Fast and Adaptive Indexing of Multi-Dimensional Observational Data

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

Sensing devices generate tremendous amounts of data 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 poses challenges to state-of-the-art indexing methods, as they must not only support efficient queries but also frequent updates. We propose here 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. This representation reduces the number of index entries and drastically reduces "over-coverage" by entries. Experimental results show that our approach handles real-world workloads gracefully, providing both low-overhead indexing and excellent query efficiency.

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.

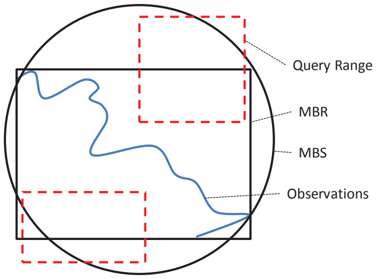


Figure 1: False positives from data sparseness.

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 [6, 7] that update the indexes in a batch manner, lowering per-item cost, and log-structured-merge trees [1, 19, 22] that incur only sequential I/Os for updates. In our previous work [26], 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 paper, we consider 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. 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 [12, 16] to represent a collection of data items. However, such representations cause "over-coverage", i.e., portions of indexed spaces that contain no actual points, as shown in Figure 1, where queries overlap with bounding objects, but not with observations. These structures force us to access false-positive entries. This issue becomes more severe as the dimensionality increases. Second, the write-intensive aspect limits resources

Proceedings of the VLDB Endowment, Vol. 9, No. 14
Copyright 2016 VLDB Endowment 2150-8097/16/10.

and possibilities to derive bounding objects with the least over-coverage. Prior work [15] 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.

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 lightweight for data ingestion, but still ensures query efficiency. Its graceful performance derives from the following three 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 by a segment in hyperspace. Thus we 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 paper 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. In 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 is continuously improved based on query-execution statistics.
- An extensive experimental evaluation on three 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 paper is organized as follows. Section 2 describes the targeted problem and key concepts in our design. Section 3 presents the overall framework. The details of bounding segment and index construction are discussed in Sections 4 and 5. We evaluate performance in Section 6. Sections 7 and 8 give related work and conclusions.

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.

2.1 Observational Data

Sensing devices are common data sources for analytic applications, especially scientific tasks. For example, coastal-margin observation [2] 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 observations of an entity (or environment) at different moments.

We rely on three characteristics of observational data that facilitate the indexing: **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 short 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.

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. The system should 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 within a d-dimensional query range r (or with $d' < d$ dimensions specified). We address the retrieval of complete query answers, where no approximate or lossy results are allowed.

This task is an typical indexing problem, but in a specific context. Thus, we have to address various issues with respect to the particularities and opportunities. For sequence analysis, qualified records should 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 the new data.

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 handle high insertion rates and concurrent query retrievals, as indexes could be refined during periods 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 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 [25]. As records are immediately appended into log files, separate WALs can be eliminated, and this aspect saves a large number of I/Os. Second, sequential scans return records in insertion order (which correlates with observation time). Furthermore, scans are highly efficient, as

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

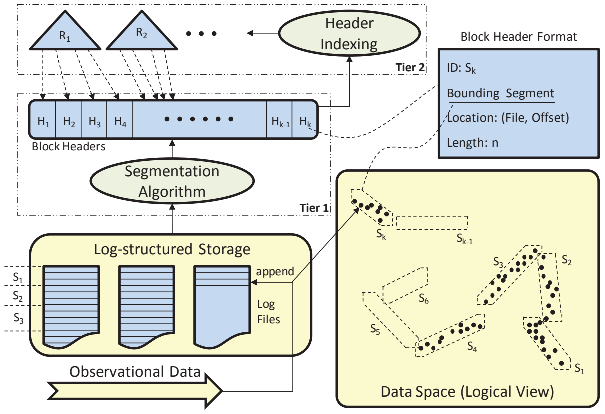


Figure 2: SICC index framework.

