion overhead and at the same time provides excellent query efficiency.

The rest of this chapter is organized as follows. In Section 5.2, we state the targeted problem and key concepts in our design. Section 5.3 presents the overall framework. The details of bounding-segment representation and index construction are discussed in Sections 5.4 and 5.5, respectively. We evaluate performance in Section 5.6. Summaries are given in Section 5.7;

5.2 Preliminaries

This section characterizes observational data and provides a description of the problem we address. We also present key concepts in our design and explain how they exploit characteristics of observational data.

5.2.1 Observational Data

Sensing devices are common data sources for analytic applications, especially scientific tasks. For example, coastal-margin observation [1] deploys underwater sensors at different sites and depths to gather data, such as water temperature, salinity and oxygen saturation. We call data collected in such a manner *observational data*, as they are in fact observations of an entity (or environment) at different moments.

In this chapter, we rely on three characteristics of observational data, as in the previous chapter, that facilitate indexing and querying: **1) Append-only:** Observations are rarely modified after entering the storage, each having its own *observation time*. **2) Value-continuity:** Observations from the same sensor inherently tend to have similar readings during a certain period. **3) Sequence analysis:** It is common to analyze series of consecutive observations rather than individual points. These traits permit us to reduce index information without compromising query efficiency.

5.2.2 Problem Description

We focus on an online observational data flow, with an unbounded stream of arriving records. Assume that each record contains d property dimensions as floating-point values, as well as an observation time. We can represent each individual record as a point[1] in a d-dimensional space R^d (or R^{d+1} with the time dimension).

The major concern is that, upon arrival, each point should be stored and indexed as quickly as possible. Therefore, the system can provide superior write-throughput for rapidly generated observations. At the same time, a query request should be able to acquire up-to-date results. Due to the nature of observational data and scientific analysis, it is uncommon to perform *exact point queries* on high-precision values. Hence, we focus on *range queries*, which find all points that lie within a d-dimensional query range r (or with $d' < d$ dimensions specified, with other dimensions as infinite ranges). We address the retrieval of complete query answers, where no approximate or lossy results are allowed.

[1]An observation (application level), a record (storage level) and a point (logical level) are used interchangeably in this paper.

This task is indeed an indexing problem, but in a specific context. Consequently, we have to address various issues with respect to the particularities and opportunities present. For sequence analysis, qualified records are expected to be returned in observation-time order, since an additional sorting phase is expensive for results with large cardinality. To provide real-time access, we also need to support incremental queries, in which results are periodically produced over just new data.

5.2.3 Basic Design of SICC Indexes

Low Overhead. The traits and challenges of observational data discussed above drive our index design. Most important is low maintenance overhead. For write-intensive applications, it is prohibitive to take up too many system resources to index constantly arriving records at the expense of input throughput or query speed. We prefer lightweight methods that handles high insertion rates and concurrent query retrievals, as indexes could be refined later when the system has available resources.

Log-Structured Storage and Sequential I/Os. Sequence analysis is common for observational data. For this purpose, we use a log-structured storage system as our data storage. Log-structured storage is an ideal platform for us to store and query data with sequential writes and reads. First, it has higher write-throughput compared to update-in-place systems. As records are immediately appended into log files, separate WALs are eliminated, and this aspect fundamentally saves a large number of I/Os for data ingestion. Second, sequential scans return records in insertion order (which correlates with observation time). Furthermore, scans are highly efficient, as they avoid disk seeks and yield high bandwidth. Given a single disk-head seek ($\sim 5ms$), accessing one record ($\sim 10\mu s$) or hundreds of physically contiguous records have comparable costs. Therefore, the cost of a data block scan should be comparable to a single random record access.

Intrinsic Clustering versus Induced Clustering. Traditional index methods organize records with similar values into the same physical page to save query I/Os. This arrangement can be called *induced data clustering*. As new records arrive, they are always stored nearby similar records. Thus the physical index organization keeps changing, incurring extra overhead. In contrast, there is *intrinsic data clustering* in observational data, which provides a similar effect to induced clustering, but with little overhead. By simply appending newly arrived records into contiguous disk pages, potential results of a range query tend to be grouped together. Although not all results of a query will be adjacent, they are likely clustered into consecutive sequences on the disk. This phenomenon becomes more attractive in multi-dimensional spaces. When storing a multi-dimensional point on disk, the "closest" neighbors on each dimension cannot be all physically nearby [57]. Hence there is no perfect induced clustering for all dimensions, while intrinsic clustering still provides reasonable effectiveness.

5.3 Index Framework

In this section, we introduce our `SICC` framework for indexing observational data. The architecture is shown in Figure 5.2. Arriving data are stored sequentially in log-structured storage before being indexed. On top of the storage, we maintain two tiers of index structures.

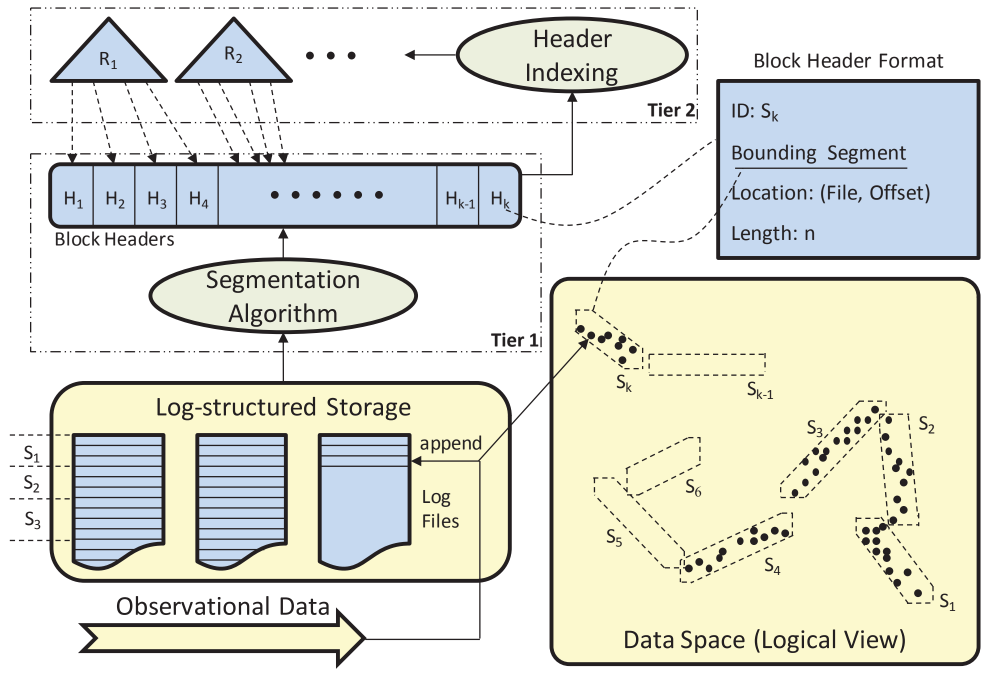


Figure 5.2: `SICC` index framework.

In the first tier, all records are divided into logical blocks, via a *segmentation* *algorithm*. A block refers to a number of successive records and is the finest unit for data access. For any query referring to a certain subset of the records in a block, the whole block will be fetched as a batch. For each block, we generate a *block header*, which keeps necessary information for processing queries, i.e., that needed to determine whether and where to access those records. More specifically, the *bounding segment* in the header determines if the block possibly contains query results. Only when the bounding segment intersects the query range will the system fetch that data block. These headers are generated by the segmentation algorithm on the fly. Since a group of records share a single header, index size is reduced.

In the second tier, block headers are organized and indexed, so that we need test only some of the headers to get the complete result of a query. During frequent data insertion, good organization is essential to ensure query efficiency, as the number of blocks can quickly grow large. This tier performs as an indexing structure for bounding segments in the headers. Additionally, the organization must facilitate sequential scans for data blocks, so that disk bandwidth can be fully utilized.

Overall, there are three critical issues that dominate the effectiveness and efficiency of our `SICC` framework:

- Given a block of records, how do we derive a proper bounding segment, so that "over-coverage" in multi-dimensional space is minimized? (Section 5.4)
- During massive data ingestion, how do we generate bounding segments and index them without compromising system performance? (Section 5.5.1)
- Given that the system still has available resources after handling data ingestion and query requests, how can we further refine the index for better query efficiency? (Section 5.5.2)

5.4 Bounding Segment

In this section, we present the structure of *bounding segments*. Algorithms that quickly compute bounding segments, match segments against queries, and calculate segment volumes are also provided.

