RC23426 (W0411-101) November 10, 2004
Computer Science

IBM Research Report

Interval Query Indexing for Efficient Stream Processing

Kun-Lung Wu, Shyh-Kwei Chen, Philip S. Yu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Interval Query Indexing for Efficient Stream Processing

Kun-Lung Wu, Shyh-Kwei Chen and Philip S. Yu
IBM T.J. Watson Research Center
Hawthorne, NY 10532
{klwu, skchen, psyu}@us.ibm.com

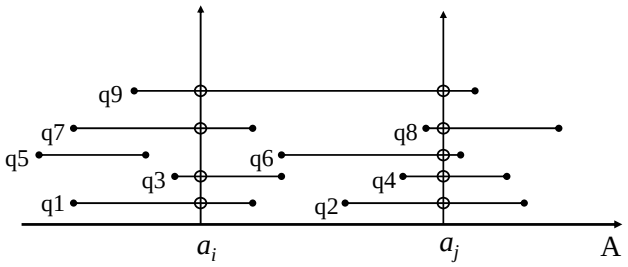
Abstract

A large number of continual range queries can be issued against a data stream. Usually, a main memory-based query index with a small storage cost and a fast search time is needed, especially if the stream is rapid. In this paper, we present a CEI-based query index that meets both criteria for efficient processing of continual interval queries in a streaming environment. This new query index is centered around a set of predefined virtual *containment-encoded intervals*, or CEIs. The CEIs are used to first decompose query intervals and then perform efficient search operations. The CEIs are defined and labeled such that containment relationships among them are encoded in their IDs. The containment encoding makes decomposition and search operations efficient because integer additions and logical shifts can be used to carry out most of the operations. Simulations are conducted to evaluate the effectiveness of the CEI-based query index and to compare it with alternative approaches. The results show that the CEI-based query index significantly outperforms existing approaches in terms of both storage cost and search time.

1 Introduction

Many data stream applications have been recognized [1, 9]. For example, financial applications, network monitoring, security, telecommunications data management, sensor networks and other applications are best modeled as data streams. In a stream model, individual data items can be relational tuples with well-defined attributes, e.g., network measurements, call records, meta data records, web page visits, sensor readings, and so on. These data items arrive in the form of streams continually and perhaps rapidly.

In order to monitor a data stream and take proper actions, if needed, a large number of range queries or filtering predicates can be created and evaluated continually against each data item in the incoming stream. For example, in a financial stream application, various continual range queries can be created to monitor the prices of stocks, bonds, or interest rates. In a sensor network stream application, continual range queries can be used to monitor the temperatures, flows of traffic or other readings.



Intervals containing a_i: q1, q3, q7, q9
Intervals containing a_j: q2, q4, q6, q8, q9

Figure 1: Example of identifying intervals that contain a data value.

One approach to quickly evaluating these continual range queries is to use a query index. Each data value in the incoming stream is used to search the query index and identify those relevant queries that contain the data value. Though conceptually simple, it is quite challenging to design such an interval query index in a streaming environment, especially when the stream is rapid. The interval query index is preferably main memory-based and it must have two important properties: low storage cost and excellent search performance. Low storage cost is important so that the entire query index can be loaded into main memory. Excellent search performance is critical so that the query index can handle a rapid stream.

The goal of interval query indexing is to help identify, from a set of query intervals, those that contain an incoming data value. This was referred to as a *stabbing query problem* [10], i.e., finding all the intervals that are stabbed by the data value. Fig. 1 shows an example of a stabbing query problem, where intervals are drawn horizontally. The intervals that intersect with the vertical line drawn at a data value are those that contain that data value. For instance, query intervals q_1, q_3, q_7 and q_9 contain data value a_i while q_2, q_4, q_6, q_8 and q_9 contain data value a_j.

There have been existing approaches for interval indexing to help solve stabbing queries, such as segment trees, interval trees [10], R-trees [4], interval binary search trees (IBS-trees) [6] and interval skip lists (IS-lists) [7]. However, they are generally not suitable for streams. Segment trees and interval trees generally work well in a static environment, but are not adequate when it is necessary to dynamically add or delete intervals. Originally designed to handle spatial data, such as rectangles, R-trees can be used to index intervals. However, as indicated in [7], when there is heavy overlap among intervals, search performance degenerates quickly. Moreover, R-trees are primarily disk-based indexes.

IBS-trees and IS-lists were designed for main memory-based interval indexing [6, 7]. They were the first dynamic approaches that can handle a large number of overlapping intervals. As with other dynamic search trees, IBS-trees and IS-lists require $O(\log(n))$ search time and $O(n \log(n))$ storage cost, where n is the total number of intervals. Moreover, as pointed out in [7], in order to achieve the $O(\log(n))$ search time, a complex "adjustment" of the index structure is needed after an insertion or deletion. The adjustment is needed to re-balance the index structure. More importantly, the adjustment makes it difficult to reliably implement the algorithms in practice. Previous studies [7] indicated that IS-lists perform better than IBS-trees and are easier to implement, even though dynamic adjustments of the interval skip lists are still needed.

There are other spatial indexing methods that can be used to handle one dimensional intervals [8, 3, 10]. However, most of them are designed specifically for multidimensional spatial objects. In addition, they tend to be secondary storage-based indexes. In contrast, a main memory-based index is usually preferred for stream processing, especially if the number intervals indexed is large.

All the aforementioned query indexes are *direct indexing* approach. The query index is built directly with the query boundaries and search is conducted by comparing a data value with the query boundaries maintained in the index. Even with search time of $O(\log(n))$, such as the IS-lists approach, the search time may not be fast enough to handle a rapid stream, especially if n is large.

In contrast, we propose an *indirect indexing* approach. It is based on the concept of *virtual constructs* (VCs). A set of virtual constructs is predefined and properly labeled. Each query is decomposed into one or more VCs and the query ID is inserted into the ID lists associated with the decomposed VCs. Search is conducted indirectly via the VCs. No comparison of a data value with any of the query boundaries is needed during a search. Because of the query decomposition, the search result is contained in the ID lists associated with the covering VCs of a data value. We define VCs in such a way that there is only a small and fixed number of VCs that can cover any data value. As a result, search time is independent of n, in terms of locating the covering VCs.

Fig. 2 shows an example of a VC-based query index. We show 5 virtual construct intervals, $v1, \cdots, v5$, which are used to decompose query intervals, $Q2, Q3$ and $Q4$. After decomposition, query IDs are inserted into the associated ID lists. To perform a search on data value x, we first

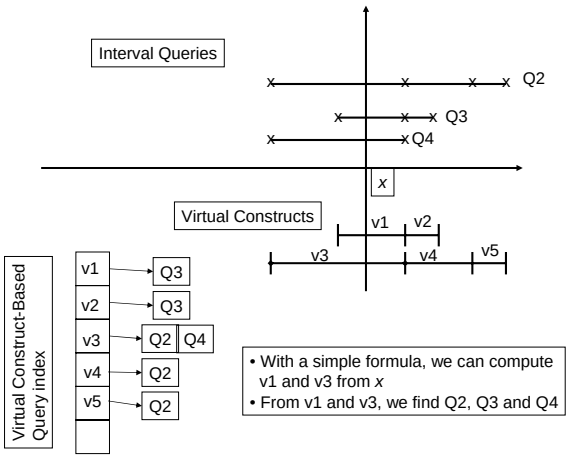


Figure 2: Example of a VC-based query index.

find the covering VCs for x. In this case, they are $v1$ and $v3$. The search result is contained in the ID lists associated with $v1$ and $v3$. From the VC-based query index, we find $Q2, Q3$ and $Q4$.