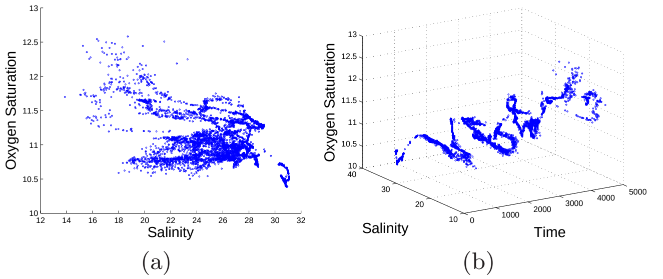
they avoid disk seeks and yield high bandwidth. Given a single disk-head seek ($\sim 5ms$), accessing one record ($\sim 10\mu s$) or hundreds of physically contiguous records have comparable costs. Therefore, a data-block scan have cost comparable to a single random-record access.

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 [14]. Hence there is no perfect induced clustering for all dimensions, while intrinsic clustering still provides reasonable effectiveness.

3. INDEX FRAMEWORK

In this section, we introduce our SICC framework for indexing observational data. The architecture is shown in Figure 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.

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, the total index size is quite small.



(a)

(b)

Figure 3: Observations from an estuary that contain salinity and oxygen saturation: (a) distribution across a time period; (b) distribution over time.

In the second tier, block headers are organized and indexed, so that we need test only some of the headers to get the complete query result. 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 block headers. Additionally, it also facilitates 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 4)
- During massive data ingestion, how do we generate bounding segments and index them without compromising system performance? (Section 5.1)
- If there are resources available after handling data ingestion and query requests, how can we refine the index for better query efficiency? (Section 5.2)

4. BOUNDING SEGMENTS

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.

4.1 Continuity among Observations

For a sequence of d-dimensional observations, the corresponding points form a continuous path in d-space, provided that the values in each dimension are continuous over time. This pattern can be observed in real-world data sources. Figure 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 3(b), these two dimensions are not completely correlated, as shown in Figure 3(a). No global correlation means that it is ineffective to directly apply dimension-reduction techniques [9] 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.

4.2 Bounding Segment Representation

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:

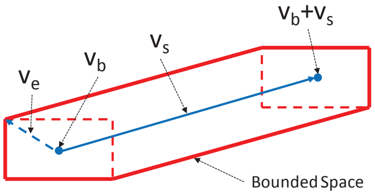


Figure 4: $2d$ bounding segment visualization.

- 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 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:

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

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 the axis is fixed, the extent v_e is still not unique. It is possible to increase ranges in some dimensions in order to reduce ranges in other dimensions. (See Figure 5.)

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 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.

4.3.1 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 subject to 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 [27], 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). CCIPCA is used because of its efficiency and simplicity for implementation, but other IPCA algorithms should also be applicable.

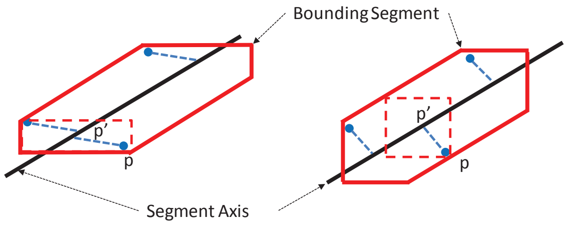


Figure 5: Different point projections onto the segment axis in 2-dimensions.

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

4.3.2 Segment Extent

The segment axis is the line passing through the mean of all points in the block, along the direction of v_s. After obtaining the segment axis, we can "project" points onto the axis. 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 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 the $mbb_e(x)$ function). Figure 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). 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.

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

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

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 segment v_b to $v_b + v_s$ against 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 $\cap_{1 \leq i \leq n} [l_i, u_i] \neq \varnothing$.

4.5 Calculating Segment Volume

To support online segmentation algorithms (Section 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. Calculating the volume with above formula only needs v_e and v_s, so we estimate these two values to get an approximation. For the extent v_e, another 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 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' is a good estimate of v_e. Similarly, 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 then estimated from the two endpoints that are at maximal distance to the origin in opposite directions.

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.

5.1 Index Construction

A SICC index can be easily constructed when data records are initially ingested into log-storage, hence these data are immediately ready for answering queries. 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.

5.1.1 Block Header

A block header contains two types of data: (1) that for 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 is the location information used to fetch the data block.