5.4.1 Continuity among Observations

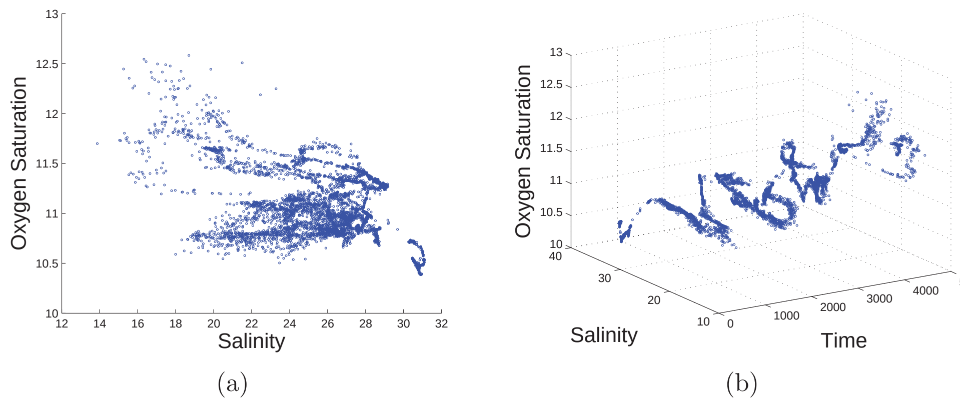


Figure 5.3: Observations from an estuary that contain salinity and oxygen saturation: (a) distribution within a time period; (b) distribution over time.

For a sequence of d-dimensional observations, the corresponding points form a continuous path in the d-space, provided that the values in each dimension are continuous over time. This pattern can be observed in real-world data sources. Figure 5.3 shows such a situation, where the salinity and oxygen saturation status of an estuary are expressed in a $2d$-space. Although the path is obvious in Figure 5.3(b), these two dimensions are not completely correlated, as can be seen in Figure 5.3(a). No global correlation means that it is ineffective to directly apply dimension-reduction techniques [43] to cut down the dimensionality. However, the path (exhibiting "local correlations") can be exploited. For a properly chosen sequence of points, their path can be approximated as a segment of a hyperline in d-space. This opportunity motivates our design for bounding segments.

5.4.2 Bounding Segment Format

Here we present the structure of a bounding segment, and visualize the space bounded by a segment. Assume that we are considering a d-dimensional space. A bounding segment S consists of three d-dimensional values:

- v_b represents the *base* point of the segment,
- v_s represents the *segment* direction and length, and

- v_e represents the *extent* of points on the segment.

Figure 5.4 depicts these values and the bounded space, when $d = 2$. Points v_b and $(v_b + v_s)$ are the two endpoints of the underlying hyperline segment. The extent v_e only contains non-negative scalars. It can be viewed as a function $mbb_e(p)$ that extends any point p to be an MBB centered at p, with bounding range $[p_{(i)} - v_{e(i)}, p_{(i)} + v_{e(i)}]$. [2]

The total extent bounded by a bounding segment S is the union of those extended MBBs from all points between the endpoints v_b and $(v_b + v_s)$, which can be expressed as:

$$ext(S) = \bigcup_{\alpha \in [0,1]} mbb_e(v_b + \alpha \cdot v_s)$$

5.4.3 Computing Bounding Segments

A bounding segment S covers a collection of points P if $P \subset ext(S)$. However, unlike MBBs, which are easy to calculate, the optimal bounding segment for a collection P is harder to discover, due to the many degrees of freedom. First, the axis v_s is hard to determine, as the search space grows exponentially with increasing dimensionality. Second, even once the axis has been fixed, the extent v_e is still not unique. It is always possible to increase ranges in some dimensions in order to reduce ranges in other dimensions, as illustrated in Figure 5.5.

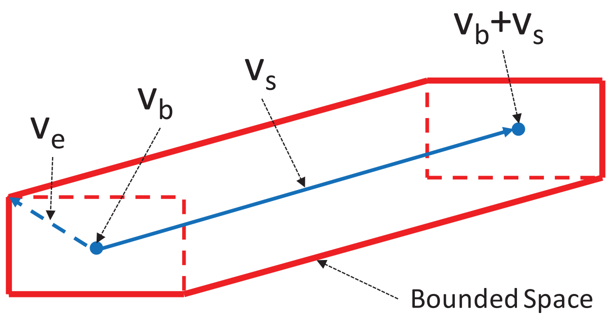


Figure 5.4: 2d bounding segment visualization.

Considering low overhead, we prefer simple algorithms that find bounding segments with reasonable but not necessarily optimal pruning effectiveness. Here we provide a linear solution for computing bounding segments in two

[2]We use $x_{(i)}$ to denote the i^{th} element of vector x.

steps: (1) determine the segment axis (direction of v_s); (2) determine the segment extent (v_b, v_e and scale of v_s). In our method, only the segment axis is maintained incrementally as points arrive. The segment extent is determined only after all points in the block are available.

Segment Axis

The segment axis is determined using *principal-component analysis* (PCA) techniques. PCA is a statistical procedure that finds orthogonal axes, so-called *principal components* (PCs), so that points are linearly uncorrelated when aligned to those components. The first PC has the largest variance, and each succeeding component has largest variance while being orthogonal to all preceding components.

In our scenario, a proper segment direction of v_s can be considered as the direction with largest variance, i.e., the first PC. However, calculating the exact first PC for n d-dimensional points has complexity $O(nd^2)$ or $O(n^2d)$. To ensure that the total procedure is linear in the data size, an incremental PCA (IPCA) algorithm, called CCIPCA [88], is adopted. It approximates the first k PCs in an incremental manner with only $O(kd)$ complexity per point. Another reason that an IPCA algorithm is desirable is that the support of incremental updates facilitates online segmentation (Section 5.5). CCIPCA is used because of its efficiency and simplicity for implementation, but other IPCA algorithms should also be applicable.

Note that the effectiveness of the bounding segment S does not depend on the sequential order of points inside. Once all points are close to the segment axis, S should have good pruning effectiveness, even if consecutive points are not always adjacent to each other. Therefore, the bounding segment can tolerate small discontinuities in the data, e.g., data disorder.

Segment Extent

The segment axis is the line passing through the mean along the direction of v_s. After obtaining the segment axis, we can "project" points onto the axis. Recall that the extent of a bounding segment is the union of all MBBs of points between two endpoints. For any point p represented by the bounding segment, there must exist at least one point p' on the axis so that $p \in mbb_e(p')$. To derive

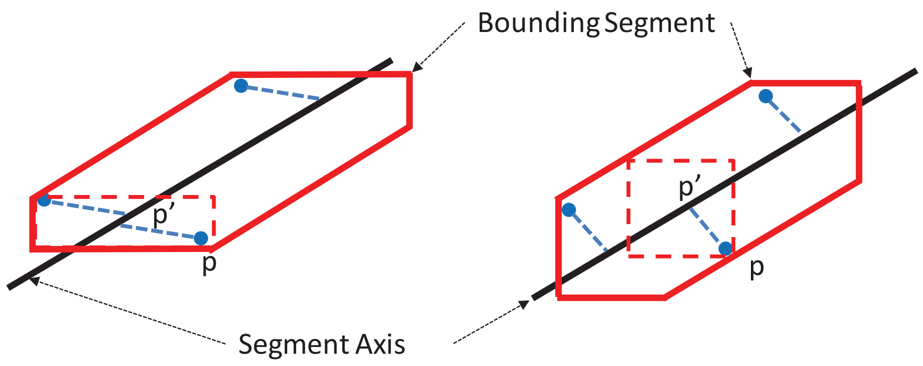


Figure 5.5: Different point projections onto the segment axis in 2-dimensions.

v_e, we need to find a p' for each point p and update v_e, such that:

$$v_{e(i)} = \max\left(v_{e(i)}, |p'_{(i)} - p_{(i)}|\right)$$

However, the projection of p to p' is not unique, and different projections will affect v_e (and thus $mbb_e(x)$ function). Figure 5.5 illustrates the effect of different projections.

To simplify this procedure, we project points onto the segment axis along norm vectors, which are orthogonal to the axis (as shown in the right part of Figure 5.5). This projection is optimized for the squared error between p and p'. The projected point p' can be expressed as:

$$p' = o_{mean} + \theta \cdot \frac{v_s}{\|v_s\|}$$

$$\theta = (p - o_{mean}) \cdot \frac{v_s}{\|v_s\|}$$

where o_{mean} is the mean of all points in the segment. The two endpoints are also easy to find, by recording the minimal and maximal θ encountered, so as:

$$v_b = o_{mean} + \theta_{min} \cdot \frac{v_s}{\|v_s\|}$$

$$v_s = (\theta_{max} - \theta_{min}) \cdot \frac{v_s}{\|v_s\|}$$

By this method, all three values in a bounding segment are computed. Note that the extent v_e could be replaced with two values, one for the upper bounds and the other for the lower bounds, to further reduce the total extent.

5.4.4 Matching against a Query

The main reason that we record the extent v_e aligned to the original axes is that this choice makes the test against queries extremely simple. The extent of a bounding segment S can be easily "transferred" to a query, by enlarging the query range. The intersection check for S (with v_e) against the query range $[q_{min}, q_{max}]$ is equivalent to the check of a pure segment (without extent) against the query range $[q_{min} - v_e, q_{max} + v_e]$.