Though conceptually simple, there are challenges in a VC-based query index. These include the definition and proper labeling of the virtual constructs, the decomposition of query intervals and the computation of the covering VCs that contain any data value during a search. These issues must be carefully addressed in order to have a query index that has both low storage cost and fast search time.

In this paper, we present a CEI-based query index that addresses all these challenging issues. It is memory-based and has the properties of low storage cost and fast search operation, particularly suitable for stream processing. Unlike the IS-lists approach, no re-balancing is required. Hence, it is much easier to implement than IS-lists. The new query index is centered around a set of predefined virtual *containment-encoded intervals*, or CEIs. We partition the range of interest of a numerical attribute into multiple segments. For each segment, a set of virtual CEIs are predefined. The CEIs are defined and labeled such that the containment relationships among them are encoded in their IDs. The containment encoding makes decomposition and search operations efficient because integer additions and logical shifts can be used to carry out most of the operations. Simulation studies show that the CEI-based query index outperforms the IS-lists approach in terms of both storage cost and search time.

The paper is organized as follows. Section 2 describes alternative virtual construct intervals and their associated challenges. Section 3 describes the CEI-based query index. The containment-encoded intervals, the decomposition of a query interval into one or more CEIs, the search

algorithm and other system design issues are presented. Section 4 presents performance studies. We evaluate and compare the CEI-based query index with alternative indexes. Finally, Section 5 summarizes the paper.

2 Virtual construct-based query indexes

2.1 System model

There are two kinds of intervals discussed in this paper: (a) query or predicate intervals and (b) virtual construct intervals. Query intervals are real. They are defined by users over a numerical attribute to monitor a data stream. On the other hand, virtual construct intervals are created to decompose query intervals and perform fast search operations.

We focus on simple queries, where each query is specified with a single interval predicate on a numerical attribute. However, the result of this paper is applicable to complex queries, where each query is specified with a conjunction of multiple interval predicates on various attributes. For example, a CEI-based query index can be maintained for each attribute in a two-phase algorithm involving complex queries, such as the ones presented in [2, 12]. In the first phase, search operations against the CEI-based indexes are performed at individual attributes. The matched results are then merged in the second phase.

We assume that all query intervals are specified on the same attribute. The attribute can be of integer or non-integer type. Query intervals have two integer endpoints with an inclusive left endpoint and an exclusive right endpoint. [1] Namely, they can be specified as $[a, b)$, where a and b are integers and $a < b$. However, data values used for search can be any real numbers. Other design issues will be discussed on Section 3.4 for cases where both endpoints are inclusive or there is only one endpoint.

2.2 Alternative VCs and their challenges

In a VC-based query index, a query ID q is replicated by d times, where d is the number of VCs used in the decomposition of q. For each VC, a pointer is needed to maintain the associated ID list. Hence, the storage cost of a VC-based query index depends on two parameters: the total number of VCs defined, N, and the total number of query IDs inserted due to decomposition,

[1] If the endpoints of a query interval are not integers, we can expand them to the nearest integers. The expanded interval is then decomposed and indexed. The VC-based indexing is still effective in identifying a set of candidate queries. Nevertheless, a final checking is needed to determine if the candidate queries indeed contain the data value.

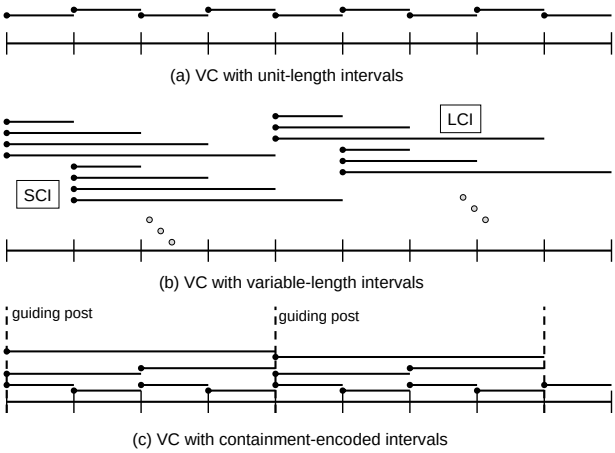


Figure 3: Example of three alternative virtual construct intervals.

Table 1: Comparison of three alternative virtual construct intervals.

VC scheme	Total VCs defined (N)	Total covering VCs	labeling scheme	decomposition algorithm	storage cost	search time
UI	r	1	obvious	obvious	high	fast
LCI	$(1+\log(L))r$	$2L-1$	simple	simple	moderate	slow
CEI	$2r - r/L$	$(\log(L)+1)$	simple, but non-obvious	simple, but non-obvious	low	fast

D. Assume C is the number of VCs that can possibly cover a data value. The search time of a VC-based query index depends on C. Different VC-based indexes have different N's, D's and C's, resulting in different storage costs and search times.

Fig. 3 shows three kinds of alternative virtual constructs and Table 1 provides a summarized comparison of the three. The first one is the simplest approach. It uses *unit-length intervals*, or UI (see Fig. 3(a)). The total number of VCs defined is r, where r is the scope or range of the attribute. With UIs, labeling and decomposition are obvious. We simply label the unit-length intervals sequentially. The decomposition is by partitioning a query interval into one or more unit-length UIs. Search time is extremely fast because there is only a single VC that can cover any given data value x. However, the storage cost can be high. This is because we need to use $\bar{w}$ UIs to decompose a query interval on average, where $\bar{w}$ is the mean length of a query interval. Hence, the cost for storing the ID lists becomes large. If $\bar{w}$ is large, the UI-based scheme cannot be scaled to a large n because the storage cost can be too large to fit into main memory.

The second kind of VCs aims to reduce the storage cost by using multiple variable-length VCs. A larger-sized VC can be used for decomposition, reducing the number of VCs in the decomposition. Since a query ID is inserted into all the decomposed VCs, the cost for the ID lists is reduced. In its simplest form, we define for each integer attribute value a set of L virtual intervals with length of $1, 2, \cdots, L$, where L is the maximal length of a VC. These L VCs all start at the same position but ends at different positions. We call it a simple construct interval (SCI) approach (see Fig. 3(b)). The total number of VCs defined is rL. More VCs increase the storage cost for maintaining the pointers to the ID lists, offsetting the benefit of low storage cost achieved due to fewer decomposed VCs.

