RC23249 (W0406-102) June 18, 2004
Computer Science

IBM Research Report

Indexing Continual Range Queries 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

Indexing Continual Range Queries for Efficient Stream Processing

Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu

IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
{*klwu, skchen, psyu*}*@us.ibm.com*

Abstract. A large number of continual range queries could be issued against numerical data streams, such as stock prices, sensor readings, temperatures, and others. To efficiently process these long-running queries, only the potentially relevant queries should be evaluated against the data. We develop a virtual construct-based query indexing approach to efficiently identifying the range queries that match each data object in the streams. A set of virtual constructs, e.g., intervals in 1D space or rectangular regions in 2D space, are predefined such that it is efficient to find all the virtual constructs containing any given data object. Each virtual construct has a unique ID and an associated query ID list. The query index is built as follows. Each range query is first decomposed into one or more virtual constructs. The query ID is then inserted into the query ID lists associated with those decomposed virtual constructs. Search becomes extremely efficient. For a given data object, we first find all virtual constructs covering it. Then, we report the matched queries from the ID lists associated with the covering virtual constructs.

1 Introduction

A large number of continual range queries can be issued against a numerical data stream. Many stream applications can be modeled as continual range queries against a data stream, such as financial applications monitoring stock prices or interest rates, and security control applications monitoring sensor readings. In a stream model, individual data items can be relational tuples with well-defined attributes, such as network measurements, call records, meta data records, sensor readings and so on. These data items arrive in the form of streams continually and perhaps rapidly. Conceptually, every query must be continually evaluated against every data item in the stream.

As data items are streamed at an increasingly rapid rate, the processing of continual range queries against the stream becomes difficult, if not impossible, because CPU quickly becomes limited. Data items may have to be dropped without processing [18], or the system cannot handle so many continual queries. However, it is desirable for a system to process as many continual queries as possible against a stream that is as rapid as possible. Hence, it is important that only the potentially relevant queries are evaluated against a data item in the incoming stream.

One approach to quickly evaluating the relevant continual range queries is to use a query index. Each data item in an incoming stream is used to search the query index and identify the relevant queries. Though maybe conceptually simple, it is quite challenging to design such a query index in a streaming environment, especially if the stream is rapid. The range 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.

Range queries are generally difficult to index. Though existing spatial indexes, such as R-trees [9, 8], can be used to index range queries [17], most of them are disk-based approaches. A main memory-based query index is preferable for fast search performance in a streaming environment, especially if the number of continual range queries is large. In addition, R-trees degenerate quickly if range queries are highly overlapped [8].

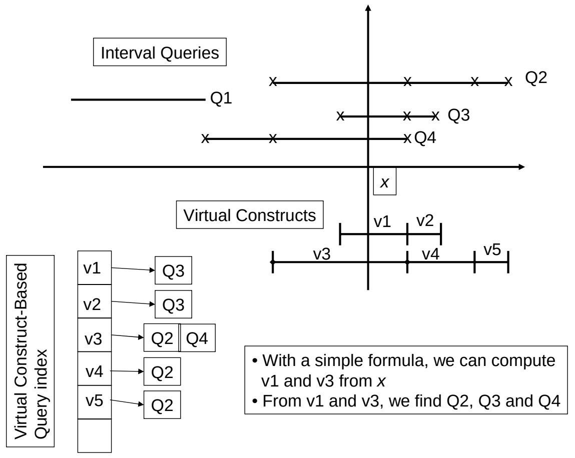
In this paper, we describe a virtual construct-based query indexing method for efficient processing of continual range queries against a numerical data stream. The key concept of a VC-based query index is indirect pre-computation of search results. A set of virtual constructs, VCs, are predefined. Each VC has a unique ID and an associated query ID list. The range query is decomposed into one or more VCs and the query ID is inserted into the query ID lists associated with the decomposed VCs. Search is conducted indirectly via these VCs by finding all the covering VCs for any data point. The challenges of such a VC-based query index include the definition and labeling of VCs, and the decomposition and search algorithms.

We present two VC-based query indexes that meet both objectives of low storage cost and fast search performance: a VCI (virtual construct interval) index for 1D range queries and a VCR (virtual construct rectangle) index for 2D range queries. The VCI index was first developed for efficient 1D interval indexing for fast event matching in content-based subscription e-commerce and e-service [20]. The VCR index was first developed for efficient indexing of highly-overlapping multidimensional range queries for fast event matching [19]. In this paper, we describe the use of both indexes in a streaming environment. Research is continuing to explore various alternative VCs such that the storage cost and the search performance can be further improved.

The paper is organized as follows. Section 2 presents the VCI index for one dimensional continual interval queries. Section 3 describes the VCR index for two dimensional continual range queries. Section 4 discusses related work. Finally, Section 5 summarizes our paper.

2 VCI indexing for 1D range queries

Before describing the details, we show an example of a VCI-based query index in Fig. 1. There are 5 virtual construct intervals, $v1, \cdots, v5$, which are used to decompose query intervals, $Q2, Q3$ and $Q4$. Note that there are more VCIs, but for simplicity we only show 5 of them. After decomposition, query IDs are inserted into the associated ID lists. To perform a search on data value x, we first 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 VCI-based query index, we find $Q2, Q3$ and $Q4$.