To test the segment v_b to $v_b + v_s$ against the box $[q_{min} - v_e, q_{max} + v_e]$, we can decompose this task into multiple 1-dimensional overlap checks. For each dimension i, we use a pair (l_i, u_i) where $0 \leq l_i \leq u_i \leq 1$, to express the intersected portion:

$$[v_{b(i)} + l_i \cdot v_{s(i)}, v_{b(i)} + u_i \cdot v_{s(i)}]$$

$$= [v_{b(i)}, v_{b(i)} + v_{s(i)}] \cap [q_{min(i)} - v_{e(i)}, q_{max(i)} + v_{e(i)}]$$

The final test is true only if $\cap_{1 \leq i \leq n}[l_i, u_i] \neq \varnothing$.

5.4.5 Calculating Segment Volume

To support online segmentation algorithms (Section 5.5), we need to calculate (or at least estimate) the volume bounded by a segment, especially in an incremental way.

Once we have a bounding segment S, we can calculate an accurate volume of the bounded space as:

$$\text{Vol}(S) = \left(\prod_{i=1}^{n} 2v_{e(i)}\right) \cdot \left(1 + \sum_{j=1}^{n} \frac{v_{s(j)}}{2v_{e(j)}}\right)$$

However, the v_e component can only be obtained by processing all points in the block whenever the segment axis is updated. Hence, the complexity of getting the up-to-date volume after m points will be $O(md)$. For a sequence of n points, the total cost of getting the volume of the bounding segment after each incremental update will be $O(n^2 d)$, which is likely unaffordable.

We provide here a simple way to estimate the volume. Note that calculating the segment volume using the formula above only needs v_e and v_s. We can therefore estimate these two values in order to get an approximate volume. For the extent v_e, another value v_e' is maintained during incremental update. For each new point p, we first compute its projection p' on the current segment axis, and update the v_e' in the same way we calculate v_e. As the segment axis keeps adjusting, v_e' will no longer capture the real extent. However, so long as the segment axis changes only slightly during the updates, v_e' will be a good estimate of v_e. Following the same idea, we can estimate v_s incrementally. For each point p, we calculate the distance between p' and the axis origin. The approximate v_s' is therefore estimated from the two endpoints that are at maximal distance to the origin in opposite directions.

5.5 Indexing and Refining

This section presents the initial construction of SICC in a lightweight manner when ingesting new data, in-order to maximize write-throughput. We then discuss the strategy for continuously refining the index to improve query efficiency.

5.5.1 Index Construction

The SICC can be easily constructed when data records are initially ingested into log-storage, hence these data are ready for answering queries as soon as possible. To ensure throughput, the construction is lightweight. We segment consecutive records into disjoint blocks, each of which are then represented by a bounding segment.

The bounding segments are based on the insight that a collection of consecutive points are close to an "implicit" segment axis. Its effectiveness is more sensitive to the location of points, compared to that of MBBs, i.e., a point is less probable to be near a axis than within a box. Therefore, segmentation algorithms that determine the scope of each block (i.e., the start and end record of a block) are critical. We used an *eager segmentation* algorithm to take newly arrived records as input and divide them into logical blocks. For each block, the algorithm generates a *block header* as output. Block headers allow us to skip blocks of data that do not contribute results to a given query. All generated headers are appended into a *header file*. To avoid checking all headers for each query, we design *OR-trees* to index them in a write-optimized way.

Block Header

A block header basically contains two types of information: (1) that for block matching against queries, including a bounding segment; and (2) that for locating the block contents for retrieval, including the block id, file id, file offset and count of records. Given a block header and a query, the bounding segment is first tested against the query range. Only on a positive test will the location information be used to fetch the data block.

Eager Segmentation

We use an eager segmentation method to segment blocks during data ingestion due to its simplicity. It is extremely lightweight that requires only a single pass of data, with amortized $O(d)$ complexity per record. The main idea is that we maintain one active block to accept incoming records. When a new record arrives, we immediately decide whether to put it into the active block, or to close that block and initiate a new one. To guide the decision, we follow a rule: **For answering any query with r results, the number of accessed records should be less than $\mu \cdot r$, where μ (> 1) is an amplification factor.**

First, we address a sub-problem at the block level: given a block B with $|B|$ records and r^B as the number of expected results in B, how to determine whether fewer than $\mu \cdot r^B$ records will be accessed for answering queries. To make the analysis, pre-knowledge of the query workload is required. For range queries, the relevant aspects are: (1) the expected range extent of queries; and (2) the distribution of queries. To simplify the problem, we assume that query workload has a uniform distribution, and the expected extent of the query range is $\bar{l}$.

We define the *query-enlarged* area of a point p to be the MBB that has range $[p_{(i)}, p_{(i)} + \bar{l}_{(i)}]$ for each dimension i. For each point p, the probability that it is returned as a query result is proportional to the volume of its query-enlarged area, that is:

$$\text{Res}_{point}(p) \sim \prod_{i=1}^{n} \bar{l}_{(i)}$$

Algorithm 3: Eager Segmentation(Record r)

```
Input: a newly arrived record $r$
Output: a new block header or null
/* global variables */
1  Block $active$;
/* local variables */
2  Header $ret$ = null;
3  Boolean $closed$ = false;
/* check constraint */
4  add $r$ into $active$ and update segment axis;
5  if $active.enlargedVol$ / $active.PointsVol \geq \mu$ then
6      $closed$ = true;
7      remove $r$ from $active$;
/* check to close the block */
8  if $closed$ is true then
       /* create header for the block */
9      compute segment extent of $active$;
10     $ret$ = new Header($active$);
11     $active$ = new Block();
12     $active$.filePosition = $r$.position;
13     add $r$ into $active$;
14 return $ret$;
```

Hence, for the whole block B, the expected number of results r^B it contains can be expressed as:

$$r^B = \sum_{p \in B} \text{Res}_{point}(p) \sim |B| \cdot \prod_{i=1}^{n} \bar{l}_{(i)}$$

Now we consider the expected frequency with which B needs to be fetched and accessed, which depends on the bounding object representing the block. More specifically, the frequency is proportional to the query-enlarged area of the bounding object, which should cover the query-enlarged area of all contained points. For a block B, we need the query-enlarged segment S'_B. It should be the same as bounding segment S_B, except that the extent v_e is replaced by $v_e + \frac{1}{2}\bar{l}$. As a result, the access frequency of block B is proportional to Vol(S'_B). (See Section 5.4.5 for calculating segment volume.) For each access, the whole block will be fetched and scanned. Therefore, the expected number of accessed records is $|B| \cdot \text{Vol}(S'_B)$.

To ensure that fewer than $\mu \cdot r^B$ records are accessed, we have the following constraint:

$$\mu > \frac{\text{number of accessed records in } B}{\text{number of results in } B} = \frac{\text{Vol}(S'_B)}{\prod_{i=1}^{n} \bar{l}_{(i)}}$$

If every block meets this constraint, the total number of accessed records is expected to be less than $\mu \cdot r$. Note that no assumption on data distribution or data continuity is required. Algorithm 3 shows the eager segmentation procedure. We keep loading new records into the active block until it violates the constraint respect to a given μ, and then start a new block.

Header Indexing

Similar to data stored in log-storage, block headers are also organized sequentially on disk. Generated headers are appended to the end of a $headerfile$. When answering a query, the simplest way to check headers is a brute-force scan of the entire header file. Although such a method is not intelligent, it does have advantages: it always returns qualified headers in insertion order. This order facilitates the scan in underlying log-storage, as consecutive qualified data blocks can be fetched in a single round without extra disk seeks. In addition, final results can be returned in order as well, which benefits sequence analysis from a user's perspective.

Intuitively, we can adopt R-trees [50] to index block headers, using the MBB of the bounding segment as the spatial key. Although R-trees outperform brute-force scans on access, their high maintenance cost make them prohibitive for write-intensive scenarios. If observations come in a high rate, node splits and re-organizations will be the bottleneck. Also, as index entries are no longer accessed in insertion order, an additional sorting phase for qualified entries is required, if we want to fully utilize bandwidth to access data blocks.

To combine the strengths of R-trees and brute-force scans, we propose an R-tree variant that provides both write throughput and query efficiency. In our approach, the header file will be indexed by a number of sub-trees, termed *OR-trees*, and only their root nodes are put into a global R-tree. Each sub-tree is organized similarly to a normal R-tree, but constructed in a bottom-up manner in which block headers are grouped and indexed in insertion order. Hence we name it *Ordered R-tree* (OR-tree). Figure 5.6 illustrates this organization. As

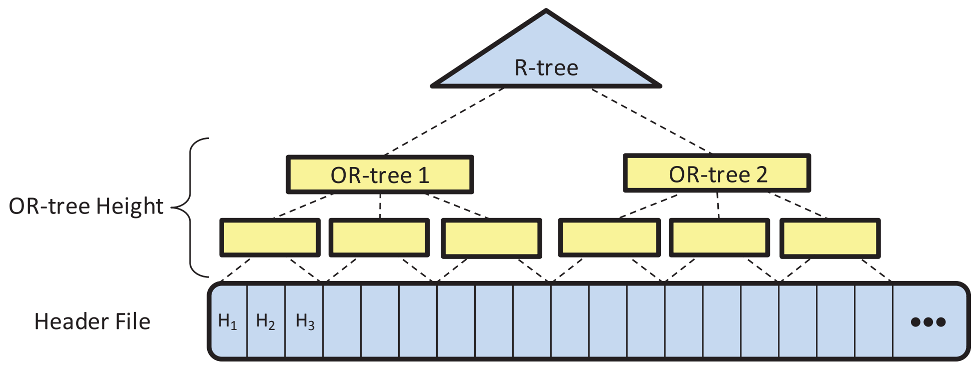


Figure 5.6: Indexing block headers with OR-trees of height 2.

consecutive observations are close together, the MBBs of internal nodes in an OR-tree should be compact and that property ensures its effectiveness. To keep qualified entries in order, we only need to sort the retrieved OR-tree roots from the global R-tree. All subsequent traversals inside an OR-tree naturally return headers in order. The height of the OR-trees reflects the trade-off between construction cost and effectiveness. In the extreme, we can maintain a single huge OR-tree to entirely eliminate node splits and sort phases.