A more efficient variable-length VC approach would be a logarithmic construct interval (LCI) approach (see Fig. 3(b)). In LCI, the total number of VCs defined is $(1 + \log(L))r$ because we use $1 + \log(L)$ variable-length VCs for each integer attribute value. For the rest of the paper, we use LCI as the representative for the variable-length interval approach. Both labeling and decomposition are simple under the LCI-based approach. One can sequentially label the $(1 + \log(L))$ VCs with the same left endpoints. The decomposition tries to use as few LCIs as possible by always choosing the largest available LCI for decomposition beginning from the left endpoint of the query interval. It can be proved that the number of LCIs that can cover any data value x is $2L - 1$ [11]. The storage cost is moderate for LCI. However, the search time becomes moderate or slow, especially if a large L is chosen in order to reduce the storage cost.

In this paper, we describe in detail a third alternative approach using containment-encoded intervals (see Fig. 3(c)). It has the advantages of both low storage cost and fast search time. These advantages make CEI-based query index particularly suitable for a streaming environment. In CEI, the range of attribute values are partitioned into segments of length L, where L is an integer that is a power of 2. Within each segment, we define $2L-1$ containment-encoded intervals. As a result, the total number of VCs defined is only $2r - r/L$, and the number of covering VCs for any data value x is $1+\log(L)$. However, the labeling scheme and decomposition algorithm are non-obvious. They must be labeled carefully so that both decomposition and search operations can be carried out efficiently. In the rest of the paper, we will present our efficient solutions which are also easy to implement.

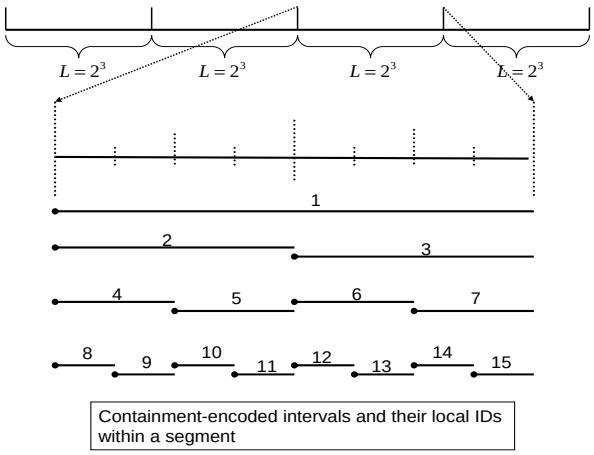


Figure 4: Example of containment-encoded intervals.

3 CEI-based query indexing

3.1 Containment-encoded intervals

Fig. 4 shows an example of containment-encoded intervals and their local ID labeling. Assume the range of interest of a numerical attribute A is $[0, r)$. First, we partition r into r/L segments of length L, denoted as S_i, where $i = 0, 1, \cdots, (r/L - 1)$, $L = 2^k$, and k is an integer. Here, we assume r is a multiple of L. If not, we can simply expand it and make it so without impacting the system performance. Segment S_i contains all the attribute values in $[iL, (i+1)L)$. Namely, $S_i = \{x | x \in A \wedge iL \leq x < (i+1)L\}$. Segment boundaries can be treated as guiding posts. Then, we define $2L - 1$ CEIs for each segment as follows: (a) Define 1 CEI of length L, containing the entire segment; (b) Define 2 CEIs of length $L/2$ by partitioning the segment; (c) Define 4 CEIs of length $L/4$ by partitioning the segment; (d) Continue the process until L CEIs of unit length ($L/L = 1$) are defined. For example, there are 1 CEI of length 8, 2 CEIs of length 4, 4 CEIs of length 2 and 8 CEIs of length 1 in Fig. 4.

Property 1 *The total number of CEIs in a segment is* $\sum_{i=0}^{i=k} 2^i = 2L - 1$, *where* $L = 2^k$.

These $2L - 1$ CEIs are defined to have containment relationships among them in a special way. Every unit-length CEI is contained by a CEI of size 2, which is in turn contained by a CEI of size 4, which is in turn contained by a CEI of size 8, ... and so on.

Now, we describe the labeling of CEIs with containment encoding. The ID of a CEI has two parts: the segment ID and the local ID. For each segment, we assign $1, 2, \cdots, 2L - 1$ to each

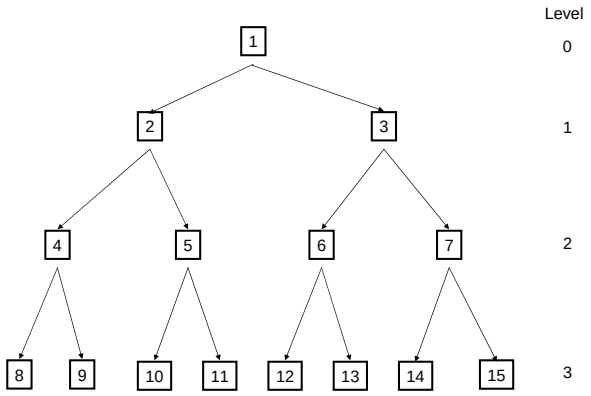


Figure 5: Example of a perfect binary tree and its labeling.

of the $2L-1$ CEIs as their local IDs. The local ID assignment follows the labeling of a perfect binary tree (see Fig. 5.) Local ID 1 is assigned to the CEI with length of L. Local ID 2 is assigned to the left CEI of length $L/2$ while local ID 3 is assigned to the right CEI of length $L/2$. Fig. 4 shows the assignment of local IDs to CEIs within a segment.

The global unique ID for a CEI in segment S_i, where $i = 0, 1, \cdots, (r/L) - 1$, is simply computed as $l + 2iL$, where l is the local ID. Note that we assign $2L$ local IDs for every segment, even though there are only $2L-1$ CEIs. Local ID 0 is not used.

With local IDs, all the CEIs in a segment can be organized as a perfect binary tree. The CEI with local ID 1 and length L is the root node, which contains two children CEIs of length $L/2$ whose local IDs are 2 and 3, respectively. The leaf CEIs have unit length and local IDs from L to $2L-1$. Figs. 4 and 5 show that, for $L = 8$, the leaf CEIs have local IDs from 8 to 15. The labeling of CEIs based on a perfect binary tree has many nice properties. These properties make possible efficient decomposition and search for the CEI-based query index. Due to containment encoding, almost all of the decomposition and search operations can be efficiently carried out via logical shifts and integer additions.