Interval Queries
Q1
Q2
Q3
Q4
x
Virtual Constructs
v1
v2
v3
v4
v5
Virtual Construct-Based Query index
v1 Q3
v2 Q3
v3 Q2 Q4
v4 Q2
v5 Q2
• With a simple formula, we can compute v1 and v3 from x
• From v1 and v3, we find Q2, Q3 and Q4

Fig. 1. Example of a VCI-based index for 1D interval queries.

2.1 System model

For clarity, *query intervals* are user-defined intervals in 1D range queries. On the other hand, *virtual construct intervals* (VCIs) are virtual intervals used to decompose query intervals. We focus on simple 1D range queries, where each contains a single interval predicate on an attribute. The result is applicable to complex queries, where each contains a conjunction of more than one interval predicate. For example, an efficient interval index can be maintained for each attribute in a two-phase algorithm involving complex queries, such as the ones presented in [7, 21].

We assume that all query intervals are defined on an attribute A. A can be of integer or non-integer data type. Query intervals include both endpoints, and the endpoints are integers. However, if the endpoints are not integers, we can expand them to the nearest integers. The expanded intervals are then decomposed and indexed. VCI indexing is still effective in identifying candidate queries. However, a final checking is needed to determine if the candidate queries indeed match the data item. Namely, we deal with the case that a query interval is represented as $q : [x, y]$, where q is the query ID, x and y are integer endpoints and $y > x$. However, data values can be non-integers. We assume that query intervals fall between a_0 and $a_0 + R - 1$, i.e., the attribute range is R.

2.2 Defining VCIs

We describe two sets of VCIs: an SCI and an LCI. In SCI, we define L unique VCIs that start at each integer value and have respective lengths of $1, 2, \cdots, L$. L is the maximal length of a VCI. The IDs for the L VCIs that starts at attribute value $a_0 + j$ are $jL, jL + 1, \cdots, (j + 1)L - 1$, respectively. Fig. 2 shows an example of simple construct intervals (SCIs) and their unique IDs when $L = 4$. The unique ID for a VCI $[a, b]$ in an SCI approach is computed as follows:

$$v = (a - a_0)L + (b - a) - 1.$$

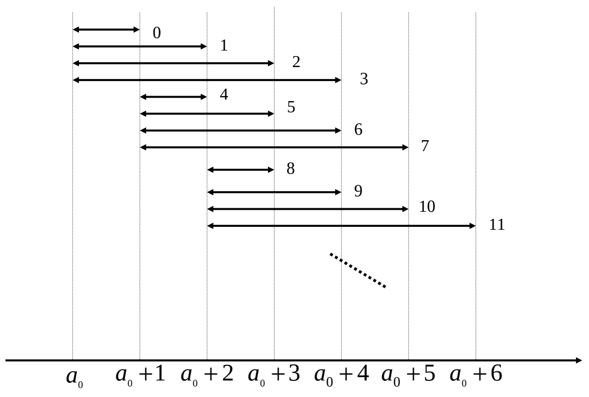


Fig. 2. Example of simple construct intervals (SCIs) with $L = 4$.

In contrast, LCI defines $\log(L)+1$ VCIs for each integer attribute value. Assume $L = 2^k$, where k is an integer, the lengths of virtual construct intervals in LCI are $2^0, 2^1, \cdots, 2^k$. Fig. 3 shows an example of LCIs with $L = 4$ and $k = 2$. Compared with SCI, LCI has in general fewer number of VCIs for the same L. The unique ID of a VCI $[a, b]$, where $b - a = 2^l$, $0 \leq l \leq k$, is computed as follows:

$$c = (a - a_0)(k + 1) + l.$$

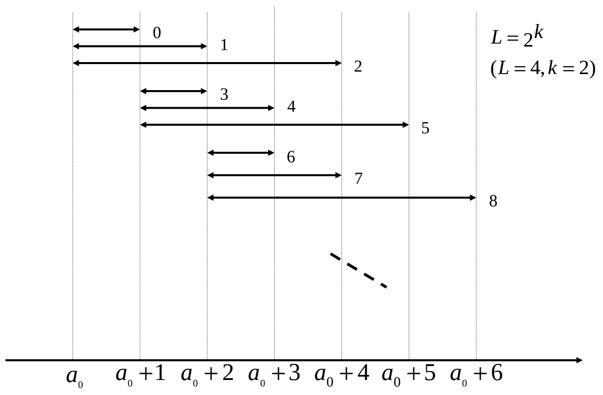


Fig. 3. Example of logarithmic construct intervals (LCIs) with $L = 4$.

2.3 Decomposition algorithm

The decomposition algorithms for SCI and LCI are rather simple and straightforward. Both are mostly similar except for the handling of a remnant interval with length less than L. The decomposition algorithm works as follows. First, we initialize a remainder interval to be the query interval $[x, y]$. If $y - x < L$, then no decomposition is needed. The VCI $[x, y]$ can be used to decompose the query. Otherwise, we repeatedly use the maximal-sized VCI to decompose and cut the remainder interval from the left endpoint. This process ends until the length of the remainder interval is less than L. Hence, a query $[x, y]$ is decomposed into $m = \lceil \frac{y-x-1}{L} \rceil$ VCIs with length L and a remnant. For the SCI approach, the remnant can be decomposed with a single VCI. For the LCI approach, the remnant may need more than one VCIs to decompose it. For example, if $L = 16$ and a remnant of size 7 needs three VCIs with lengths of $1, 2$ and 3, respectively.

