Lightweight Indexing of Observational Data in Log-Structured Storage

Sheng Wang #1, David Maier †2, Beng Chin Ooi #3
National University of Singapore, † Portland State University
1,3 {wangsh,ooibc}@comp.nus.edu.sg, 2 maier@cs.pdx.edu

ABSTRACT

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 paper, 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.

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 work 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 [10] and LSM-trees [17], 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 [2] utilize a combination of RDBMS and netCDF [6] 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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 7
Copyright 2014 VLDB Endowment 2150-8097/14/03.

physical order perform well (and can be further improved through off-line reorganization [20]). 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

- 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.
- 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.
- 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 paper is organized as follows. In Section 2, we provide background on observational data and storage choices. Section 3 presents a scheme for storing data. In Section 4, we present the design of our indexing structure. We evaluate the performance in Section 5. Related work and conclusions are given in Sections 6 and 7.

2. PRELIMINARIES

This section presents some characteristics and applications of observational data and common query types. We introduce an open source system, *LogBase* [20], which is the choice of storage in our implementation.

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*.

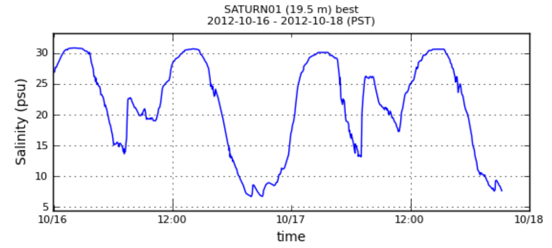


Figure 1: CMOP SATURN-01 salinity trend

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.

2.1.1 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 1 (from observation station SATURN-01 in the CMOP [2] 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 1 for salinity above 32.0, then analyze how such periods of high salinity correlate with oxygen saturation and acidity.

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 4.6.

2.1.2 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.

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.

2.2.1 Storage Types

In *ordered* log-store, such as HBase [1], 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.

2.2.2 LogBase

For implementing data storage and indexing, we start with *LogBase* [20], an open-source unordered log-store. Our CR-index is proposed as part of LogBase project. 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. When a record arrives, its attributes are divided into separate cells and appended to the file. 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 machines or log files.

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.

3.1 Logical View

Observational data in different scenarios might vary in many aspects, such as the number of variables and active sensors. Figure 2 shows an instance of a generic schema,

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 2: Schema logical view

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 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 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.

3.2 Physical View

Recall that LogBase splits records into attribute cells before appending them to the log files, with different groups in different files. Figure 3 shows the physical organization of the records in Figure 2, for the *Water* group.

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

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

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

Figure 3: Schema physical view

roughly ordered. Data disorder will be discussed in Section 4.3.

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. 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.

4.1 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 lightweight 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

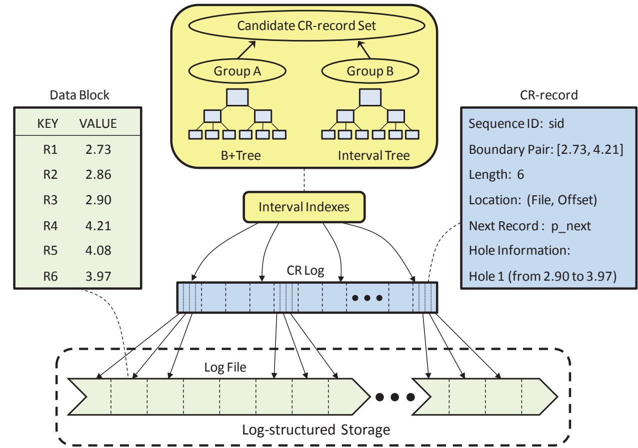


Figure 4: The CR-index structure

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 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.

4.1.1 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 4.5.3 will show that query performance degrades sub-linearly with increasing block length.

4.1.2 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 [7], 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 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 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

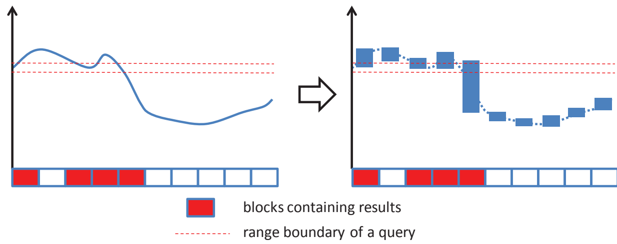


Figure 5: Abstraction of continuous data in blocks: original data on the left and block representation on the right.

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:

- **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 4.2.