5.1.2 Eager Segmentation

We use an eager segmentation algorithm 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 called the *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: $\text{Res}_{point}(p) \sim \prod_{i=1}^{n} \bar{l}_{(i)}$. 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

Algorithm 1: Eager Segmentation (Record r)

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

$v_e + \frac{1}{2}\bar{l}$. As a result, the access frequency of block B is proportional to $\mathrm{Vol}(S'_B)$. (See Section 4.5 for calculating segment volume.) As the whole block will be fetched and scanned for each access, the expected number of accessed records is $|B| \cdot \mathrm{Vol}(S'_B)$.

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

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

If every block meets this constraint, the expected number of accessed records is less than $\mu \cdot r$. Algorithm 1 shows the eager segmentation procedure. We keep loading new records into the active block until it violates the constraint for a given μ, and then start a new block. Note that no assumption on data distribution or continuity is required. To adapt this analysis to arbitrary query distributions, all we need is a query histogram containing query frequencies and range sizes. We replace $\bar{l}$ with actual range sizes and weight the query-enlarged areas with the recorded frequencies.

The amplification factor μ controls the trade-off between index-construction cost and query efficiency. Though it is user-defined, its in-world meaning is straightforward, i.e, the number of records we can tolerate before getting a result. There is no clear optimal setting, as we can always reduce it so long as the system can handle increased overhead.

5.1.3 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 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.

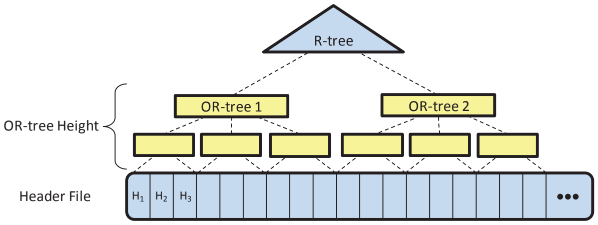


Figure 6: Indexing with OR-trees of height 2.

Intuitively, we might consider R-trees [12] 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. Since 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 strengths of R-trees and brute-force scans, we propose an R-tree variant that provides both write throughput and query efficiency. 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 like 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 6 illustrates this organization. As 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 a 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 algorithm [26].

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 distributions, 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.

5.2.1 Refinement Criterion

The performance of recent query requests reflects 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 caused its data block to be fetched.
- $C_{co-fetch}$: the number of queries that fetched both B's data block and its 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* 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.

5.2.2 Splitting and Exhaustive Segmentation

Recall that users provide a factor μ to limit expected read amplification (Section 5.1.2). 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" replaces a block's original header with two or more new headers, each covering a sub-block. Splitting costly blocks 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 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 the 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 predefined 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.