After decomposition, the query ID is inserted into the query ID lists associated with the decomposed VCIs. From Fig. 1, the storage cost depends on the query ID lists and the array of pointers to the associated query ID lists. For the same L, the SCI approaches defines more VCIs than the LCI approach. Hence, the pointer array is larger for the SCI approach. On the other hand, a query ID is inserted into more ID lists for the LCI approach. In general, LCI has a lower storage cost [20].

2.4 Search algorithm

To find the queries that cover a data value, we need to first find all the VCIs that can potentially cover the data value. Fig. 4 shows the covering VCIs for any data value $a_0 + j$ for the SCI approach; Fig. 5 shows covering VCIs for the LCI approach. These VCIs can be efficiently enumerated.

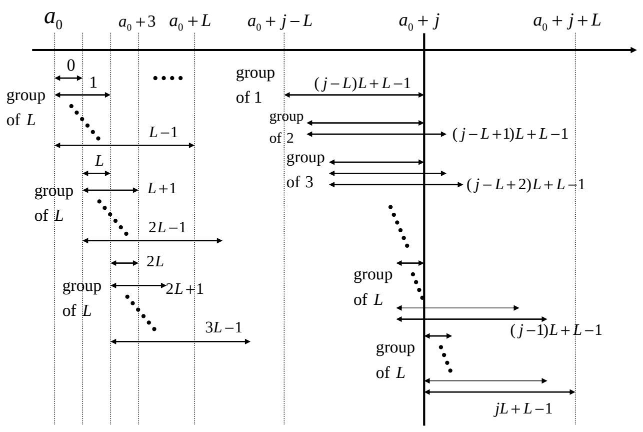


Fig. 4. Covering VCIs for a data point $a_0 + j$ for the SCI approach.

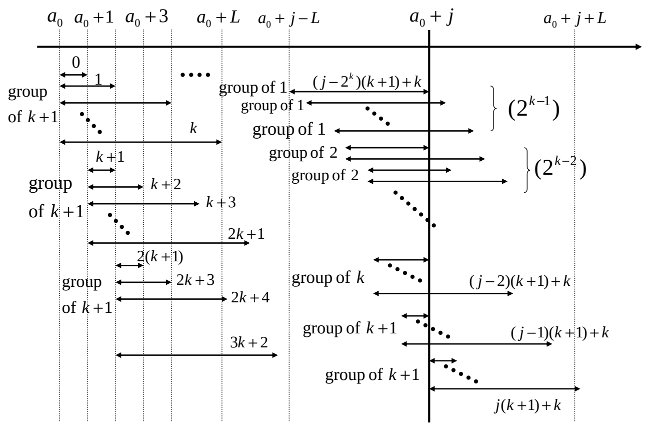


Fig. 5. Covering VCIs for a data point $a_0 + j$ for the LCI approach.

Let $V_j^{\min}$ denote the minimal ID of the covering VCIs of $a_0 + j$. From Fig. 4, $V_j^{\min} = (j - L)L + L - 1$ for the SCI approach. On the other hand, $V_j^{\min} = (j - L)(k + 1) + k$ for the LCI approach (see Fig. 5). Let us also define a difference table D_j for a data point $a_0 + j$: D_j stores the differences between all the covering VCIs of $a_0 + j$ and $V_j^{\min}$. There is an important property among the difference tables for all the attribute values. $D_i = D_j$ even if $i \neq j$ for $i, j \geq L$. This property can be easily verified from Fig. 4 and Fig. 5. Essentially, we can move around as a unit the entire drawings surrounding $a_0 + j$ of these two figures and they become the covering VCIs of another attribute value. In other words, the relative distance among the covering VCIs of an attribute value stay the same for all data points. Because $D_i = D_j$ even if $i \neq j$, we can pre-compute, store and use a single D to enumerate all the IDs of the covering VCIs for any attribute value. We only need to compute $V_j^{\min}$ and the entire covering VCIs can be computed by simple additions of $V_j^{\min}$ to each element stored in D. After the covering VCIs are found, the search results are contained in the query ID lists associated with these covering VCIs. Hence, search can be very efficient with VCI-based query indexing.

2.5 Performance

Detailed simulations were conducted to evaluate the performance of VCIs under various conditions [20]. Fig. 6 and Fig. 7 show the average search time and total index storage cost, respectively, with $R = 5000$ for various alternative query indexing schemes. The IS-list approach [12] is the best prior art in main memory-based interval index. However, it does not have a good search performance, nor a low storage cost. The Dlist approach is a simple and direct listing method. It has the best search performance, but has a rather high storage cost. Different optimization techniques were applied to the LCI approach [20]. The best one (LCI-BV(DT)) has a near-best search performance and the lowest storage cost. These two figures show that indeed the VCI-based query index has a low storage cost and excellent search performance. Hence, it is suitable for indexing continual interval queries in a stream environment.