Queries with time-range conditions are essential in scientific analysis. The header file and OR-tree support such queries well. It is straightforward to filter out an entire OR-tree that is disjoint from the time range, provided that each header contains temporal information. For supporting incremental queries over new data, we bypass the OR-tree indexes and directly scan the tail of the header file. We mark the current end of the header file after each round and then can continue to scan from the mark later. The specific procedure for answering range queries can be easily derived from the one-dimensional case 4.

5.5.2 Index Refinement

Given a bounded amount of system resources, `SICC` always ensures that all incoming data are indexed as soon as possible. The remaining resources can then be used to answer query requests. During index construction, eager segmentation relies on estimated query ranges and uniformity assumptions. Hence, query performance is not guaranteed. In practice, query workloads are complicated and data of interest may change drastically over time. We avoid the adaption in initial construction to keep it simple and fast. Instead, the index structure is continuously refined in the background when the system has available resources.

Refinement Criterion

The performance of recent query requests determines which parts of the index need to be refined. To collect query statistics, we assign a number of counters for each accessed block header B, as follows:

- C_{fetch}: the number of queries that matched B, hence its data block was fetched.
- $C_{co-fetch}$: the number of queries that fetched both B's data block and the following block.
- C_{result}: the total number of records that are returned as results from B's data block over all queries.

These counters are easy to maintain in memory, causing negligible overhead for queries. We can derive the actual query cost for block B as

$$cost(B) = C_{fetch}(B) \cdot |B|$$

This value is the main characteristic to help us determine if a part of the index needs to be refined. We can either *split* a block to improve efficiency, or *merge* two consecutive blocks to remove unnecessary index entries. The refinement frequency depends on the quality of segmentation against active queries. Thus, it tends to be low when segmentation algorithms are able to provide effective segments by themselves, or after extended period of similar queries.

Splitting and Exhaustive Segmentation

Recall that users provide a factor μ to limit expected read amplification (Section 5.5.1). With the help of the counters above, we can verify whether a block satisfies the users expectation. We might need to split a block B when

$$\frac{cost(B)}{C_{result}(B)} > \mu$$

"Split" means we replace a block's original header with two or more new headers, each covering a sub-block. Splitting costly blocks into smaller ones can reduce both scan cost and the false-positive rate. In the case that more than one block is to be split, we always choose the one with most "potential" benefit from splitting, i.e., the one with maximum $cost(B) - C_{result}(B)$.

To conduct the split, we use *exhaustive segmentation*, which takes the records in the original block as input and generates a number of new block headers as output. Compared to eager segmentation, this algorithm is more computational-intensive, as it considers every possible choice for splitting the block. The goal of exhaustive segmentation is **to segment a collection of records into K blocks that minimize the total query cost.**

Similar to previous analysis, we use $w(B) = |B| \cdot \mathrm{Vol}(S'_B)$ as the query cost of block B, where S'_B is its enlarged bounding segment. Note that we can use recent query requests to determine the enlarged area, instead of a pre-defined range expectation. For a segmentation of a sequence of records into K blocks $B_1, B_2, ..., B_K$, our target is to minimize the total query cost W as:

$$W = \sum_{i=1}^{K} w(B_i) = \sum_{i=1}^{K} |B_i| \cdot \mathrm{Vol}(S'_{B_i})$$

It is straightforward to solve this problem when $K = 2$. We just need two passes, i.e., forward and backward, to compute segments of a partial block and keep corresponding volumes. The split point can be found by enumeration in linear time, and we omit the details.

Sometimes a costly block may need to be split into $K > 2$ pieces. There are $\binom{n-1}{K-1}$ choices to spit K blocks with n records. It is prohibitive to explicitly enumerate all options. Fortunately, the problem can be solved efficiently by dynamic programming.

Suppose there are n records $r_1, r_2, ...r_n$. Let $B_{(i,j)}$ denote the block containing records from r_i to r_j, and $w(i, j)$ denote the query cost of $B_{(i,j)}$. Let $f(k, j)$ denote the minimum cost of segmenting r_1 to r_j into k blocks. Thus, $f(k, j)$ can be derived recursively as:

$$f(k,j) = \begin{cases} \min\limits_{k-1 \le i < j} (f(k-1,i) + w(i+1,j)), & k > 0 \\ 0, & k = 0 \text{ and } j = 0 \\ +\infty, & k = 0 \text{ and } j > 0 \end{cases}$$

Algorithm 4: Exhaustive-Core(Matrix w, Integer K)

```
   Input: the query-cost matrix $w$
   Input: the number of target splits $K$
   Output: a list of split positions
 1 List<Integer> $ret$;
 2 Integer $n = w$.size;
 3 Double $f[0 ... K][0 ... n]$;
 4 set all elements in $f$ to $+\infty$;
 5 $f[0][0] = 0$;
   /* compute optimal cost for f[K][n]                          */
 6 for $i = 0$ to $K$ do
 7     for $j = i$ to $n$-1 do
 8         if $f[i][j] == +\infty$ then continue;
 9         for $k = j + 1$ to $n$ do
10             if $f[i + 1][k] > f[i][j] + w(j + 1, k)$ then
11                 $f[i + 1][k] = f[i][j] + w(j + 1, k)$;
   /* backtrack segmented positions                             */
12 Integer $p = n$;
13 add $p$ into $ret$;
14 for $l = K$ downto 1 do
15     find $p'$ with $f[l][p] = f[l - 1][p'] + w(p' + 1, p)$;
16     add $p'$ into $ret$;
17     $p = p'$;
18 reverse elements in $ret$;
   /* $i$-th block is from record $ret[i] + 1$ to record $ret[i + 1]$     */
19 return $ret$;
```

No more than $K \cdot n$ terms of f need to be computed to obtain $f(K, n)$. The computational complexity is therefore $O(Kn^2)$. During the computation, The $w(i, j)$'s are required in advance. They can be pre-processed and stored within $O(dn^2)$ time and $O(n^2)$ space. Therefore, the overall cost is $O((K+d)n^2)$. The actual split positions can be found by backtracking through intermediate values. To further reduce the computation when n is large, we can select a subset of split positions, e.g., every 10 records, to reduce the search space. Algorithm 4 shows the core of exhaustive segmentation that takes pre-calculated $w(i, j)$'s and find the optimal K-way split.

Merge

In contrast to the split refinement, merge is used when data are fragmented into too many pieces through past splits, and those pieces are not so helpful for the current query workload. Specifically, we detect that a group of consecutive blocks are always accessed together by most queries. We can simply combine their index entries without compromising query efficiency. The $C_{co-fetch}$ counter helps us find all sequences of co-accessed blocks. Merging two co-accessed blocks should not introduce too much over-coverage. Hence, for a block B and the next block B', we can estimate the query cost of the merged block as:

$$\begin{aligned} merge(B, B') =& (C_{fetch}(B) + C_{fetch}(B') - C_{co-fetch}(B)) \\ & \cdot (|B| + |B'|) \end{aligned}$$

Two blocks are allowed to merge when the bound on read amplification is still met, i.e., $merge(B, B')/(C_{result}(B) + C_{result}(B')) < \mu$. When multiple candidates are available, we will choose the pair with least penalty for query performance, i.e., the one with minimum $merge(B, B') - cost(B) - cost(B')$. The merge procedure is quite simple, as we only need to produce a new block header for the merged block to replace old ones.

Minor Refinement v.s. Major Refinement

Based on available system resources and the number of accumulated refine-operations, we can conduct the index refinement in two different ways, namely minor refinement and major refinement.

A *minor refinement* is preferred when resources are limited and we only need to change a small number of blocks. In this case, the header log is only slightly modified. During minor refinement, no locks are required and no queries will be blocked. For each OR-tree, an additional patch file is maintained to keep all updates. Generated headers from splits and merges are appended to the patch file. Afterwards, an atomic modification in the header file is done to tag old headers and redirect to new ones. The OR-tree is left unchanged, as we will access the header file as normal but jump to the patch file when a tagged header is encountered.