Algorithm 2: 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
List<Integer> $ret$;
Integer $n = w$.size;
Double $f[0 ... K][0 ... n]$;
set all elements in $f$ to $+\infty$;
$f[0][0] = 0$;
/* compute optimal cost for f[K][n]                    */
for $i = 0$ to $K$ do
    for $j = i$ to $n$-1 do
        if $f[i][j] == +\infty$ then continue;
        for $k = j + 1$ to $n$ do
            if $f[i+1][k] > f[i][j] + w(j+1, k)$ then
                $f[i+1][k] = f[i][j] + w(j+1, k)$;
/* backtrack segmented positions                       */
Integer $p = n$;
add $p$ into $ret$;
for $l = K$ downto 1 do
    find $p'$ with $f[l][p] = f[l-1][p'] + w(p'+1, p)$;
    add $p'$ into $ret$;
    $p = p'$;
reverse elements in $ret$;
/* $i$-th block is from record $ret[i] + 1$ to record
    $ret[i+1]$                                         */
return $ret$;
```

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:

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

No more than $K \cdot n$ terms of f are needed 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 limit split positions, e.g., at every 10 records, to reduce the search space. Algorithm 2 shows the core of exhaustive segmentation that takes pre-calculated $w(i,j)$'s and finds the optimal K-way split.

Although exhaustive segmentation is more expensive than eager segmentation, it happens only on write and minimizes subsequent query cost. If we do not need optimal performance, a cheaper alternative is multiple passes of eager segmentation. We choose an amplification factor μ' (via binary

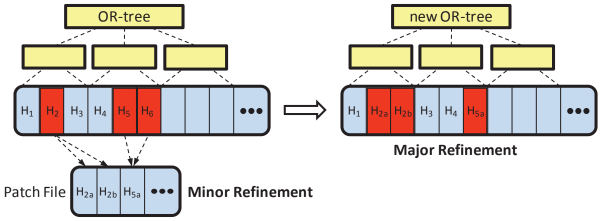


Figure 7: Minor refinement and major refinement.

search) before each pass, and terminate the iteration once we get a μ' that results in a K-way split.

5.2.3 *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 if 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:

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

Blocks are allowed to merge as long as 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 simple, as we only need to produce a new block header for the merged block to replace the old ones.

5.2.4 *Minor Refinement versus Major Refinement*

Based on available system resources and the number of accumulated refine operations, we can conduct a minor refinement or a 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 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 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 7 shows minor and major refinements. 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, the underlying data are unchanged, so we can support time-range queries and sequence analysis efficiently. Better performance might be obtained by reclustering the data and building a new SICC on top of them; we leave that possibility to future investigation.

It might happen that there are no resources for major compaction at times. As we cannot tolerate data loss by rejecting write requests, we have to stop or delay the compaction to preserve resources. This situation may affect query efficiency, possibly limiting query requests. However, we should assume resources for compaction become available some later time, otherwise the system is under-provisioned.

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.

6.1 Data Sets

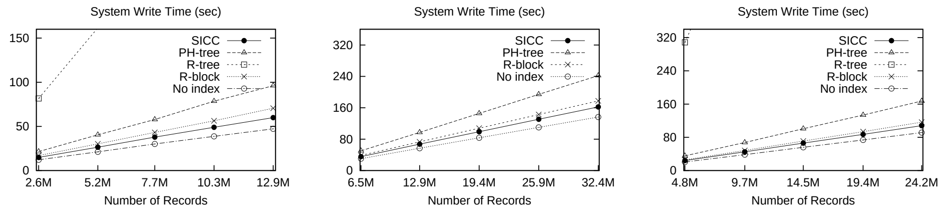
Experiments are conducted on three real datasets. All are observations collected from sensing devices, but with differing application scenarios and degrees of value continuity.

- **Coastal-Margin Observation (CMOP):** This dataset contains coastal margin observations collected from the CMOP [2] 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 subcomponents. The power consumption may vary significantly due to a 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 concatenate observations from all balls alternately in play to construct a continuous source.

6.2 Methods and Implementations

We compare the SICC index with three baseline methods, based on PH-trees [28] and R-trees [12]. All methods are implemented as secondary indexes, in which index entries contain record references to log-storage, and storage access fetches final results. All implementations are in Java, and integrated into a log-structured storage [25].

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



(a) CMOP

(b) POWER

(c) SOCC

Figure 8: Overall system-load time.

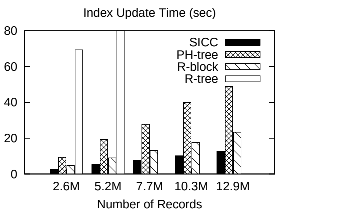


Figure 9: Index-maintenance cost. (CMOP)

Table 1: Average number of records in a block.

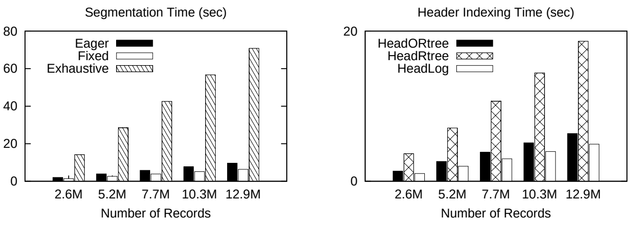