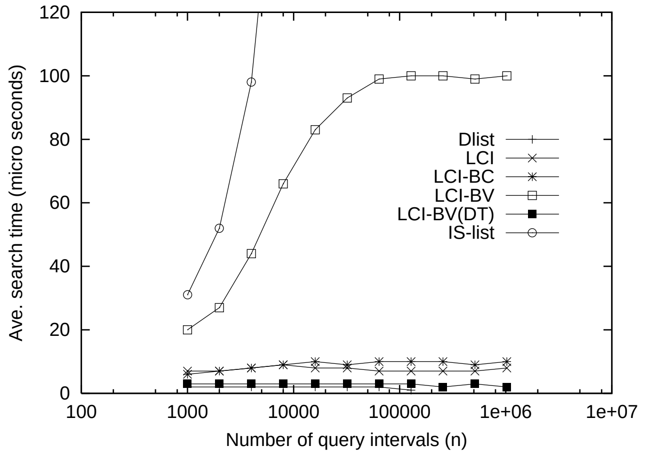


Fig. 6. Comparisons of various interval query indexing schemes in terms of average search performance.

3 VCR indexing for 2D range queries

3.1 System model

Now we examine the case where queries are conjunctions of two intervals involving attributes X and Y. For simplicity, assume that the attribute ranges are $0 \leq X < R_x$ and $0 \leq Y < R_y$, respectively. Query boundaries are assumed to be defined along the integer lines of X and Y. However, data points can be any non-integer numbers. For example, in a 2D space, the spatial ranges for specifying user queries are defined with integers based on a virtual grid imposed upon a monitoring region, but object positions can be any non-integer. Fig. 8 shows a scenario where two queries are defined along the integer grid lines: $q_1 : (3, 3, 5, 6)$ and $q_2 : (6, 7, 5, 5)$. Query q_1 has the bottom-left corner at $(3, 3)$ and a width of 5 and a height of 6. Three data items are specified with non-integer numbers: $d_1 : (9.3, 4.15)$, $d_2 : (3.6, 5.2)$ and $d_3 : (7.5, 9.5)$.

3.2 Defining VCRs

For each integer grid point (a, b), where $0 \leq a < R_x$ and $0 \leq b < R_y$, we define a set of B virtual construct rectangles as basic building blocks. These B VCRs share the common bottom-left corner at (a, b) but have different shapes and sizes. Fig. 9 shows an example of 9 VCRs whose bottom-left corners are at $(0, 0)$. We assume the maximum side lengths of a VCR are $L_x = 2^k$ and $L_y = 2^k$, where $k \geq 0$ and k

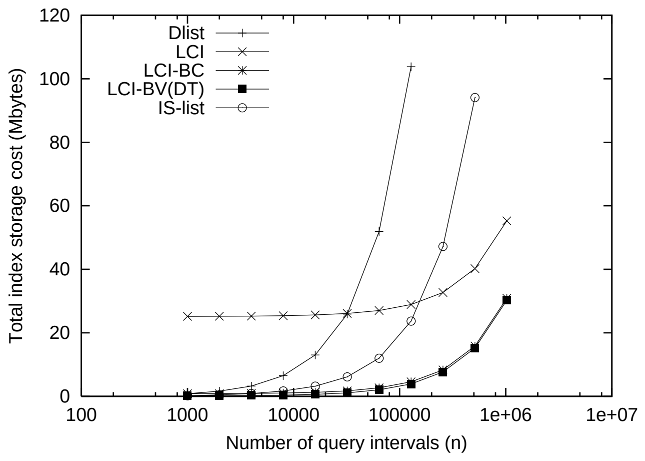


Fig. 7. Comparisons of various interval query indexing schemes in terms of total storage cost.

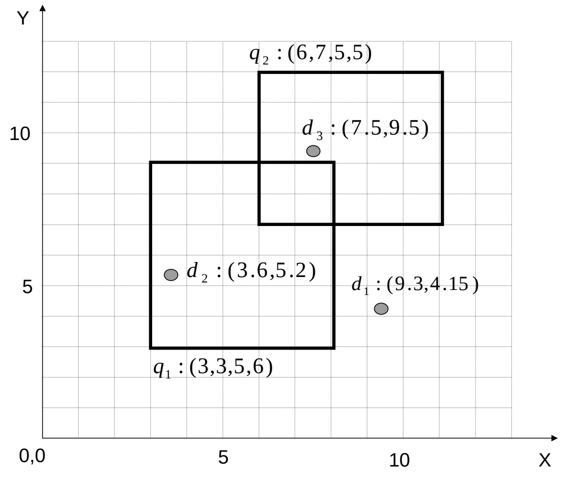


Fig. 8. System model for 2D range queries and data points.

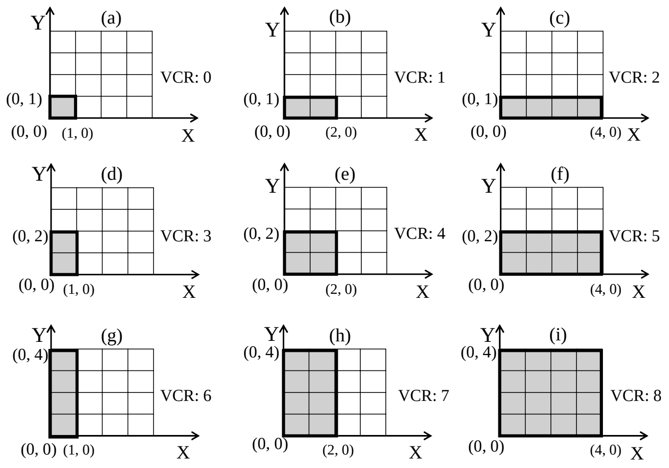


Fig. 9. An example of 9 VCRs with bottom-left corners at $(0, 0)$.

is an integer. If l_x and l_y are the side length of a VCR, then $l_x = 2^i$ and $l_y = 2^j$, where $0 \leq i \leq k$, $0 \leq j \leq k$, and i and j are integers.