Property 2 *There are $k+1$ levels in the perfect binary tree formed with all the CEIs defined within a segment. Level i has 2^i CEIs, each with length $L/2^i$, for $i = 0, 1, \cdots, k$. The local IDs of CEIs at level i starts from 2^i to $2^{i+1} - 1$.*

Property 3 *Any non-root CEI is fully contained by its parent.*

Property 4 *For any non-root CEI with a local ID l, the local ID of its parent can be computed by $\lfloor l/2 \rfloor$, or a logical right shift by 1 bit of the binary representation of l.*

Property 5 *For any non-leaf CEI with a local ID l, the local IDs of its children are $2l$ and $2l+1$, respectively. Note that $2l$ can be computed by a logical left shift by 1 bit of the binary representation of l.*

Property 6 *For any data value $x \in A$, there are exactly $k+1$ CEIs that contain it because there are $k+1$ levels of non-overlapping CEIs in a segment.*

Property 7 *For any data value $x \in A$, the global ID of the unit-length CEI that contains x is $l + 2iL$, where $i = \lfloor \lfloor x \rfloor / L \rfloor$, and $l = \lfloor x \rfloor - iL + L$.*

From Property 2, the unit-length CEIs in any segment have local IDs from $2^k = L$ to $2L-1$. Hence, the local ID of the CEI containing x is $\lfloor x \rfloor - iL + L$, where $i = \lfloor \lfloor x \rfloor / L \rfloor$. Note that i can also be computed as $\lfloor x/L \rfloor$. However, it would involve a complex floating point division because x is a non-integer. In contrast, $\lfloor \lfloor x \rfloor / L \rfloor$ can be computed by first an integer conversion from a floating point number, $\lfloor x \rfloor$, and then a logical right shift by k bits because $L = 2^k$.

3.2 Query insertion and deletion

To insert a query interval, it is first decomposed into one or more CEIs, then its ID is inserted into the ID lists associated with the decomposed CEIs. The CEI-based query index maintains a set of query ID lists, one for each CEI. Similar to the inverted lists typically used in information retrieval, the query ID list associated with a CEI maintains all the query intervals that contain that CEI.

The decomposition algorithm is conceptually simple. It involves the concepts of guiding posts and remnant intervals. We use segment boundaries as guiding posts. For each query $q : [a, b)$, we define two boundary guiding posts: P_L and P_R. The left boundary guiding post $P_L = L\lceil a/L \rceil$ is the leftmost guiding post that is greater than or equal to a. The right boundary guiding post $P_R = L\lfloor b/L \rfloor$ is the rightmost guiding post that is smaller than or equal to b. These two boundary guiding posts are used to compute two remnant intervals, if any. The left remnant interval, R_L, is defined as $[a, P_L)$. If P_L equals a, then R_L is empty. The right remnant interval, R_R, is defined as $[P_R, b)$. If P_R equals b, then R_R is empty.

Fig. 6 shows the pseudo code for the decomposition algorithm. First, we compute the boundary guiding posts, P_L and P_R, and the remnant intervals, R_L and R_R. Then, we check if P_L is greater than P_R. If yes, it means that q completely lies between 2 guiding posts without

intersecting any. In this case, we call function **BestPartition** to decompose q. On the other hand, if $P_L < P_R$, then we insert the query ID q into the largest CEI of each segment that is completely contained by q between the two boundary guiding posts. This process repeats for $(P_R - P_L)/L$ times. After that, we use **BestPartition** to further decompose the two remnant intervals, if there is any.

There are more than one way to decompose an interval residing within a segment, or between two consecutive guiding posts. For example, we can always decompose it using consecutive unit-length CEIs. As a query ID is inserted into all the ID lists associated with the decomposed CEIs, using more CEIs means more storage cost for the index. The goal of **BestPartition** algorithm is thus to minimize the number of CEIs used in a decomposition. In other words, we want to use maximal-sized CEIs, if possible, for decomposition. **BestPartition** is based on the following property, which is derived from CEI labeling and can be easily observed from Fig 4.

Property 8 *Two CEIs at level i with consecutive local IDs can be replaced by a CEI at level $i-1$ of double length only if the smaller ID of the two CEIs at level i is even.*

BestPartition can be viewed as a *merging* process, starting from the leftmost unit-length CEI at level k towards the root CEI at level 0. For any interval of length m, where $0 < m < L$, which lies between two consecutive guiding posts, it can be decomposed into m unit-length CEIs at level k. These unit-length CEIs have consecutive local IDs. The algorithm initializes a remnant interval R to be $[s, e)$ which is the interval to be decomposed. It starts with the local ID of the leftmost unit-length CEI. If the ID is an even number, then we try to replace two children CEIs with their parent CEI. Namely, we double the size of the CEI used in the decomposition. This is accomplished by dividing the ID by 2, or right shifting by 1 bit. Of course, if the right endpoint of the interval e is less than the right endpoint of the parent CEI, then we cannot use the parent CEI. In that case or if a local ID is an odd number, then we output a maximal CEI, remove the maximal CEI from R, and continue to decompose R from the local ID of the leftmost unit-length CEI for R. This process ends when R is empty.

Fig 7(a) shows an example of decomposing $[8, 14)$ using **BestPartition** algorithm. The query interval has 6 unit-length CEIs with consecutive local IDs from 8 to 13. The algorithm initializes R to be $[8, 14)$. It then starts from the leftmost unit-length CEI for R, which has local ID 8, and successively tries to use its parent CEI by dividing the local ID by 2. The if-condition that checks l is true when the local IDs are 8 and 4, but it is not true when the local ID is 2 since its

```
Decomposition (q, a, b) {
  P_L = L⌈a/L⌉; P_R = L⌊b/L⌋;
  R_L = [a, P_L); // left remnant interval
  R_R = [P_R, b); // right remnant interval
  if (P_L > P_R) { // q lies between 2 consecutive posts
    BestPartition(q, a, b, P_R);
  else {
    if (P_L < P_R) {
      for (p = P_L; p < P_R; p = p + L)
        insert(2p + 1, q);
    }
    if (R_L ≠ φ) BestPartition(q, a, P_L, P_L − L);
    if (R_R ≠ φ) BestPartition(q, P_R, b, P_R);
  }
}

BestPartition (q, s, e, P_L) {
  R = [s, e); // initial remnant interval
  l = s − P_L + L;
        // local ID of leftmost unit-length CEI
  while (1) {
    if ((l is even) ∧ (CEI ⌊l/2⌋ does not exceed e))
      l = l/2; // local ID of parent CEI
    else {
      insert(2P_L + l, q); // a maximal-sized CEI
      remove CEI l from R;
      if (R == φ) exit;
      else
        l = local ID of leftmost unit-length CEI for R;
    }
  }
}
```

Figure 6: Pseudo code for decomposition algorithm.

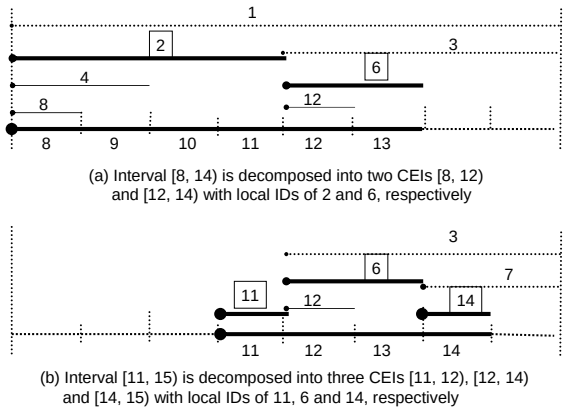


Figure 7: Examples of decomposition with BestPartition.

parent CEI has a right endpoint exceeding 14. Therefore, it finds a maximal-sized CEI with local ID 2. This CEI is removed from R and the new remnant becomes $[12, 14)$. Hence, the algorithm continues with the leftmost unit-length CEI for R, which now has local ID 12. This time the process stops when the local ID is 6 because its parent CEI (local ID 3) has a right endpoint exceeding 14. Hence, the second maximal-sized CEI is the one with local ID 6. After that, the algorithm stops because there is no more remnant interval, i.e., $R = \phi$.