	$\mu = 1.2$	$\mu = 2$	$\mu = \mathbf{4}$	$\mu = 8$
CMOP	24.1	41.8	**84.7**	181.1
POWER	54.9	117.5	**207.8**	359.9
SOCC	152.6	325.5	**583.4**	908.2

- **SICC.** Our proposed index framework (Section 3) using bounding segments (Section 4). It applies eager segmentation and OR-tree header indexing (Section 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-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 primitive version of SICC that directly uses MBBs as bounding objects. A fixed number of consecutive records are first grouped as a block, and the block's MBB is then indexed in the R-tree. We include it to measure over-coverage.

To evaluate the effect of 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, we have: (a) **HeadLog**: no index for header log; (b) **HeadRtree**: an R-tree; and (c) **HeadORtree**: an OR-tree.

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 plus positions in two dimensions).[6] Each inserted record contains a string (its unique key), the indexed fields and other existing fields. The number of records ingested into storage are 13 million, 32 million and 24 million, respectively. The total disk space used by the storage system is 2.4GB, 5.6GB and 4.8GB, respectively.



(a) Segmentation

(b) Header Indexing

Figure 10: Decomposed SICC maintenance cost for segmentation algorithms and header indexes. (CMOP)

All range queries are pre-generated from a uniform distribution 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 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 average block-length quantifies the value-continuity of a dataset, i.e., the longer the better.

6.4 Write Overhead

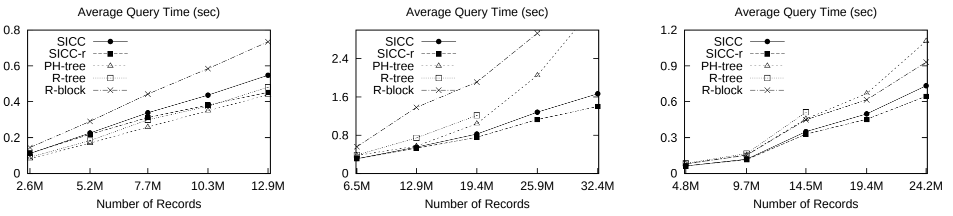
Index overhead is critical in write-intensive scenarios. This subsection evaluates that overhead, on both time and space consumption, for different approaches.

6.4.1 *System-Load Time*

Figure 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 split and index re-organization. We only succeeded in three rounds of insertions for the POWER and SOCC workloads due to its unaffordable cost. 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.

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

[6] Here we did not index time dimension, but it is a straightforward extension.



(a) CMOP (b) POWER (c) SOCC

Figure 11: Average query-response time.

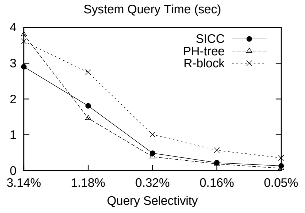


Figure 12: Query selectivity. (CMOP)

Table 2: Disk Consumption for indexes.

	SICC	R-block	R-tree	PH-tree
CMOP	16.0M	13.6M	739M	N/A
POWER	20.2M	17.6M	1.8G	N/A
SOCC	5.3M	4.3M	1.5G	N/A

6.4.2 Index-Maintenance Cost

By removing the loading time inside storage, Figure 9 shows the pure index cost more clearly with the CMOP workload. We can see that SICC has less than half the cost of any other approach. Though constructing a bounding segment is a bit costlier than an MBB, with the OR-trees, SICC outperforms R-block. On closer examination of the results, we find that about three-quarters of the 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.

6.4.3 Maintenance Cost of SICC Components

We further decompose SICC's cost into segmentation and header-indexing costs. Figure 10 compares design choices on the CMOP workload. As Figure 10(a) shows, 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 contrast, 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, it is suitable for continuous refinement due to its effectiveness.

Figure 10(b), also on the CMOP workload, presents the choices for indexing block headers, i.e., the header log. The result shows that indexing the header log with an OR-tree (HeadORtree) is nearly as efficient as just maintaining a header log (HeadLog) alone, due to the fast construction of append-only sub-trees. However, the overhead of constructing an entire R-tree (HeadRtree) is considerable.

6.4.4 Index-Space Consumption

Table 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., a segment needs three arrays (v_b,v_s,v_e) while MBB needs just two arrays (lower and upper bounds). We see 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.

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.

6.5.1 Query Execution Time