Each VCR has a unique ID. The ID of a VCR $(a, b, 2^i, 2^j)$ can be computed as follows:

$$V(a, b, 2^i, 2^j) = B_r(a + bR_x) + j(k + 1) + i. \tag{1}$$

The first term is derived by horizontally scanning the integer grid points from $(0, 0)$ to $(R_x - 1, 0)$, then from $(0, 1)$ to $(R_x - 1, 1), \cdots$, until $(a - 1, b)$. There are $(a + bR_x)$ such grid points (see Fig. 8 for scanning the grid points). For each grid point, there are B VCRs defined. The second term is derived by the ID assignment shown in Fig. 9. Note that these VCRs are virtual. A virtual VCR becomes activated when it is used to cover the region of a continual range query.

3.3 Decomposition algorithm

A range query $q : (a, b, w, h)$ is first decomposed into one or more VCRs. After decomposition, the query ID q is inserted into each of the ID lists associated with those decomposed VCRs. Here, we present two decomposition methods: simple decomposition and overlapped decomposition.

Fig. 10(a) shows an example of using a simple decomposition (SD) to decompose a query rectangle $(3, 3, 11, 6)$. Assume that $L_x = 4$ and $L_y = 4$. First, it is decomposed into 2 strip rectangles: $(3, 3, 11, 4)$ and $(3, 7, 11, 2)$. Then each strip rectangle is decomposed into 4 VCRs each. Hence, $(3, 3, 11, 6)$ is decomposed into a total of 8 VCRs. These 8 VCRs have different sizes, 4×4, 4×2, 2×4, 2×2, 1×4 and 1×2. The overlapping among them is minimal. It occurs only on the boundary lines.

In SD, many small-sized VCRs may be activated by query decompositions. Hence, the average number of decomposed VCRs per query can be large. Since a query ID is inserted into each of the ID lists associated with the decomposed VCRs, storage requirement may become an issue.

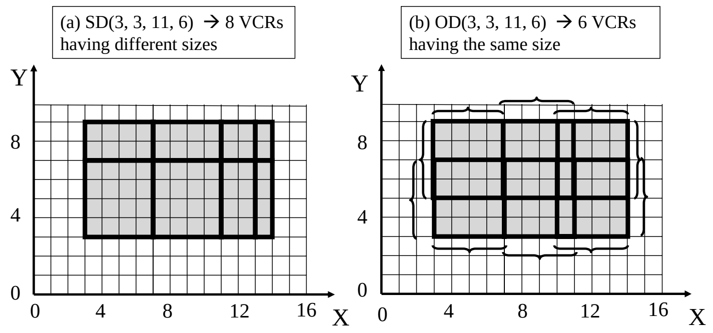

(a) SD(3, 3, 11, 6) → 8 VCRs having different sizes
(b) OD(3, 3, 11, 6) → 6 VCRs having the same size
Y
X
0
4
8
12
16

Fig. 10. Examples of (a) a simple decomposition; (b) an overlapped decomposition.

In contrast to SD, the overlapped decomposition (OD) uses the same-sized VCR to decompose a given range query. Basically, OD is very similar to SD in creating strip rectangles and using the largest VCR to decompose each strip rectangle. The difference between OD and SD is in how they handle the remnants of a strip rectangle. Overlapping is allowed in OD. To achieve this, the left boundary of the last VCR is allowed to shift backward so that the same sized VCR is used in the decomposition. Similarly, the bottom of the last strip rectangle is allowed to shift downward so that the last strip rectangle has the same height as those of the other strip rectangles.

As an example, Fig. 10(b) show the result of using OD to decompose the same query rectangle $(3, 3, 11, 6)$. We only use a 4×4 VCR for the decomposition and there are only 6 decomposed VCRs, instead of 8 as in SD. Compared with SD, OD better facilitates the sharing and reuse of decomposed VCRs among queries. It reduces the number of activated VCRs. Fewer activated VCRs make it more effective to reduce the storage requirement.

3.4 Search algorithm

To search for all the queries that contain a data point, we only need to find all the activated VCRs that contain that point. Assume that $CQ(d)$ is the set of queries that contain a data point d; $CV(d)$ is the set of covering VCRs that contain d. $CQ(d)$ can be computed from $CV(d)$ and the VCR-based query index. Because the way VCRs are predefined, $CV(d)$ can be efficiently enumerated.

$CV(\cdot)$'s for all object positions share two common properties: *constant size* and *identical gap pattern*. For ease of discussion, we focus on a data point (x, y) that is not under the boundary regions. The boundary regions are defined by $0 \leq x < L$ or $R_x - L \leq x < R_x$ or $0 \leq y < L$ or $R_y - L \leq y < R_y$. However, we can also efficiently compute $CV()$'s for locations in these regions.