A *major refinement* is needed when the patch file grows too large. A long patch file causes a chain of jumps before we can get updated headers. During

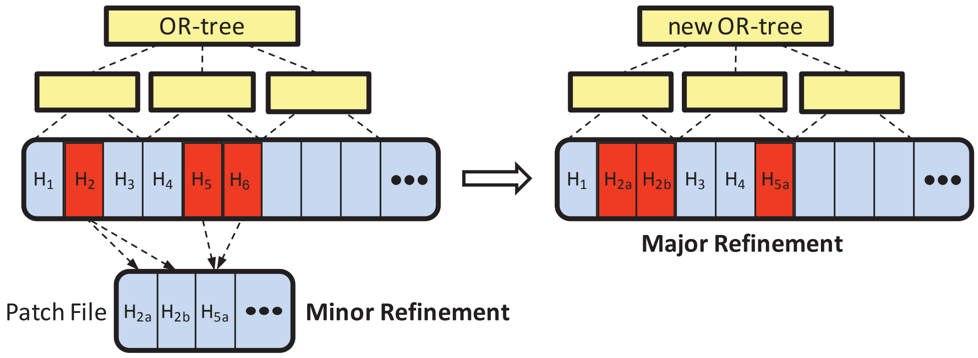


Figure 5.7: Minor refinement and major refinement.

major refinement, header chains are eliminated by re-writing all updated headers back into the header log, and the patch file is discarded. If there are any headers that cannot fit in the header file, they will be kept in a new but smaller patch file. After the re-writing, a new OR-tree is constructed from scratch. We delete the old OR-tree root from the global R-tree and insert the new one.

Figure 5.7 shows how minor and major refinement are conducted. In minor refinement, H_2 is split to H_{2a}, H_{2b} and H_5, H_6 are merged to H_{5a}. In major refinement, H_{2a}, H_{2b}, H_{5a} in the patch file are written back to the header file, and a new OR-tree is built. Note that, at all times, underlying data are not changed, so we can support time-range queries and sequence analysis efficiently. Better performance might be obtained by re-clustering the data and building a new `SICC` on top of them, we leave that possibility to future investigation.

5.6 Experimental Evaluation

In this section, we evaluate the performance of the proposed `SICC` index and alternative methods for indexing and querying multi-dimensional observational data. We consider two aspects of performance: index-maintenance overhead and query efficiency.

5.6.1 Data Sets

Experiments are conducted on three real datasets. All of them are observations collected from sensing devices, but with differing application scenarios and degrees of value continuity.

- **Coastal-Margin Observation (CMOP):**
  This dataset contains coastal margin observations collected from the CMOP [1] SATURN observing system. The data were collected between April 2011 and August 2012 from station SATURN-01. Each record contains diverse variables reflecting ocean and river status, such as salinity, temperature and oxygen saturation.

- **Hi-Tech Equipment Observation (POWER):**
  This dataset[3] contains status observations for a huge hi-tech manufacturing installation. Monitoring data is recorded by the manufacturing equipment itself using an embedded PC and a set of sensors. Each record contains power consumption and state flags for sub-components. The power consumption may differ significantly due to the change of running status.

- **Real-time Soccer Observation (SOCC):**
  This dataset[4] contains moving-object observations collected by a real-time location system on a soccer field. The data were generated by senors embedded in balls during a soccer game. Each record contains motion status, such as position, velocity and acceleration. We use observations from one ball (sensor 8) as a continuous source.

5.6.2 Methods and Implementations

We compare the `SICC` index with three baseline methods, based on PH-trees [90] and R-trees [50]. All methods are implemented as secondary indexes, in which index entries contain record references to log-storage, and storage accesses are required for fetching final results. All implementations are in Java, and integrated into `LogBase`.

- **`SICC`.** Our proposed index framework (Section 5.3) using bounding segments (Section 5.4). It applies eager segmentation and OR-tree header-indexing (Section 5.3).

- **PH-tree.** A state-of-the-art point access method based on binary PATRICIA-tries and hypercubes, reported to outperform other PAMs, such as kD-

[3]http://www.csw.inf.fu-berlin.de/debs2012/grandchallenge.html
[4]http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails

trees and critical-bit trees. We index each observation individually in the PH-tree. We use the implementation provided by its author. Note that this implementation is memory-based, while other methods we used are disk-based.

- **R-tree.** A well known spatial access method. Each observation is individually indexed in an R-tree. We use an open-source implementation for it[5].
- **R-block.** A block-oriented variant of R-trees. A fixed number of consecutive records are first grouped as a block, and the block is then indexed (using its MBB) in the R-tree.

In order to evaluate the effect of our design choices, we further compare `SICC` with different settings. For the segmentation algorithms, we have: (a) **Fixed**: fixed-length segmentation, which segments blocks into equal lengths; (b) **Eager**: eager segmentation; and (d) **Exhaustive**: buffers batches of records and then applies the exhaustive algorithm to minimize estimated query-cost. For the block-header indexing methods, we have: (a) **HeadLog**: no index for the header log; (b) **HeadRtree**: an R-tree; and (c) **HeadORtree**: an OR-tree.

5.6.3 Experimental Setup

All experiments were conducted on a server with a quad-core processor, 8GB physical memory and 500GB disk capacity. Secondary indexes are built on three dimensions for each dataset: CMOP (salinity, temperature and oxygen saturation), POWER (power consumption for three components), SOCC (speed and positions in two dimension). The numbers of records ingested into the storage are 13 million, 10 million and 7 million, respectively. All range queries are pre-generated randomly with 1% expected coverage of the indexed space for each. A group of 100 queries is issued sequentially, after each fifth of data is ingested.

We use $\mu = 4$ as the default setting for eager segmentation, which leads to different block-lengths in each dataset, as shown in Table 5.1. Hence, block-lengths in fixed-length segmentation (for both `SICC` and R-block) are configured differently across datasets to be consistent with the eager segmentation. The

[5]https://github.com/oschrenk/spatialindex

average block-length quantifies the value-continuity of a dataset, i.e., the longer the better.

Table 5.1: Average number of records in a block with different μ.

	$\mu = 1.2$	$\mu = 2$	$\mu = 4$	$\mu = 8$
CMOP	24.1	41.8	**84.7**	181.1
POWER	54.9	117.5	**207.8**	359.9
SOCC	152.6	325.5	**583.4**	908.2

5.6.4 Write Overhead

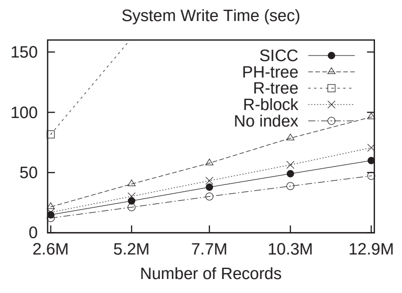
Index overhead is critical in write-intensive scenarios. This subsection evaluates that overhead, on both time and space consumption, for different approaches.

System-Load Time

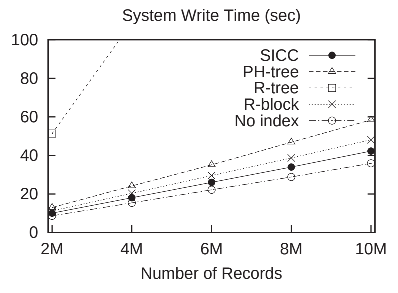
Figure 5.8 shows the total time for loading data into storage and having them indexed under different workloads. Among all indexes, `SICC` has the lowest system-load time for all three workloads. It exhibits indexing overhead of at most 20% (and only 5% in SOCC), compared to the cost of storing data in the storage without indexes (No Index). This superior performance is expected, as the number of index entries is reduced by orders of magnitude compared to record-level indexes. The R-tree degrades system performance unacceptably, as frequently inserting new entries causes frequent node splits and index reorganization. PH-tree, a point access method, exhibits much lower insertion cost compared to R-tree. However, its overhead is still significant. As can be seen, even the in-memory PH-tree is worse than the disk-based `SICC` and R-block. Overall, block-level indexes are more competitive in loading write-intensive workloads.

Index-Maintenance Cost

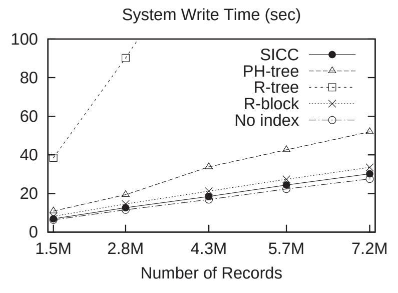
By removing the loading time inside storage, Figure 5.9 shows the pure index cost clearer with CMOP workload. We can see that the `SICC` has less than half the cost than any other approach. Though constructing a bounding segment is a bit costlier than an MBB, with the OR-trees, `SICC` outperforms the R-block. On closer examination of the results, we find that about three-quarters of the



(a) CMOP



(b) POWER



(c) SOCC

Figure 5.8: Overall system-load time.

cost of R-block comes from R-tree construction. We also observe that the cost of continuous index refinement is negligible compared to that of creating initial segments and header indexes, and omit the comparison here.