Fig 7(b) shows the decomposition of $[11, 15)$. The interval has 4 unit-length CEIs with local IDs from 11 to 14. Initially, R is $[11, 15)$. The algorithm starts from the leftmost unit-length CEI whose local ID is 11. Since 11 is an odd number, it finds one maximal-sized CEI with local ID 11 and this CEI is removed from R. Hence, R becomes $[12, 15)$. It continues with the leftmost unit-length CEI with local ID 12. It finds another maximal-sized CEI with local ID 6. Finally, it finds the third maximal-sized CEI with local ID 14.

Note that the **BestPartition** algorithm incrementally computes the local IDs for ancestor CEIs and the leftmost unit-length CEI for the remnant interval R, if any. The computation involves only additions and logical shifts by 1 bit. Also note that decomposition means partitioning the query intervals into a set of CEIs. A decomposition is minimal when the number of CEIs in the decomposition is minimal among all the possible decompositions. We don't want the decomposed CEIs to be overlapping because it can result in duplicated query IDs in a search result. The storage cost would be higher as well because a query ID would be inserted into more ID lists.

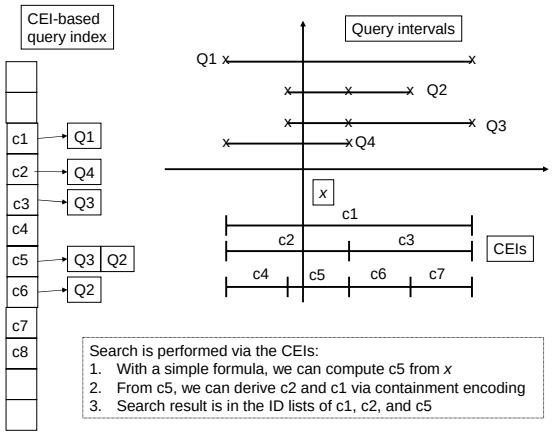


Figure 8: Example of CEI-based query indexing.

Lemma 1 *Assume there is a decomposition, D_i, that includes a CEI c_i. If there is another decomposition, D_j, that includes a CEI c_j and CEI c_j is an ancestor of c_i, then D_i cannot be a minimal decomposition.*

Lemma 2 *A minimal decomposition is unique.*

Theorem 1 BestPartition *decomposes any query interval of length less than L into a minimal number of CEIs.*

The proofs of both lemmas and the theorem are provided in the Appendix. Fig. 8 shows an example of a CEI-based query index. It shows the decomposition of four query intervals: $Q1, Q2, Q3$ and $Q4$ within a specific segment containing CEIs of $c1, \cdots, c7$. $Q1$ completely covers the segment, and its ID is inserted into $c1$. $Q2$ lies within the segment and is decomposed into $c5$ and $c6$, the largest CEIs that can be used. $Q3$ also resides within the segment, but its right endpoint coincides with a guiding post. As a result, we can use $c3$, instead of $c7$ and $c8$ for decomposition. Similarly, $c2$ is used to decompose $Q4$. As shown in Fig. 8, query IDs are inserted into the ID lists associated with the decomposed CEIs.

To delete a query interval, it is decomposed to one or more CEIs. Then the query ID is deleted from the ID lists associated with the decomposed CEIs.

```
Search (x) { // x ∈ A and x is a non-integer number
  i = ⌊⌊x⌋/L⌋; // segment ID
  l = ⌊x⌋ − iL + L;
       // local ID of the leaf CEI that contains x
  for (j = 0; j ≤ k; j = j + 1) {
    c = 2iL + l; // global ID of the CEI
    if (IDList[c] ≠ NULL) { output(IDList[c]); }
    l = l/2; // local ID of parent
  }
}
```

Figure 9: Pseudo code for the search algorithm.

3.3 Searching the query index

Fig. 9 shows the pseudo code for the search algorithm. It first computes the ID of the segment that contains a data value x. From Property 7, this ID is computed via $\lfloor \lfloor x \rfloor / L \rfloor$. Then, it computes the local ID of the leaf CEI that contains x, which is $\lfloor x \rfloor - iL + L$. Based on Property 6, there are exactly $k+1$ CEIs that contain any data value x. Hence, the search results are stored in the $k+1$ ID lists associated with these CEIs. Because of the containment encoding (see Property 4), the local IDs of these $k+1$ CEIs can be computed by successively dividing that of the leaf CEI by 2.

The search algorithm is simple and efficient. For each search, there is an integer conversion from a floating point number. The rest involves only integer additions and logical shifts. No complex floating point multiplications or divisions are needed. As a result, the search operation is extremely fast. The search time is independent of n, where n is the total number of query intervals maintained, in terms of finding the $k+1$ ID lists. However, if reporting time is also included, the search time increases as n increases. This is because there are more IDs stored in each ID lists, on average, as n increases.

As an example, to search with a data value x in Fig. 8, we first compute the local ID of the unit-length CEI that contains it. In this case it is $c5$. Then, from $c5$, we can compute the local IDs of all its ancestors that contain $c5$. In this case, they are $c2$ and $c1$. As a result, the search result is contained in the 3 ID lists associated with $c1, c2$ and $c5$. We can verify from Fig. 8 that the result indeed contains $Q1, Q2, Q3$ and $Q4$.

3.4 Other system issues

So far, we have assumed $q : [a, b)$. When the right endpoint is also inclusive, i.e., $[a, b]$, we need to use additional r point CEIs. These points are maintained separately. They will be useful only if the attribute is of integer data type. During decomposition, the query ID is additionally inserted into the ID list associated with the point CEI representing b. At search, we also report the IDs stored in the ID list associated with the integer data value.

A query interval may be an open-ended interval, i.e., $A > 4$. In this case, a query ID can be inserted into r/L CEIs in the worst case. If L is small, the storage cost can be high. Fortunately, we can choose a large L to reduce the storage cost. In fact, we can set $L = r$. In that case, the search time may degrade a bit because we need to collect the search result from exactly $k + 1$ ID lists, where $k = \log(L) = \log(r)$.

In practice, the CEI-based query index is naturally suited for parallel processing. We can control both storage cost and search time by choosing a relatively large L and by properly separating r into partitions. One machine can then be used to process a partition.