Figure 11 presents query-response time using different indexes. R-block cannot achieve satisfactory efficiency, due to the over-coverage introduced by MBRs. Its performance is sensitive to workload distribution. In contrast, SICC has comparable efficiency to record-level indexes at small scale, and outperforms them as data grows. 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. As the data volume keeps growing, successive disk accesses are more spread out, raising the disk-seek overhead and limiting scalability. This issue can be resolved by collecting and sorting disk-offset before actually accessing the disk, but that requires more memory resources, or an external sort in the worst case. When dealing with highly continuous data, scans with large 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.

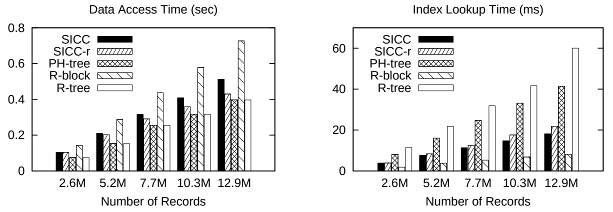
6.5.2 Data-Access Cost

Figure 13(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.

Figure 14(a) presents SICC data-access cost under different segmentation algorithms. Exhaustive segmentation is expected to be superior. With a reasonable amplification factor μ, eager segmentation is also efficient. Considering its low overhead, the performance is more than satisfactory. Since fixed-length segmentation does not consider data distribution, it rarely achieves good pruning effectiveness.

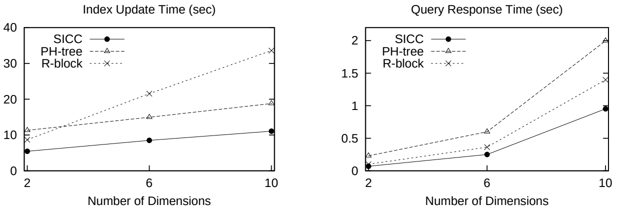
6.5.3 Index-Lookup Cost

Figure 13(b) presents index-lookup cost. Among all methods, R-block has the lowest cost, as all block headers are well



(a) Data Access (b) Index Lookup

Figure 13: Decomposed query cost. (CMOP)



(a) Maintenance Overhead (b) Query Efficiency

Figure 15: Index performance with increasing number of dimensions. (Synthetic Data)

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 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.

Figure 14(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, while its query efficiency is comparable to the R-tree's.

6.6 Exploratory Study

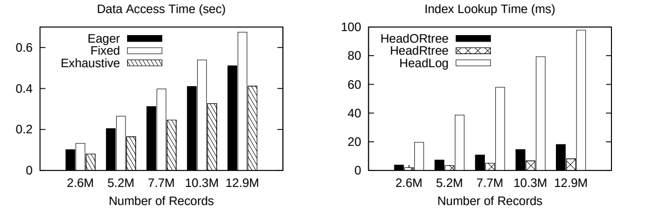
In this subsection, we explore the effect of bounding objects, query selectivity and dimensionality.

6.6.1 *Bounding Segments vs. Bounding Boxes*

Table 3 shows the huge gap between bounding segments and bounding boxes, in terms of bounded volume and estimated query cost. Bounding segments reduce over-coverage by an order of magnitude. Their query costs are also less than half those of bounding boxes. Table 3 also confirms the effectiveness of different segmentations. The reduction of over-coverage is also sensitive to workload continuity and distribution. We observed that the volume reductions in bounding segments are about 12.2× for CMOP, 17.5× for POWER and 5.8× for SOCC, relative to R-block.

6.6.2 *Query Selectivity*

Figure 12 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, has an advantage over SICC. Even for true-positive blocks, SICC may get only a small fraction of results from each block. However, at low selectivities, false-positive blocks factor less, and true-positive block are 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.



(a) Data Access (b) Index Lookup

Figure 14: Decomposed SICC query cost for segmentation algorithms and header indexes. (CMOP)

Table 3: Volume and Access-Cost in the CMOP dataset (baseline: fixed-length segmentation)

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

6.6.3 *Dimensionality*

In high dimension, SICC can capture correlations from the PCA component to achieve good pruning effectiveness. For a fair comparison, we generate a synthetic dataset, in which each dimension is independent, to minimize correlations. Figure 15 illustrates the results. In each test, we evaluate the overhead of indexing 5 million records, and the average query time of 100 random queries. Figure 15(a) shows that the maintenance overhead scales 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 in the number of dimensions. Figure 15(b) shows that the SICC also achieves the best query scalability among the approaches.

7. RELATED WORK

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), as examined in several surveys [10, 17]. In PAMs, the whole space is divided into subspaces, either flatly, e.g., grid files [18], or hierarchically, e.g., kD-trees [4] and quadtrees [8]. Recently, a space-efficient PH-tree [28] 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 and its variants [3, 12, 5]. In SAMs, bounding objects are used to approximate the objects and bound the subtree spaces. Such representations may incur over-coverage, which affects query efficiency. An earlier work [15] uses polygons to reduce over-coverage, but is costly to construct.