Property 1. The number of VCRs potentially containing a data point is the same for all data points. Namely, $|CV(d_i)| = |CV(d_j)|$ even if $d_i \neq d_j$.

Property 2. If we sort, in an increasing order, the IDs of VCRs in each $CV(\cdot)$, then $v^{d_i}_{m+1} - v^{d_i}_{m} = v^{d_j}_{m+1} - v^{d_j}_{m}$ for $1 \leq m < |CV(\cdot)|$ and any two objects d_i and d_j. Here,

$v_m^{d_i}$ is the m-th VCR in the sorted $CV(d_i)$. In other words, the gap between any two VCRs of matching positions is identical for any two data points.

These two properties can be easily verified via an example. Fig. 11 shows $CV(d)$ for a data point d which is expressed as (x, y), where $a < x < a + 1$, $b < y < b + 1$, and a and b are integers. The bottom-left corners of these covering VCRs must reside in the south-west shaded area of (x, y) and the upper-right corners must reside in the north-east shaded area of (x, y). It can be easily verified that if a VCR whose bottom-left and upper-right corners are positioned in the respective shaded areas, it will indeed contain (x, y). These two properties can be proved by first grouping all the drawings in Fig. 11 as a unit and then moving it around. When the center is moved from (a, b) to another point (c, d), the relative positions of all the covering VCRs stay the same.

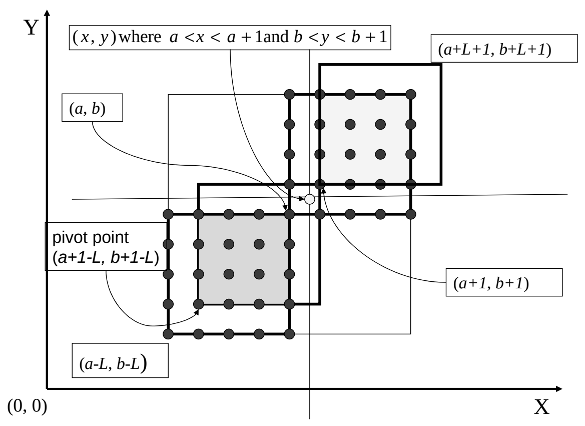


Fig. 11. Covering VCRs that contain a data point.

With these two properties, we can design an efficient algorithm for computing $CV(d)$ at location (x, y). We first define a *pivot point* as P whose location is $(\lfloor x \rfloor + 1 - L, \lfloor y \rfloor + 1 - L)$ and a *pivot VCR* as V_p which is defined as $(\lfloor x \rfloor + 1 - L, \lfloor y \rfloor + 1 - L, 2^0, 2^0)$. Namely, the bottom-left corner of V_p is at the pivot point P and V_p is a unit square. Then we use a pre-computed *difference array* D, which stores the differences of the ID's between two neighboring VCRs in a sorted $CV(\cdot)$, and the pivot VCR V_p to enumerate $CV(d)$. All the VCRs in $CV(d)$ can be efficiently computed at runtime by simple additions of the pivot VCR ID to each element stored in D.

3.5 Performance

Detailed simulations were conducted to study the performance of VCR-based query index under various conditions for event matching [19]. Here, we show a couple of important results to illustrate that a VCR-based query index can be used for stream processing.

The maximal VCR size has important performance impacts, both on the average search time and the total storage cost. Let the maximal size length of a VCR be L. Namely, $L_x = L_y = L$. If L is too small, a large number of small VCRs will be activated, increasing search time and storage cost. If L is too large, both the total number of VCRs and the size of a covering VCR set will be large, also increasing the search time and storage cost. The optimal L depends on the workload, especially the distribution of query sizes.

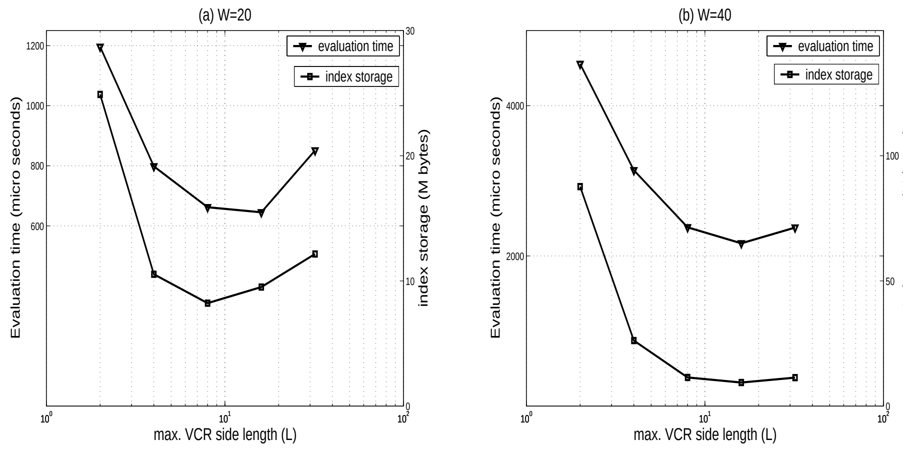


Fig. 12. The impacts of maximal VCR side length when (a) W=20; (b) W=40.