4 Performance evaluation

4.1 Simulation studies

Simulations were conducted to evaluate and compare the CEI-based index with the LCI-based index, the UI-based index and the IS-lists approach [7]. We focused on storage cost and search time because they are two of the most important metrics in determining the effectiveness of a query index for stream processing. A low storage cost is desirable so that the entire index can be loaded into main memory. A fast search time is important so that a rapid stream can be properly handled. The IS-lists approach represents the state of the art direct indexing approach. It has $O(\log(n))$ search time and $O(n \log(n))$ storage cost. In contrast, the search times, in terms of finding the ID lists from the covering VCs, of the UI-based, LCI-based and CEI-based indexes are independent of n. However, in our simulations, the search time included the report time. As a result, it generally increases as n increases. This is because on average the ID lists contain more query IDs. We implemented the three alternative VC-based indexes. For the IS-lists approach, we downloaded from the Web an implementation from the author [5] and used it to run similar set of input data. The simulations were conducted on an RS6000 model 43P machine running AIX 5.1.

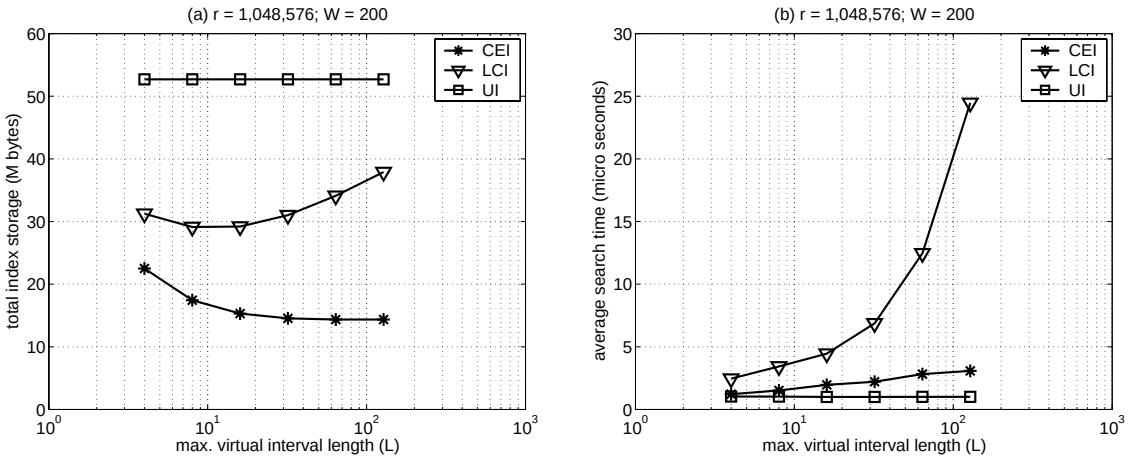


Figure 10: The impacts of L, when r is large, on (a) total index storage cost; (b) average search time.

Attribute values range from 0 to r, with 1,048,576 as the large r and $65,536$ as the small r. The starting point of a query interval was randomly chosen between 1 and $r-1$ with a length of w randomly chosen between 1 and W. Namely, $\bar{w} = W/2$. Various W's were used, with a default 200. A total of n query intervals were generated and inserted. In the experiments, n was varied from 5,000 to 640,000. After insertion, we performed 50,000 random searches and computed the average search time and storage cost. The data value for search was a floating point number chosen randomly between 1 and r. The search time included the report time, which involves reporting the query IDs in the ID lists. If there are large number of IDs stored in the ID lists, the report time will be large as a result.

4.2 Impact of maximum interval length (L)

We first examine the impact of the maximum interval length L on the three alternative VC-based indexes in terms of storage cost and average search time. For this set of experiments, we compare the UI-based, LCI-based and CEI-based indexes. Figs. 10(a) and (b) show the storage cost and average search time, respectively, of the three indexes when r is large. In contrast, Figs. 11(a) and (b) show storage cost and average search time, respectively, when r is small. For the entire set of experiments, a default W of 200 was used. The total number of query intervals inserted was 100,000.

As expected, the UI-based index has a high storage cost and fast search time for a large r and a small r. Moreover, the storage cost and search time are independent of L because it used

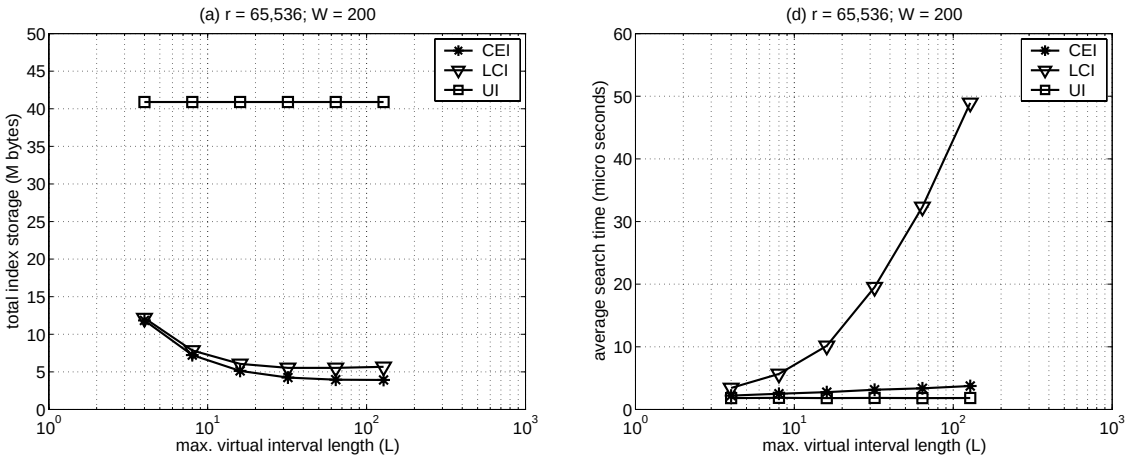


Figure 11: The impacts of L, when r is small, on (a) total index storage cost; (b) average search time

unit-length virtual intervals. The CEI-based index has both low storage cost and fast search time for both r's. In general, the average search time increases as L increases. This is particularly true for the LCI-based index because its search time increases much more quickly than the CEI-based index as L increases. This can be seen from the formula for C described in Table 1. Hence, it argues for a small L for the LCI-based index from the search time perspective. However, if L is too small, the storage cost tends to increase for both schemes. This is because more query IDs are stored in the ID lists because the total number of decompositions D is larger with a smaller L (see Table 1). Moreover, when r is large, the storage cost of the LCI-based index also increases as L increases (see Fig. 10(a)). However, under all cases, the storage cost of the CEI-based index decreases as L increases.

The average search time of the CEI-based index increases much slower than the LCI-based index as L increases. This is because the report time is the same for all different L's since n remains the same for all the experiments. The difference is due to the computation of the IDs of the extra covering CEIs for a larger L. Because $C = 1 + \log(L)$, it increases much slower than that of the LCI-based index, which is $2L - 1$. Hence, for the CEI-based index, it argues for a larger L since the storage cost tends to decrease and the average search time increases only slightly as L increases. This result is important in practice. We can choose a larger L in order to minimize the storage cost without significantly degrading the average search time.