4.1.3 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 [11]. 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

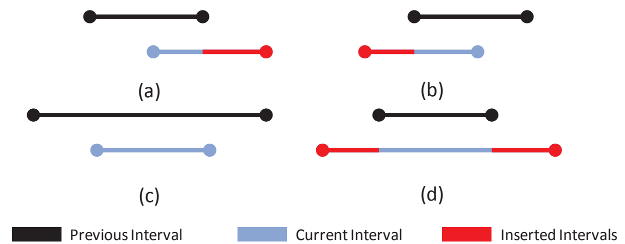


Figure 6: Cases of Delta Intervals

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 [19].

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.

4.2.1 Index with 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 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 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

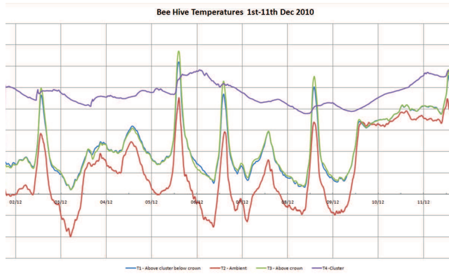


Figure 7: Multiple temperatures in a beehive

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)`

```
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).

4.2.2 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 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.

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.

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

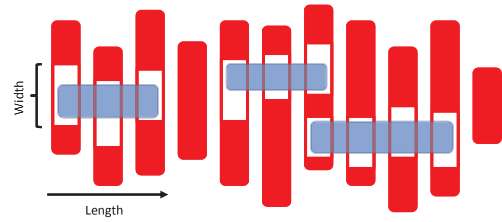


Figure 8: Holes in continuous ranges

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 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 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.

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 `detectHole(List CRrecords, Range range)`