Fig. 12(a) and Fig. 12(b) show the impacts of the maximal VCR side length L on the average search time and storage cost when $W_x = W_y = W = 20$ and 40, respectively. Here, W_x and W_y are the maximal widths of a 2D range query in the X and Y attributes, respectively. Query widths were uniformly distributed between $[1, W_x]$ and $[1, W_y]$, respectively. For this experiment, a moderate R_xR_y of 60,000 was used. For both figures, the left y-axis was used for average search time and the right y-axis was used for the index storage requirement. For the case of $W = 20$, the optimal L is 8 in terms of storage cost. However, the optimal L is 16 in terms of average search time. For the case of $W = 40$, the optimal L is 16 in terms of storage and search time. In general, the larger the average size of query ranges is, the larger the optimal VCR side length becomes. For the experiments in the rest of the paper, we chose $L = 8$ for the case of $W = 20$ and $L = 16$ for the case of $W = 40$.

3.6 Comparison of VCR with R-tree

In this section, we compare VCR with an R-tree under a moderate R_xR_y of 60,000. Fig. 13(a) shows the average search times of VCR indexing and R-tree indexing with different degrees of query overlapping. Queries overlap more for the 90%-10% case and they overlap less for the uniform case. To model overlapping, the bottom-left corners of 90% of the range queries were randomly chosen from 10% of the monitoring region. Simple decomposition was used for the VCR indexing schemes.

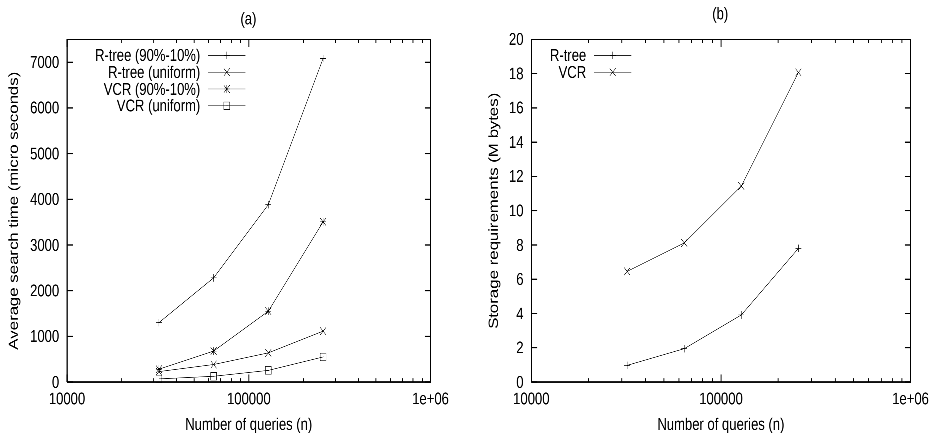


Fig. 13. Comparison of an R-tree and VCR in terms of (a) average search time; (b) storage requirement.

Fig. 13(a) shows that the search time of VCR is substantially better than R-tree, especially when queries are highly overlapping. For instance, even for a small n (32,000), the average search times are $1,300\mu$ and 304μ seconds for the 90%-10% and uniform cases, respectively. In comparison, the search times of VCR indexing are 280μ and 60μ seconds, respectively. For a large n, the performance difference between VCR and an R-tree is more substantial. Fig. 13(a) clearly shows that an R-tree is not effective for handling range queries that are highly overlapping. Note that, with highly overlapped queries, the minimum bounding rectangles in the internal nodes of an R-tree are highly overlapped. Thus, the search in an R-tree quickly degenerates into almost a full-tree traversal.

The efficient search time of VCR indexing in Fig. 13(a) is achieved with an increase in storage cost. However, such an increase is rather modest. Fig. 13(b) shows the corresponding storage requirements of an R-tree and VCR. Note that the storage requirement does not depend on the degree of query overlapping. Hence, we only show one line for each indexing scheme.

4 Related work

Various interval indexing approaches have been proposed, including segment trees, interval trees [17], R-trees [9], interval binary search trees (IBS-trees) [11] and interval skip lists (IS-lists) [12]. 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 [12], when there is heavy overlap among the intervals, the search time can degenerate rapidly.

Both IBS-trees and IS-lists were designed for main memory-based interval indexing [11, 12]. They were the first dynamic approaches that can handle a large number of overlapping query 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 query intervals. Moreover, as pointed out in [12], 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. This adjustment increases the insertion/deletion time complexity. For example, the insertion time complexity for IS-lists is $O(\log^2(n))$. More importantly, the adjustment makes it difficult to reliably implement the algorithms in practice. Previous studies [12] indicated that IS-lists are easier to implement compared with IBS-trees, even though dynamic adjustments of the interval skip lists are still needed. In contrast, no dynamic adjustment is needed in VCI indexing.

There are strong interests in event matching schemes for content-based pub/sub systems [1, 16, 7, 3, 21] and triggers [10]. There are roughly two kinds of pub/sub algorithms. The first kind of schemes consists of a single phase. For example, the Gryphon system builds a search tree with subscription predicate clauses [1]. However, no non-equality predicate clauses were considered in [1]. A second kind of event matching algorithms involves two phases [16, 7, 21]. The predicate clauses are tested in the first step, and then the matching subscriptions are computed using the results from the first step.