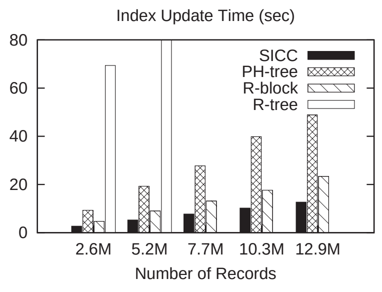
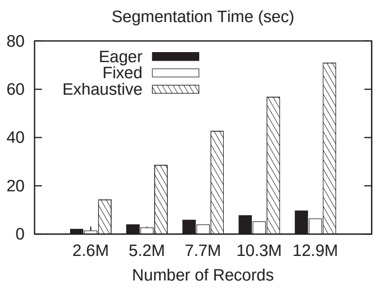


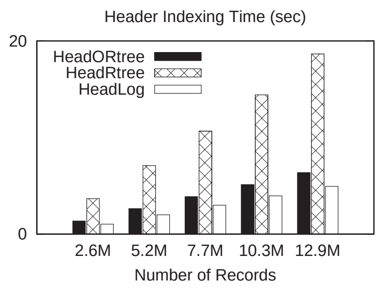
Figure 5.9: Index-maintenance cost. (CMOP)

Maintenance Cost of SICC Components

We further decompose SICC's cost into segmentation cost and header-indexing cost. Figure 5.10 contains comparisons of design choices on the CMOP workload. As can be observed from Figure 5.10(a), eager segmentation (Eager) costs only slightly more than naive fixed-length segmentation (Fixed). The dominant cost for these two methods is the I/O for persisting block headers. In contrary, exhaustive segmentation (Exhaustive) is dominated by expensive computations. To get the best segments, it buffers a large number of records and conducts exhaustive segmentation batch by batch. Such a method is costly and delays the availability for new data, hence is unaffordable for initial index construction. However, that segmentation is suitable for continuous refinement in the background.



(a) Segmentation



(b) Header Indexing

Figure 5.10: Decomposed `SICC` maintenance cost for different segmentation algorithms and header indexes. (CMOP)

Figure 5.10(b), also on CMOP workload, presents the choices for indexing block headers, i.e., header log. The result shows that indexing header log with an OR-tree (HeadORtree) is nearly as efficient as just maintaining a header log (HeadLog), due to the fast construction of append-only sub-trees. However, the overhead of constructing an entire R-tree (HeadRtree) is considerable.

Index-Space Consumption

Table 5.2 lists the disk-space consumption of different indexing methods. `SICC` requires slightly more space than R-block, as the representation of bounding segments is larger than for MBBs, i.e., segment needs three arrays (v_b,v_s,v_e) while MBB needs just two arrays (lower and upper bounds). As can be seen, the R-tree consumes much more disk-space in order to index individual records. Since the PH-tree implementation is totally in-memory, we omit its disk consumption.

Table 5.2: Disk Consumption for indexes.

	SICC	R-block	R-tree	PH-tree
CMOP	16.0M	13.6M	739M	N/A
POWER	6.1M	5.1M	568M	N/A
SOCC	1.1M	1.1M	407M	N/A

5.6.5 Query Efficiency

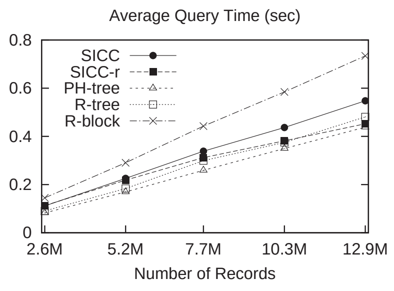
This subsection focuses on the evaluation of query performance. We evaluate the overall query response time for different datasets. Decomposed costs for data access and index lookup are also analyzed.

Query Execution Time

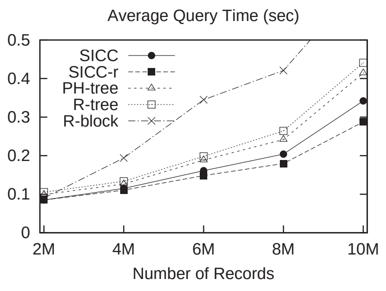
Figure 5.11 presents overall query-response time using different indexes. For both CMOP and POWER, R-block cannot achieve satisfactory efficiency, due to the over-coverage introduced by MBRs. In contrary, `SICC` has comparable efficiency to record-level indexes. R-tree and PH-tree have similar efficiency, since the dominant costs are the I/O for fetching results from storage. However, since they cannot return index entries in insertion order, random disk seeks are unavoidable when fetching data. That is the reason that `SICC` and R-block can beat them in SOCC workload. When dealing with highly continuous data, scans with larger blocks can benefit more from high bandwidth. `SICC` performs even better when it is continuously refined at the background (`SICC`-r). This result verifies our design of bounding segments as well as the concept of intrinsic clustering in observational data.

Data-Access Cost

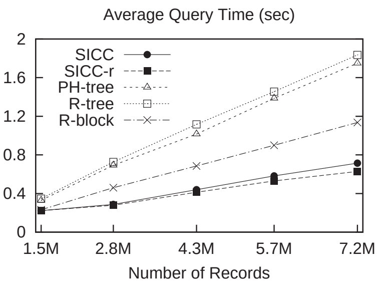
Figure 5.12(a) examines the data-access cost after obtaining block references (in `SICC` and R-block) or record references (in PH-tree and R-tree). Fetching records from storage is often the dominant cost of executing a range query. Hence, it follows the same trend as query-response time. As shown in the figure, PH-tree and R-tree have the lowest data-access cost, as both of them only fetch disk pages that are guaranteed to contain results. `SICC` and R-blocks may read disk pages with no results, because of over-coverage by bounding objects. Even with false-positive accesses, `SICC` still performs quite well, with the help of bounding segments.



(a) CMOP



(b) POWER



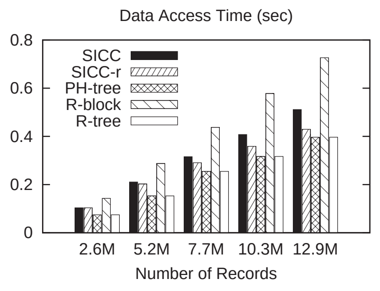
(c) SOCC

Figure 5.11: Average query-response time.

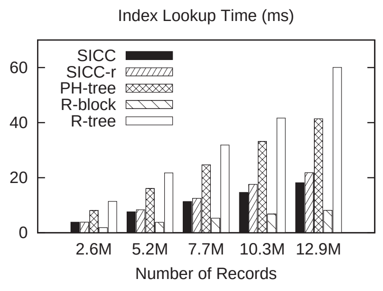
Figure 5.13(a) presents `SICC` data access cost under different segmentation algorithms. Exhaustive segmentation is expected to have superior performance. With a reasonable amplification factor μ, eager segmentation is also efficient. Considering its low overhead, the performance is more than satisfactory. Since the fixed-length segmentation does not consider data distribution, it rarely achieves good pruning effectiveness.

Index-Lookup Cost

Figure 5.12(b) presents index-lookup cost. Among all methods, R-block has the lowest cost, as all block headers are well organized in an R-tree and only a small number of block headers are tested during a lookup. For `SICC`, the higher lookup time is attributed to the ordered OR-trees, which affect the pruning effectiveness. In addition, checking a bounding segment requires more computation than checking an MBB. It is not surprising that lookups in the PH-tree and R-tree are expensive, since both of them contains many more index entries than block-level indexes. Such performance gaps are wider in workloads with better value-continuity.

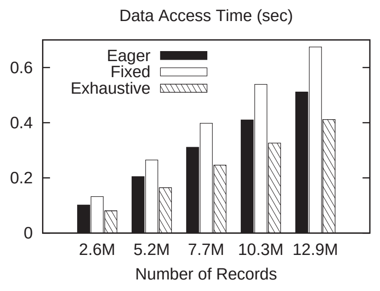


(a) Data Access

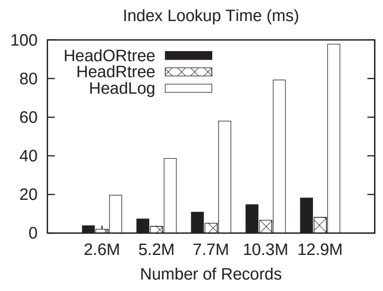


(b) Index Lookup

Figure 5.12: Decomposed query cost. (CMOP)



(a) Data Access



(b) Index Lookup

Figure 5.13: Decomposed `SICC` query cost for different segmentation algorithms and header indexes. (CMOP)

Figure 5.13(b) illustrates the lookup cost in `SICC` with different header-indexing methods. The cost of scanning an entire header log is high, as expected. Overall, OR-tree appears a good choice for indexing headers, as its maintenance overhead is nearly as low as a pure header log, and its query efficiency is comparable to the R-tree's.

5.6.6 Exploratory Study

In this subsection, we explore the effectiveness of bounding objects and the effect of query selectivity and dimensionality.

Bounding Segments vs. Bounding Boxes

Table 5.3 shows the huge gap between bounding segments and bounding boxes, in terms of bounded volume and estimated query cost. As can be seen, bounding segments reduce over-coverage by an order of magnitude. their query-costs ares also less than half that of bounding boxes'. We can also confirm the effectiveness of different segmentations from Table 5.3.