```
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();
```

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.

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

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 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 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.

4.5 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.

4.5.1 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.

4.5.2 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:

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

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:

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

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.

4.5.3 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 4.5.2, 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.

4.6 Multi-Attribute Queries

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

4.6.1 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.

We note that there are inherent holes in multi-dimensional data, even when each dimension is ideally continuous. In Figure 9, we show two sources that are ideally continuous over time, but where there exist large holes.

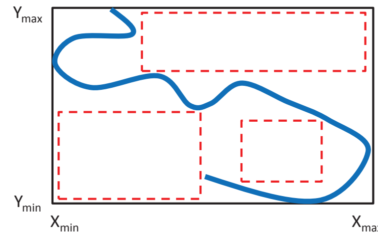


Figure 9: Inevitable holes (dashed-rectangles) in 2-dimensional continuous data sources

4.6.2 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-indexes 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 5.6.

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 [4] and LevelDB [5] 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).

5.1 Data Sets

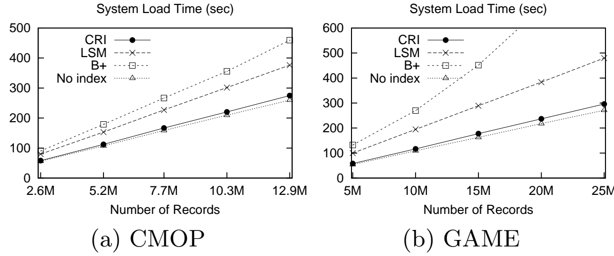
We use two real sensor datasets for our test, one from scientific observations, the other from an instrumented sports game. The first dataset is strongly continuous, while the second one has numerous holes.

5.1.1 CMOP Coastal Margin Data

This dataset contains coastal margin data collected from the CMOP [2] 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.

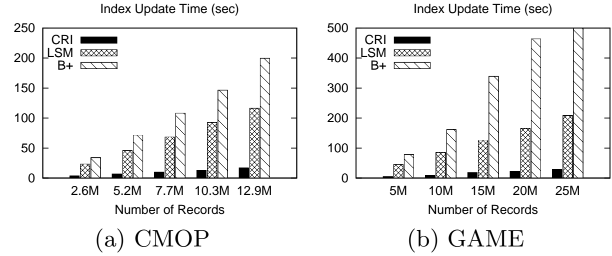
5.1.2 Real-time Soccer Game Data

The second dataset is from the DEBS 2013 Grand Challenge [3]. 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,



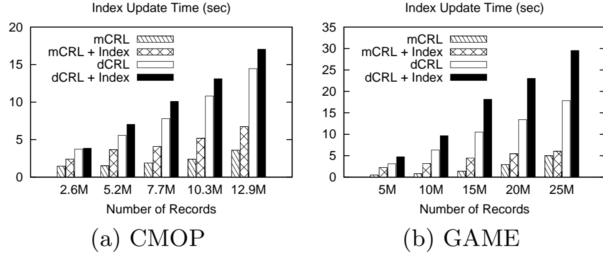
(a) CMOP (b) GAME

Figure 10: Overall system load time



(a) CMOP (b) GAME

Figure 11: Index maintenance cost



(a) CMOP (b) GAME

Figure 12: CR-index variants maintenance cost

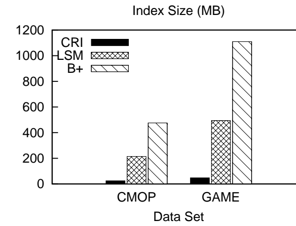


Figure 13: Index space consumption

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.

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* system.

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.

5.3 Write Performance

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

5.3.1 System-Load Time

Figure 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.

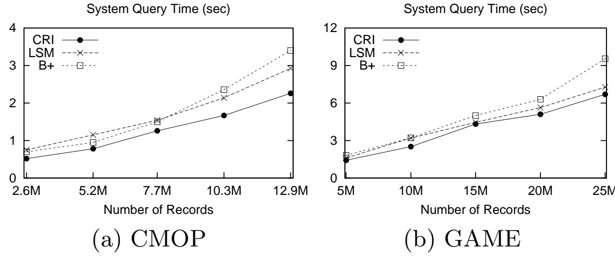
5.3.2 Index-Update Time

Figure 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.

Figure 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.

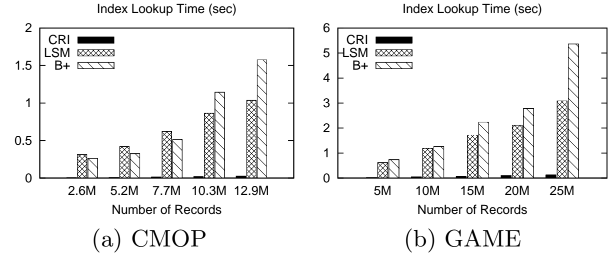
5.3.3 Index-Space Consumption

Figure 13 summarizes the disk-space consumption of different indexes. We only sum up the disk space, ignoring any



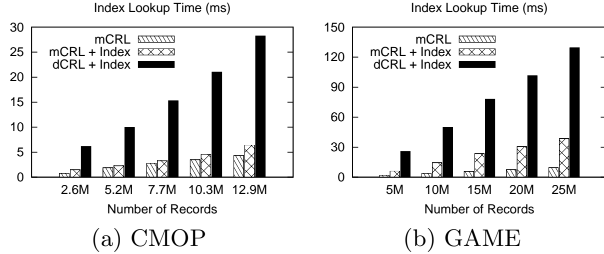
(a) CMOP (b) GAME

Figure 14: Overall system query response time



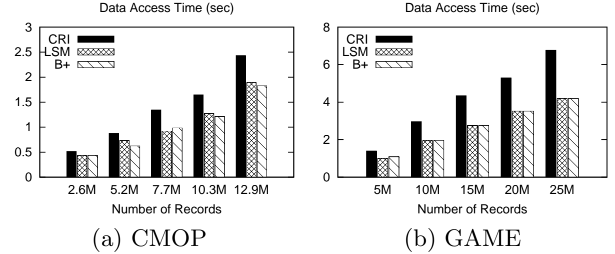
(a) CMOP (b) GAME

Figure 15: Index lookup phase cost



(a) CMOP (b) GAME

Figure 16: CR-index variants index lookup cost



(a) CMOP (b) GAME

Figure 17: Data access phase cost

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, 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.

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.

5.4.1 Query-Response Time

Figure 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 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.

5.4.2 Index-Lookup Cost versus Data-Access Cost

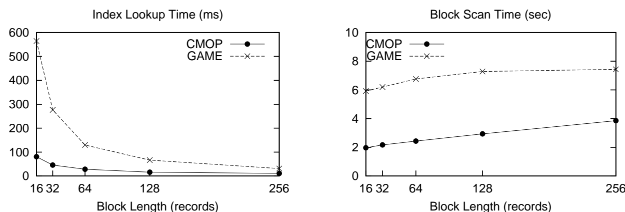
Figure 15 examines the index-lookup cost. As can be observed, the CR-index spends much less time than other approaches in this phase. The total cost is only 3-7% of that of the LSM-tree and 4-9% of the B$^+$-tree. These values are not surprising, since the number of entries in the CR-index is only 1/64 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.

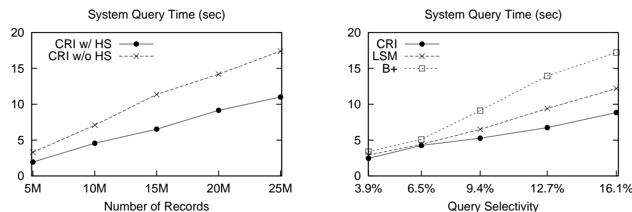
Table 1: Result Sequences in Datasets

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

Figure 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 (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 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 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 4.5.2, which is pessimistic. In real datasets, the number of seeks performed is much less than the theoretical bound.





(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 18: CR-index performance affected by different factors

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).

5.5.1 Block Length

Block length dominates both the number of generated CR-records and the block-scan cost. Figure 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 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 4.5 and supports our point that index size and update cost can be reduced significantly with only a moderate effect on query performance.