Recently, there has been research work on selective dissemination system that can handle subscriptions written in XPath for XML documents [2, 6, 4, 14]. XPath queries were converted into various data structures which react to XML parsing events. In contrast, the event matching problem discussed in this paper is for simpler subscriptions that contain conjunction of predicate intervals.

Continual queries [5, 15, 10] have been developed to permit users to be notified about changes that occur in the database. They evaluate conditions that combine incoming event with predicates on a current database state. This makes it difficult for these systems to scale over hundreds of thousands of queries because they must check hundreds of thousands of complex conditions each time a new event modifies the database state. In contrast, we focus on fast event matching for highly overlapping multidimensional range predicates.

Note that the VCRs defined in this paper are different from the space-filling curves, such as the Hilbert curve and the Z-ordering [8], that are used to store multidimensional point data. In a space-filling curve, the universe is first partitioned into grid cells. Each of the grid cells is labeled with a unique number that defines its position in the total order of the space-filling curve. The objective is to preserve spatial proximity in the original point data. In contrast, a set of VCRs is defined for each point. These VCRs are used to decompose queries, which are spatial objects such as rectangles. Furthermore, VCRs are not designed to partition the universe.

The SR-tree presented in [13] is the most related to the VCR indexing in that both are designed for multidimensional interval data. However, they are designed to handle different specific workload issues. SR-tree is to tackle skew in interval sizes while VCR indexing to tackle query overlapping. SR-tree is a modified R-tee. The emphasis of the SR-tree is to improve the performance of an R-tree under the workload where the interval sizes are skewed. There are large-sized intervals among otherwise small-sized intervals. By mixing large-sized intervals with small-sized ones, the minimum bounding rectangles used in an R-tree are unduly enlarged by the large-sized intervals. Enlarged bounding rectangles quickly degrade the performance of an R-tree index. The SR-tree moves the large-sized intervals from the leaf-nodes into the internal nodes of an R-tree. However, the SR-tree still does not solve the issue of interval overlapping among the small-sized intervals.

5 Summary

We have described a virtual construct-based query indexing method for efficient processing of numerous continual range queries in a streaming environment. The objective is to design a main memory-based query index that has a low storage cost and excellent search performance. These two properties are important in order to handle a rapid stream. A set of virtual constructs are predefined. Each VC has a unique ID and an associated query ID list. Each range query is first decomposed into one or more VCs. The query ID is then inserted into the query ID lists associated with the decomposed VCs. The VC-based query index provides an indirect and cost-effective way of pre-computing the search results. Search is performed indirectly via the VCs.

Two VC-based query indexing methods were described: a VCI-based index for 1D interval queries and a VCR-based index for 2D range queries. Various alternative VCs, and the decomposition and search algorithms were presented. Selective performance results in terms of average search time and total index storage cost were presented. These results demonstrate that indeed the VC-based query index has the properties of low storage cost and good search performance.

We continue to conduct research studying the VC-based query indexing method for data processing in a streaming environment. Specifically, we are looking for VCs that can further reduce the total index storage cost and search time. We are exploring various approaches to reducing the total number of VCs predefined and the number of covering VCs that may contain any given data point.

References

1. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. of Symp. on Principles of Distributed Computing*, 1999.
2. M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of Very Large Data Bases*, pages 53–64, 2000.
3. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving expressiveness and scalability in an Internet-scale event notification service. In *Proc. of Symp. on Principles of Distributed Computing*, pages 219–227, 2000.
4. C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. of IEEE Int. Conf. on Data Engineering*, pages 235–244, 2002.
5. J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 379–390, 2000.
6. C.-W. Chung, J.-K. Min, and K. Shim. APEX: An adaptive path index for XML data. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 2002.

7. 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 Int. Conf. on Management of Data*, 2001.
8. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.
9. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1984.
10. E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proc. of IEEE Int. Conf. on Data Engineering*, pages 266–275, 1999.
11. E. Hanson, M. Chaaboun, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 271–280, 1990.
12. E. Hanson and T. Johnson. Selection predicate indexing for active databases using interval skip lists. *Information Systems*, 21(3):269–298, 1996.
13. C. P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1991.
14. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of Very Large Data Bases*, 2001.
15. L. Liu, C. Pu, and W. Tang. Continual queries for Internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):610–628, July/Aug. 1999.
16. J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for Web-based publish/subscribe systems. In *Proc. of Int. Conf. on Cooperative Information Systems*, pages 162–173, 2000.
17. H. Samet. *Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.
18. N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of Very Large Data Bases*, 2003.
19. K.-L. Wu, S.-K. Chen, and P. S. Yu. VCR indexing for fast event matching for highly-overlapping range predicates. In *Proc. of 2004 ACM Symp. on Applied Computing*, 2004.
20. K.-L. Wu, S.-K. Chen, P. S. Yu, and M. Mei. Efficient interval indexing for content-based subscription e-commerce and e-service. In *Proc. of IEEE Int. Conf. on e-Commerce Technology for Dynamic E-Business*, Sept. 2004.
21. K.-L. Wu and P. S. Yu. Efficient query monitoring using adaptive multiple key hashing. In *Proc. of ACM Int. Conf. on Information and Knowledge Management*, pages 477–484, 2002.