Table 5.3: Volume and Access-Cost in the CMOP dataset (with segments produced by fixed-length segmentation as the baseline)

Vol/Cost	Fixed	Eager	Exhaustive
Segment	1.00/1.00	0.55/0.92	0.43/0.77
Box	8.64/2.46	6.74/2.19	4.20/1.38

Query Selectivity

Figure 5.14 shows the effect of query selectivity on SICC, PH-tree and R-Block. As query selectivity increases, result size decreases. However, the number of false-positive blocks does not drop as rapidly, so their relative effect is greater at high selectivities. Thus, PH-tree, which only accesses true-positive blocks is at an advantage over SICC. Moreover, even for true-positive blocks, SICC may get only a small fraction of result records from each block. However, at low selectivities, false-positive blocks are less of a factor, and each true-positive block is likely to contain many result records, so SICC gains the edge. SICC is better than R-block at all selectivities, because over-coverage of the latter means more false-positive blocks.

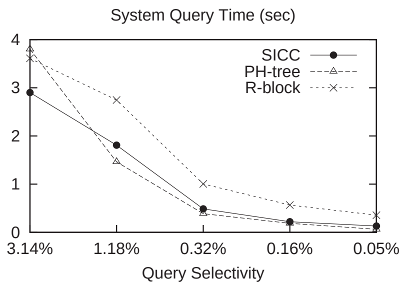
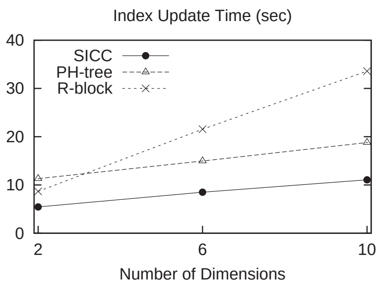


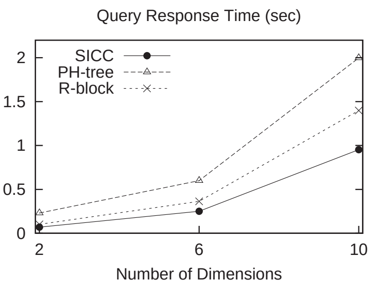
Figure 5.14: Query selectivity. (CMOP)

Dimensionality

How many dimensions can `SICC` handle? In high dimension, `SICC` can capture correlations from the PCA component to achieve good pruning effectiveness. To be fair in the comparison, we generate a synthetic dataset, in which each dimension is independent to minimize correlations. Figure 5.15 illustrates the results of the study. In each test, we evaluates the overhead of indexing 5 million records, and the average query time of 100 random queries. In Figure 5.15(a), we observe that the maintenance overhead is scale well for both PH-tree and `SICC`. In fact, insertion cost in PH-tree is proportional to the number of bits in a record. For `SICC`, eager segmentation and OR-tree have linear complexity to the number of dimensions. Figure 5.15(b) shows that the `SICC` also achieves the best query scalability among the approaches.



(a) Maintenance Overhead



(b) Query Efficiency

Figure 5.15: Index performance with increasing number of dimensions. (Synthetic Data)

5.7 Summary

It can take large amounts of system resources and time to index write-intensive observational data that arrives as streams. To reduce index cost, we propose a lightweight index method, called `SICC`, that incurs little construction overhead while efficiently supporting multi-dimensional range queries. Unlike conventional methods that cluster similar points, we exploit the intrinsic data continuity in observations, and construct indexes on local data sequences. The bounding segment is proposed to overcome the "over-coverage" problem of MBBs. It can be derived quickly and readily supports range queries. The index can be continuously refined in the background to further improve query performance. Experimental studies verify the feasibility of this index method, and confirm that the overhead of `SICC` is an order of magnitude lower than conventional indexes, while it preserves comparable query efficiency.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Nowadays, data workloads in our daily life are rapidly turning to large-scale and write-heavy, due to both requirements from modern applications and supports from cheap storage hardware. Such write-intensive trends pose new challenges to conventional solutions for both storage and indexing, i.e., high write-throughput must be guaranteed when handling unpredictably high data ingestion rate. This dissertation studied the research problem for managing write-intensive data, focusing on system write-throughput and query performance. In particular, we adopted log-structured techniques in storage to remove write bottleneck, and we exploited observational data traits in indexing methods to lower index construction cost and improve query efficiency.

First, we investigated that log-structured storage is a amendable choice for write-intensive workloads. In most state-of-the-art storage systems, ingested data are redundantly stored as at least two copies: one in write-ahead-log (to ensure durability) and another in main data repository (to optimize data access efficiency). Though such strategies improve query performance, the separation of log and application data also means nearly doubling of I/O cost, which limits throughput for write-intensive applications. On the contrary, in log-structured approaches, new arrival data are directly written into an append-only log, which serves both recovery and application queries. For write-intensive workloads, this approach has two direct advantages: first, data redundancy is completely eliminated; second, amortized write cost can be further reduced by sequential writes. Besides, recovery from failure do not need additional redo operations to reflect missing updates from log.

Based on the insights above, we implemented `LogBase`, a scalable database system with log-only repository. It is designed to be dynamically deployed in the cloud environment. It provides similar recovery capability to traditional write-ahead-logging approach while offering highly sustained throughput for write-heavy applications. To support fast key-based record access, we designed a multiversion index strategy, which can efficiently serve long tail requests. Further, we enhanced `LogBase` to support transactional semantics for read-modify-write operations and provide snapshot isolation. We conducted an extensive performance study on `LogBase`. The results confirm its efficiency and scalability in terms of write and read performance, as well as effective recovery time during node failure.

Second, we proposed a novel idea for constructing lightweight indexes on write-intensive observational data, exploiting intrinsic clustering features inside original data sources. Traditional indexing methods tend to incur prohibitive overhead when indexed data are write-intensive, due to frequent re-organization operations, such as node split in B$^+$-trees. That is because they always organize records with closest values into the same physical page to save future query I/Os. As records continuously arrive, index organization keeps changing to put new records nearby similar ones, introducing inevitable overhead. We refer such arrangement as *induced data clustering*. In contrast, we found that there is *intrinsic data clustering* in many real data sources, such as observational data, which naturally provides similar effect without extra overhead. By simply appending new records into contiguous disk pages, potential results of a query tend to be grouped together. Though such clustering is not perfect, we can still expect many query results from a single disk read.

Based on the this idea, we enhanced `LogBase` with the capability of ingesting and querying high-rate observational data. We designed a storage scheme for storing observational data in log-store that preserves data locality to facilitate indexing. On top of this physical organization, we proposed `CR-index`, a lightweight pruning-based index structure for range queries tailored for observational data, using efficient sequential I/Os. It lowers maintenance costs by combining multiple records in a single index entry and representing them together with a boundary pair. The sequential scan of potential data blocks in log-store guarantees the correctness and efficiency of index lookup. A number of optimization were applied to further reduce index size and utilize disk bandwidth. We conducted an extensive experimental evaluation on two real-world observational datasets that compares to traditional record-level indexes. The results confirm both low write overhead and query efficiency.

Third, we extended the idea of intrinsic clustering to multi-dimensions. We believe that this clustering property will become more attractive in multi-dimensional spaces. As can be proved, when storing a multi-dimensional record on disk, its closed neighbors on each dimension cannot be all physically nearby. That means we cannot find a perfect induced clustering for all dimensions even with exhaustive efforts, but intrinsic clustering can still provide reasonable effectiveness without affecting system throughput. However, it is not trivial to extend such an idea to multi-dimensions, due to data sparseness. Unlike bounding pairs, bounding objects in multi-dimension will cause "over-coverage", i.e., portions of indexed spaces containing no data. Accessing such a entry may introduce a false-positive hit. Hence, a good bounding representation is critical for query performance, while its complexity of derivation is critical for write throughput.

As a major contribution, we proposed a novel multi-dimensional indexing framework, called `SICC`, which generalizes the exploitation of intrinsic clustering. To overcome inherent data sparseness, we investigated a new representation scheme, which represents consecutive observations as a bounding segment in hyperspaces. Its effectiveness is ensured by the fact that observational points collected during a short time period nonetheless can be estimated as a segment. To minimize index overhead, we designed fast algorithms for deriving bounding segments based on incremental principle component analysis techniques, and proposed an R-tree variant for indexing generated segments. Besides, to further improve query efficiency at runtime, indexes are continuously refined based on the results from recent queries. From experimental evaluation on three real-world datasets, we verified the effectiveness of hyper-segments and the efficiency of the approach.

In summary, we concluded that log-structured storage is suitable for managing write-intensive workloads, in which write through is of great importance.

The utilization of sequential disk access and elimination of separate log improve system performance gracefully. Its undesirable read performance can be compensated by smart indexing approaches. We showed that intrinsic clustering in many real workloads is feasible to be exploited by indexing approaches, especially for write-intensive workloads where there is no resources for building powerful indexes.