Under all cases, the CEI-based index performs better than the LCI-based index in terms of both storage cost and average search time. The advantage in average search time is particularly

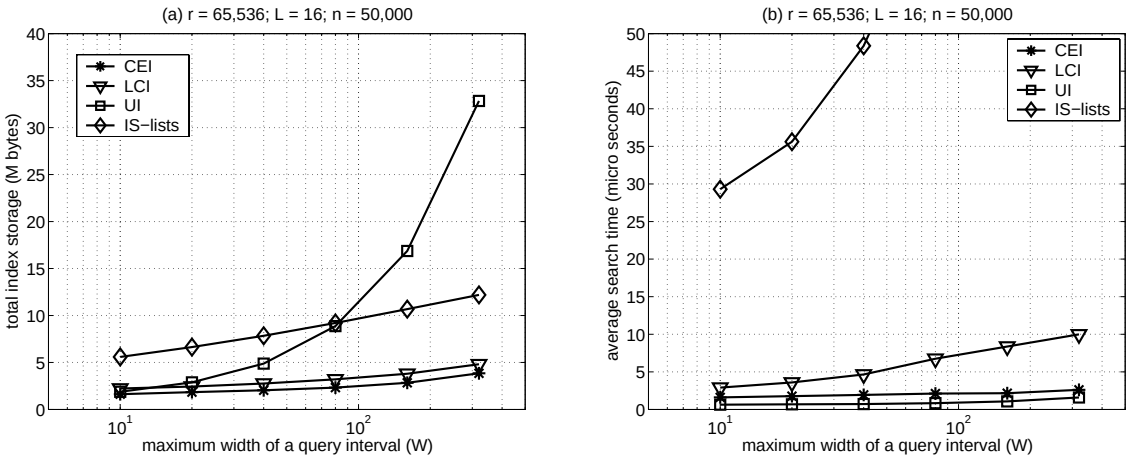


Figure 12: Impact of W on (a) index storage cost; (b) average search time.

dramatic when L is large (see Fig 10(b) and Fig. 11(b)).

4.3 Comparisons of CEI-based index with IS-lists

Now we compare the CEI-based scheme with the UI-based, the LCI-based and the IS-lists approaches. Figs. 12(a) and (b) show the impact of the maximal query interval width, W, on the performance of the four schemes. Fig. 12(a) shows the storage cost while Fig. 12(b) shows the average search time. For this experiment, W was varied from 10 to 320, $L = 16$, $r = 65,536$ and $n = 50,000$ were used.

Both figures show that the UI-based index has high storage cost but fast search time for the entire range of W's. The CEI-based has both low storage cost and fast search time. All three VC-based approaches outperform the IS-lists approach in terms of search time. For the smallest $W = 10$, the performance difference is almost 20 times. Namely, the average search time of the CEI-based index is only 1/20 that of the IS-based approach. The IS-lists approach also has a higher storage cost than both the LCI-based and CEI-based indexes. But, it has a smaller storage cost than the UI-based index when W is large (see Fig. 12(a)).

Figs 13(a) and (b) show the impact of n on the storage cost and average search time, respectively, when r is large. Figs. 14(a) and (b) show the index storage cost and average search time, respectively, when r is small. For this set of experiments, W was 200 and n was varied from 5,000 to 640,000 and $L = 16$.

From Fig. 13(b) and Fig. 14(b), the average search time of the CEI-based index outperforms both the LCI-based index and the IS-lists approach. Moreover, it is very close to the UI-based

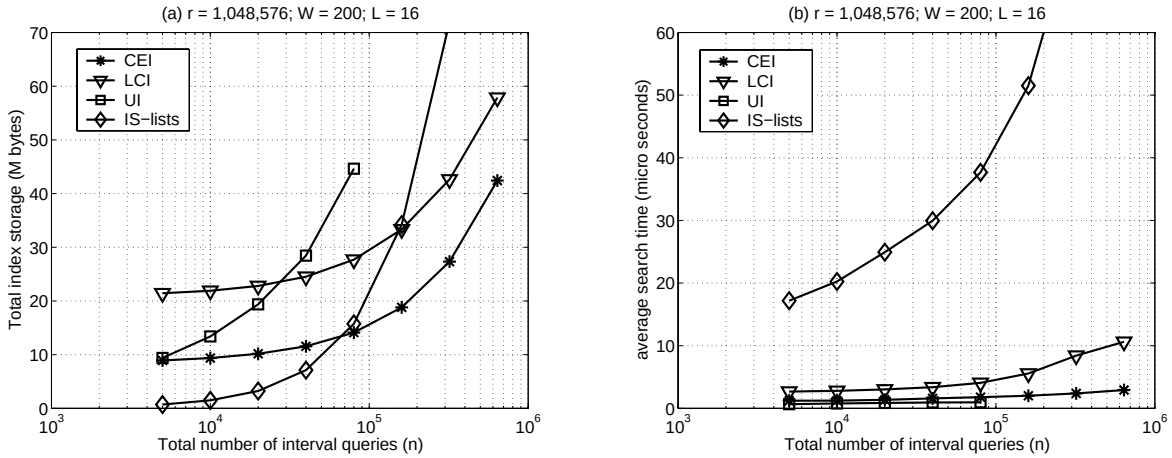


Figure 13: Impact of n, when r is large, on (a) index storage cost; (b) average search time.

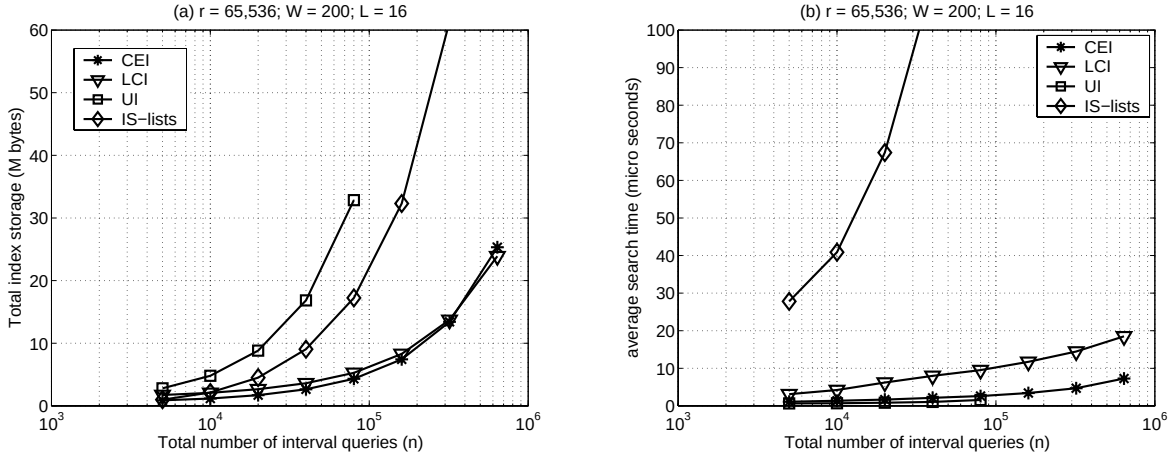


Figure 14: Impact of n, when r is small, on (a) index storage cost; (b) average search time.

index,which has the best search time performance. The IS-lists approach, which is the best direct indexing approach, has the highest search time. Note that there are missing data points for the UI-based and the IS-lists approaches. This is because for those data points, n was too large for both schemes to be executed properly on our computer system, which is an AIX model 43P machine. Memory was exhausted by the indexes.