Index-maintenance cost is a bottleneck for write-intensive applications. Various methods have been proposed to address this issue. Bulk insertions [6, 7] buffer a number of records and insert the whole batch at once. They reduce random writes, at the expense of insertion latency. LSM-trees [19, 22] eliminate random writes by merging exponential-sized sub-components using sequential I/Os. A generalization of LSM-trees [1] allows an index to be converted to append-only. Adaptive indexing [11, 13] gradually construct indexes during queries. They do not target reducing index cost, but rather the downtime before data can be queried, thus are mainly used in read-heavy data warehouses. All the methods above are at the record-level, which is limiting in write-intensive scenarios. Another direction is to exploit partial order and local clustering in unclustered data [23, 26]. Our previous work [26] focuses on single-dimensional data and constructs compact indexes on local sequences.

Trajectory indexing over position sequences of moving objects is another related area. Trajectory indexes stores positions from multiple trajectories in a single structure while preserving trajectory-level properties [21]. They are tailored for data in 2 or 3-D spaces with special assumptions [20]. This case is different from observational data where we have a single long-life entity with many dimensions. In general, trajectory indexing focuses on the time dimension, i.e., find all trajectories at a time-point, while we focus on space dimensions, i.e., find all sub-sequences within a value-range.

Data-series and time-series analyses [24, 29] also index multiple points, i.e., points in a series, in one index entry, but are fundamentally different. In time-series analysis, each series is considered as a whole, thus indexing points together is to facilitate similarity or kNN search. However, in SICC, indexing points together is to reduce maintenance cost, and points are still queried independently.

8. CONCLUSION

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 "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 SICC is an order of magnitude faster to construct than conventional record-level indexes, while it preserves comparable query efficiency.

Acknowledgments

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

9. REFERENCES

[1] 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), June 2014.

[2] A. M. Baptista et al. Infrastructure for collaborative science and societal applications in the columbia river estuary. *Frontiers of Earth Science*, 2015.

[3] 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.

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

[5] 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*, 1996.

[6] 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*, 1998.

[7] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: a generalized R-tree bulk-insertion strategy. *Proc. of Int. Symp. on SSD*, 1999.

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

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

[10] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2), June 1998.

[11] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proc. of Int. Conf. on EDBT*. ACM, 2010.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2), June 1984.

[13] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, volume 3, 2007.

[14] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2), May 1990.

[15] H. V. Jagadish. Spatial search with polyhedra. *Proc. of Int. Conf. on Data Engineering*, 1990.

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

[17] K. Markov, K. Ivanova, I. Mitov, and S. Karastanev. Advance of the access methods. *Information Technologies and Knowledge*, 2(2), 2008.

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

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

[20] D. Pfoser. Indexing the trajectories of moving objects. *IEEE Data Eng. Bull.*, 25(2), 2002.

[21] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *Proc. of VLDB*, 2000.

[22] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proc. of ACM SIGMOD*, 2012.

[23] L. Sidirourgos and M. Kersten. Column imprints: a secondary index structure. *Proc. of ACM SIGMOD*, 2013.

[24] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *SIGKDD*, 2003.

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

[26] S. Wang, D. Maier, and B. C. Ooi. Lightweight indexing of observational data in log-structured storage. *Proc. of VLDB Endow.*, 7(7), Mar. 2014.

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

[28] T. Zäschke, C. Zimmerli, and M. C. Norrie. The PH-tree: a space-efficient storage structure and multi-dimensional index. *Proc. of ACM SIGMOD*, 2014.

[29] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *Proc. of ACM SIGMOD*. ACM, 2014.