5.5.2 Effect of Hole Skipper

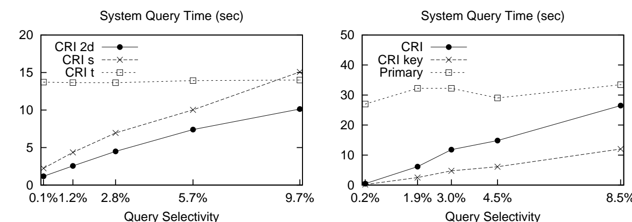
Potential discontinuities occur for many reasons. In the GAME dataset, the gaps in ball speed are produced by nature of the game: only one ball is in play alternately. Therefore, there are inherent gaps between the speed of the active ball and that of the other balls. Figure 18(c) shows the improvement 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 data because of its good continuity, and we omit the comparison here.

5.5.3 Query Selectivity

Figure 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.

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

5.6.1 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 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-indexes are available, the pre-filtering of CR-records prevents fetching most of the nonsatisfying data blocks and the consequent improvement is significant (CRI 2d). The proposed technique is extensible for more than 2 attributes.

5.6.2 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 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 4.6.2, with the improvement where the key has its own boundary-pair (CRI key), the overall response time could be much lower.

6. RELATED WORK

6.1 Storage Systems

Data and streaming warehouses are a major group of storage systems, some of which collect observational data.

DataDepot [12] is a tool for building and managing streaming warehouses in an RDBMS, providing fast data loading, automated view maintenance and data consistency control. SDAF [9], a data warehouse framework for sensor data, supports spatial queries over objects relating to location and time. A cloud-based sensor data warehouse method [13] was proposed on top of the distributed NoSQL database HBase [1]. It provides a simple key-value data model to manage sensor data in the column-oriented paradigm.

NoSQL systems, such as BigTable [8], HBase [1] and Cassandra [15], are widely used for distributed 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.

6.2 Index Structures

Index structures play an important role in supporting searches. For example, the classic B$^+$-tree [10] only needs a few I/Os for locating a search value, and it also supports efficient range queries. However, in large-scale, write-intensive applications, the required random I/Os for index updating are detrimental to write throughput.

In order to support indexing in write-intensive scenarios, a variety of log-structured indexes have been designed as alternatives to B$^+$-trees. An early log-structured index, the LSM-tree [17], makes use of exponential-sized subcomponents and merges them periodically using sequential I/Os. It significantly improves write throughput by avoiding random seeks during updates. However, its read performance is compromised, since all sub-components need to be consulted in an index access. Several variants of the LSM-tree have been proposed, such as the SSD-optimized FD-tree [16]. Many systems, such as LevelDB [5], HBase[1] and TokuDB [14], also incorporate log-structured indexes. Recently, a general purpose LSM-tree, bLSM [18], was proposed that gets near-optimal read performance by employing Bloom Filters [7]. The bLSM index outperforms B$^+$-trees in almost all scenarios. However, all these record-level indexes have to reside on disk with the data volumes we consider, because information is captured at per-item level.

7. CONCLUSION

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.

Acknowledgments

This work was supported by A*STAR project 1321202073. Maier was supported by NSF grant OCE-0424602 and Shaw Visiting Chair Professorship.

8. REFERENCES

[1] Apache HBase. `http://hbase.apache.org`.

[2] CMOP. `http://www.stccmop.org/`.

[3] DEBS 2013 Grand Challenge. `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails`.

[4] JDBM3. `https://github.com/jankotek/JDBM3`.

[5] LevelDB. `https://github.com/dain/leveldb`.

[6] netCDF. `http://www.unidata.ucar.edu/netcdf`.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[8] 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. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

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

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

[11] 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.

[12] 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.

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

[14] B. Kuszmaul. How tokudb fractal tree indexes work. *Technical report, TokuTek*, 2010.

[15] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[16] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, Sept. 2010.

[17] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[18] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *In Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 217–228, 2012.

[19] 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.

[20] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: a scalable log-structured database system in the cloud. *Proc. VLDB Endow.*, 5(10):1004–1015, June 2012.