The storage cost of IS-lists approach is better for the case of a large r when n is relatively small. But, as n increases, this advantage disappears. This is because the UI-based, LCI-based and CEI-based indexes need to maintain pointers to the the ID lists for every VC. When r is large and n is small, the storage cost for the pointers dominates the total storage cost, even though most of them are NULL. In contrast, the IS-lists approach does not have to maintain those pointers. However, for a small r, the storage cost advantage of the IS-lists approach also disappears.

5 Summary

We have presented a CEI-based query indexing approach for efficient stream processing. It is main memory-based approach and has both low storage cost and fast search time, ideally suited for stream processing. The CEI-based query index is centered on a set of virtual containment-encoded intervals. The range of a numerical attribute is partitioned into segments. For each segment, a set of virtual containment-encoded intervals were carefully defined and properly labeled as a perfect binary tree. A query interval is first decomposed into one or more CEIs and the query ID is inserted into the ID lists associated with the decomposed CEIs. Search is carried out indirectly via these CEIs. There is no need to compare a data value with any of the query boundaries during a search. The containment encoding makes possible efficient decomposition and search operations. Almost all of the operations can be carried out with integer additions and logical shifts. Simulations were conducted to evaluate the performance of the CEI-based query index. The results show that, compared with other alternatives, the CEI-based query index does have low storage cost and fast search time and it outperforms other alternatives in both.

References

[1] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applica-

tions. In *Proc. of VLDB*, 2002.

[2] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of ACM SIGMOD*, 2001.

[3] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, 1984.

[5] E. Hanson. ISlist.tar: A tar file containing C++ source code for IS-lists. http://www-pub.cise.ufl.edu/~hanson/IS-lists/.

[6] E. Hanson, M. Chaaboun, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *Proc. of ACM SIGMOD*, pages 271–280, 1990.

[7] E. Hanson and T. Johnson. Selection predicate indexing for active databases using interval skip lists. *Information Systems*, 21(3):269–298, 1996.

[8] C. P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. of ACM SIGMOD*, 1991.

[9] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of ACM SIGMOD*, 2002.

[10] H. Samet. *Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[11] K.-L. Wu, S.-K. Chen, P. S. Yu, and M. Mei. Efficient interval predicate indexing for fast event matching. Technical Report RC 23063, IBM T. J. Watson Research Center, Jan. 2004.

[12] K.-L. Wu and P. S. Yu. Efficient query monitoring using adaptive multiple key hashing. In *Proc. of ACM CIKM*, pages 477–484, 2002.

Appendix

Lemma 1 *Assume there is a decomposition, D_i, that includes a CEI c_i. If there is another decomposition, D_j, that includes a CEI c_j and CEI c_j is an ancestor of c_i, then D_i cannot be a*

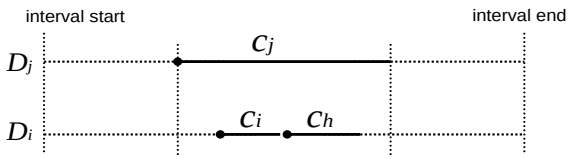


Figure 15: Example of two decompositions.

minimal decomposition.

Proof: Assume decomposition D_i is minimal. Fig. 15 shows an example of two decompositions D_i and D_j. D_i contains CEI c_i and D_j contains CEI c_j and c_j is an ancestor of c_i. From the definition of CEIs, there is no CEI of length less than $|c_j|$ that can cross the vertical lines at either endpoints of CEI c_j due to containment property. As a result, there are more than one CEI besides c_i that is needed to cover the region of c_j. Hence, D_i cannot be minimal, which is a contradiction. □

Lemma 2 *A minimal decomposition is unique.*

Proof: Suppose there are two decompositions, D_1 and D_2, that are different and are both minimal. If we sort both decompositions based on the left endpoints of the decomposed CEIs. There must exist a point where the first mismatch between D_1 and D_2 occurs if we scan the decompositions from left to right. The mismatch is due to different lengths since both have the same left endpoint. Hence, the small-sized CEI is a descendant of the the large-sized CEI. From Lemma 1, the one with the small-sized CEI cannot be minimal. Therefore, there is only one unique minimal decomposition. □

Theorem 1 BestPartition *decomposes any query interval of length less than L into a minimal number of CEIs.*

Proof: Suppose D_{opt} is the minimal decomposition. Let D_{bp} is the decomposition generated by the **BestPartition** algorithm. Consider both decompositions as sorted based on the left endpoints of the decomposed CEIs. If CEIs c_i and c_j, where $c_i \in D_{opt}$ and $c_j \in D_{bp}$, are the first CEIs that show a mismatch between the two decompositions. There are two cases to consider (see Figs. 16(a) and (b)). For the first case, $|c_i| < |c_j|$. Since c_j is an ancestor of c_i, based on

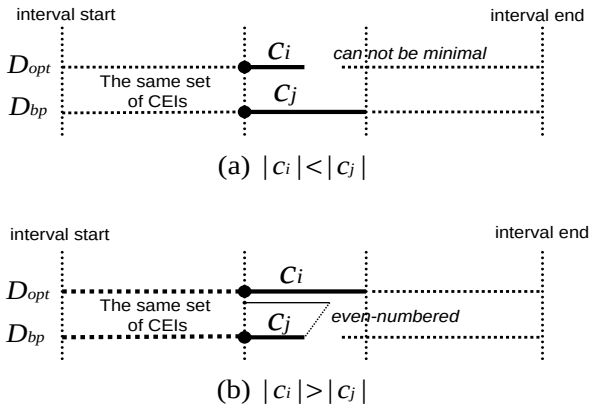


Figure 16: Scenarios of minimal decomposition relative to the decomposition by **BestPartition**.

Lemma 1, decomposition D_{opt} cannot be minimal. A contradiction occurs. For the second case, $|c_i| > |c_j|$. In this case, c_i must be an ancestor of c_j. Since both CEIs have the same left endpoint, c_j must have an even-numbered local ID, based on Property 8. This is true for any CEI that is an ancestor of CEI c_j and a descendant of CEI c_i as shown in Fig. 16(b). If CEI c_i has an even local ID, then its parent CEI must exceed the right endpoint of the query interval. Otherwise, D_{opt} would not be minimal. Hence, CEI c_i must either have an odd local ID or have a parent CEI whose length exceeds the right endpoint of the query interval. Under either condition, the **BestPartition** algorithm does not stop at CEI c_j. It would continue skipping the ancestor of CEI c_j until it reaches CEI c_i. Therefore CEI c_i, instead of c_j, should be in decomposition D_{bp}. Hence, CEIs c_i and c_j cannot be the first point where mismatch occurs. A contradiction occurs. As a result, **BestPartition** indeed finds the minimal decomposition. □.