6.2 Future Work

In the future, we plan to investigate the feasibility for applying the intrinsic clustering property and techniques proposed in Chapter 4 and 5 to general data sources. Constructing indexes using intrinsic clustering property from data sources is a novel idea, which has not been studied before. Though in this dissertation, we only shows its effectiveness on observational data, we believe that many other real-world workloads also have somehow different degree of intrinsic clustering property. For those data sources with only weak clustering property, it is possible to utilize intrinsic clustering on those parts of data which have, and create induced clustering on the rest. For example, a hybrid structure of `CR-index` and B+-tree can be used to cover the whole spectrum of intrinsic clustering degree inside different workloads. There would be an interesting tradeoff between the index construction cost and query efficiency, according to the degree of intrinsic clustering we want to exploit.

Another possible direction from storage perspective is to utilize large memory on top of log-structured disk storage to further improve update-intensive workloads. Unlike pure write-intensive workloads where data are rapidly growing so that disk repository is necessary, update-intensive workloads will not have their data volume exploding too fast. As the capacity of DRAM in modern servers are becoming larger, it is feasible to make DRAM as the main repository for update-intensive workloads, while using log-structured storage as a durable backend. It is interesting to have a storage system that can handle both exploding and constant-in-volume write-intensive applications. It should automatically distribute mixed workloads into the cluster while fully utilize large memory in each individual server.

BIBLIOGRAPHY

[1] CMOP. http://www.stccmop.org/. vii, 6, 55, 57, 76, 90, 109

[2] DEBS 2013 Grand Challenge. http://www.orgs.ttu.edu/debs2013/index.php. 76

[3] Hadoop MapReduce. [Online] http://hadoop.apache.org/mapreduce. 26

[4] HBase. http://hbase.apache.org. 13, 16, 17, 18, 22, 23, 27, 29, 31, 32, 33, 37, 38, 42, 58

[5] HDFS. [Online] http://hadoop.apache.org/hdfs. 28

[6] JDBM3. https://github.com/jankotek/JDBM3. 75

[7] LevelDB. https://github.com/dain/leveldb. 75

[8] MongoDB. http://www.mongodb.org. 4

[9] NetCDF. http://www.unidata.ucar.edu/netcdf. 55

[10] PrimeBase. http://sourceforge.net/projects/pbxt/. 13

[11] Zookeeper. http://zookeeper.apache.org. 38

[12] D. Achakeev and B. Seeger. Efficient bulk updates on multiversion b-trees. *Proc. of VLDB Endow.*, 6(14):1834–1845, 2013. 6, 14

[13] H. K. Ahn, N. Mamoulis, and H. M. Wong. A survey on multidimensional access methods. Technical report, HKUST, 2001. 16, 88

[14] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *Proc. of VLDB Endow.*, 7(10):841–852, June 2014. 6, 87

[15] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority R-tree: a practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4(1):9, 2008. 16

[16] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, pages 223–234, 2011. 24, 25

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2), May 1990. 16

[18] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975. 16

[19] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. *Proc. of Int. Conf. on VLDB*, pages 28–39, 1996. 16

[20] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proc. of SIGMOD*, pages 1–10, 1995. 38, 39

[21] P. A. Bernstein and N. Goodman. Multiversion concurrency control: Theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983. 37

[22] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *Proc. of CIDR*, pages 9–20, 2011. 14, 28, 50

[23] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. 16, 64

[24] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006. 27

[25] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proc. of SIGMOD*, pages 729–738, 2008. 40

[26] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. 22, 23

[27] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. of OSDI*, pages 205–218, 2006. 4, 13, 17, 18, 22, 27, 29

[28] L. Chen, R. Choubey, and E. A. Rundensteiner. Bulk-insertions into R-trees using the small-tree-large-tree approach. *Proc. of ACM Int. Symp. on GIS*, pages 161–162, 1998. 6, 15, 87

[29] E. Cheung, S. Nadeau, D. Landing, M. Munson, and J. Casper. Sensor data & analysis framework (SDAF) data warehouse. *Technical report, MITRE*, 2007. 17

[30] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: a generalized R-tree bulk-insertion strategy. *Proc. of Int. Symp. on SSD*, pages 91–108, 1999. 6, 15, 87

[31] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. 1

[32] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. 14, 55

[33] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008. 24, 37

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of SoCC*, pages 143–154, 2010. 44, 47

[35] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010. 25

[36] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proc. of SOCC*, pages 163–174, 2010. 25

[37] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. 65

[38] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, pages 10–10, 2004. 26

[39] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of SOSP*, pages 205–220, 2007. 4, 23

[40] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *Proc. of CIDR*, pages 195–198, 2011. 14

[41] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005. 39

[42] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974. 16

[43] I. K. Fodor. A survey of dimension reduction techniques. Technical report, 2002. 94

[44] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998. 16

[45] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007:1–16, 2012. vii, 2, 3

[46] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with DataDepot. In *In Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 847–854, 2009. 17

[47] G. Graefe. B-tree indexes for high update rates. *ACM SIGMOD Record*, 35(1):39–44, 2006. 14

[48] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proc. of Int. Conf. on EDBT*, pages 371–381. ACM, 2010. 6, 15

[49] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system r database manager. *ACM Comput. Surv.*, 13(2):223–242, 1981. 12, 21

[50] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984. 7, 16, 88, 102, 109

[51] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983. 37

[52] F. Halim, S. Idreos, P. Karras, and R. H. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, 2012. 15

[53] R. A. Hankins and J. M. Patel. Data morphing: an adaptive, cache-conscious storage technique. In *Proc. of VLDB*, pages 417–428, 2003. 24

[54] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. of USENIX*, pages 11–11, 2010. 27, 38

[55] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, volume 3, pages 1–8, 2007. 6, 15

[56] S. Idreos, S. Manegold, and G. Graefe. Adaptive indexing in modern database kernels. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 566–569. ACM, 2012. 6

[57] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2):332–342, May 1990. 92

[58] H. V. Jagadish. Spatial search with polyhedra. *Proc. of Int. Conf. on Data Engineering*, pages 311–319, 1990. 16, 88

[59] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. *ACM SIGMOD Rec.*, 26(2):369–380, 1997. 7, 88

[60] W. Ku and G. Center. The cloud-based sensor data warehouse. In *Proc of ISGC 2011 & OGF 31*, volume 1, page 75, 2011. 17

[61] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010. 4, 13, 16, 17, 18, 23, 37, 38

[62] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981. 30

[63] D. B. Lomet and B. Salzberg. Access methods for multiversion data. In *Proc. of SIGMOD*, pages 315–324, 1989. 31

[64] D. B. Lomet and B. Salzberg. Exploiting a history database for backup. In *Proc. of VLDB*, pages 380–390, 1993. 14

[65] K. Markov, K. Ivanova, I. Mitov, and S. Karastanev. Advance of the access methods. *Information Technologies and Knowledge*, 2(2):123–135, 2008. 16

[66] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992. 4, 12, 21

[67] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The lham log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, 2000. 16

[68] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, Mar. 1984. 16

[69] K. Nrvåg. The vagabond approach to logging and recovery in transaction-time temporal object database systems. *IEEE Trans. on Knowl. and Data Eng.*, 16(4):504–518, 2004. 14

[70] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999. 13

[71] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree. *Acta Inf.*, 33(4):351–385, 1996. 6, 15, 18, 31, 50, 55, 87

[72] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proc. of SOSP*, pages 29–41, 2011. 14, 23, 50

[73] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of SIGMOD*, pages 109–116, 1988. 27, 28

[74] D. Pfoser. Indexing the trajectories of moving objects. *IEEE Data Eng. Bull.*, 25(2):3–9, 2002. 17

[75] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *Proc. of VLDB*, pages 395–406, 2000. 17

[76] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992. 13

[77] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, 2014. 5, 14

[78] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proc. of ACM SIGMOD*, pages 217–228, 2012. 2, 6, 16, 18, 20, 87

[79] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An implementation of a log-structured file system for unix. In *Proc. of USENIX*, pages 3–3, 1993. 5, 13

[80] G. Sfakianakis, I. Patlakas, N. Ntarmos, and P. Triantafillou. Interval indexing and querying on key-value cloud stores. *Int. Conf. on Data Engineering (ICDE)*, 0:805–816, 2013. 66

[81] L. Sidirourgos and M. Kersten. Column imprints: a secondary index structure. *Proc. of ACM SIGMOD*, pages 893–904, 2013. 18

[82] M. Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, Apr. 2010. 4

[83] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proc. of SIGMOD*, pages 340–355, 1986. 12, 21

[84] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of postgres. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):125–142, 1990. 12, 21

[85] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Trans. on Knowl. and Data Eng.*, 10(1):173–189, 1998. 40

[86] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *Proc. of ACM SIGKDD*, pages 216–225. ACM, 2003. 18

[87] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010. 25

[88] J. Weng, Y. Zhang, and W.-S. Hwang. Candid covariance-free incremental principal component analysis. *IEEE Trans. on PAMI*, 25(8):1034–1040, 2003. 96

[89] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, 2015. 16, 18

[90] T. Zäschke, C. Zimmerli, and M. C. Norrie. The PH-tree: a space-efficient storage structure and multi-dimensional index. *Proc. of ACM SIGMOD*, pages 397–408, 2014. 16, 109

[91] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *Proc. of ACM SIGMOD*, pages 1555–1566. ACM, 2014. 18