RC23873 (W0602-086) February 9, 2006
Computer Science

IBM Research Report

On Range 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

On Range Query Indexing 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
phone: 914-784-6615

Abstract

To monitor a large number of continual range queries against a rapid data stream, each incoming data item should only be evaluated against relevant queries, not all the queries. Generally speaking, a main memory-based query index with a small storage cost and a fast search time is needed. In this paper, we study a 2D range query index that meets both criteria. It centers around a set of predefined, containment-encoded squares, or CES's. CES's are multi-layered, virtual constructs used to decompose range queries and maintain the query index. With containment-encoding, the search process is extremely efficient; most of the operations can be carried out by a simple logical-shift instruction. Simulations show that, with a small index storage cost, the CES-based query index substantially outperforms other alternatives in search time.

Keywords: Sensor Data Monitoring, Data Streams Processing, Query Indexing, and Continual Queries.

1 Introduction

Continual range queries can be issued to monitor a variety of conditions in many stream applications. For example, in a sensor network stream application, continual range queries can be used to monitor the temperature, humidity, flow of traffic and many other sensor readings. In a financial stream application, continual range queries can be created to monitor the changes in prices and volumes of company stocks or government bonds. In this paper, we focus on the efficient processing of continual 2D range queries that can be specified as rectangles, such as $(x_1 \leq X \leq x_2) \wedge (y_1 \leq Y \leq y_2)$, on two independent attributes X and Y from a data stream, possibly derived from various readings of a sensor network.

Processing continual range queries has become increasingly more difficult, if not impossible, as the stream rate continues to increase. CPU can quickly become the bottleneck. Data items may have to be dropped without processing, i.e., load shedding [2, 6, 18]. However, it is more

desirable that a system process as many data items as possible. Hence, it is important that each data item in the stream is processed against only the relevant queries, not all the queries.

One approach to quickly identifying relevant queries for processing is to use a query index [1]. Each data point in an incoming stream is used to search the query index to find the range queries containing the data point. This is referred to as the *stabbing query* problem [17], i.e., finding the range queries that are stabbed by a data point.

Though maybe conceptually simple, it is quite challenging to design an effective 2D range query index for stream processing, especially if the stream flows rapidly. Generally speaking, a main memory-based query index with a small storage cost and a fast search time is needed. Low storage cost is important so that the entire query index can be loaded into main memory, eliminating potential performance degradation due to paging. Fast search time is critical so that the system can handle a rapid stream. Existing spatial indexes [17], such as R-trees [8, 4], can be used to index range queries. However, most of them are disk-based and generally not suitable for stream processing because disk I/O's might be needed during a search operation, slowing down stream processing.

In this paper, we explore a new main memory-based query index for efficient stream processing. It is based on predefined *virtual constructs*, VC's. VC's are a set of well-defined constructs used as a reference framework to properly index arbitrarily-sized range queries. They are used to decompose range queries and maintain the query index. Each VC is associated with a query ID list, storing the queries covering that VC. For each incoming data point, search is conducted indirectly by first computing the covering VC's and then the covering queries.

There are three key challenges in designing an effective VC-based query index. (1) There should be no ambiguity on whether or not a range query really covers a data point. Otherwise, extra computation is needed to resolve the ambiguity. To eliminate ambiguity, a range query should be "perfectly" covered by one or more VC's. (2) Each range query should be decomposed into a small number of VC's in order to minimize the storage cost. (3) Each incoming data point should be covered by a small number of VC's in order to reduce the search time.

In the area of mobile computing, such as [10, 22], various query indexes were proposed to efficiently evaluate continual range queries over moving objects. These approaches can be considered as VC-based query indexing. By treating moving object locations as data points in a stream, these query indexes in mobile computing can be applied to index range queries in stream processing.

However, to the best of our knowledge, none of the previous work on VC-based query indexes

[1] Note that we are indexing the continual range queries, not the data in the stream.

meet all three challenges simultaneously. These indexes can be divided into two categories based on the VC sizes: *fixed-sized* [10] and *variable-sized* [22]. The simplest and most naive fixed-sized approach uses unit grid cells, UGC's. However, a UGC-based query index fails in the second challenge because a range query can be decomposed into a large number of UGC's, incurring a large storage overhead. In [10], cells of a fixed size $L \times L$, where $L > 1$, were proposed for query indexing. However, it fails in the first challenge because these fixed-sized cells cannot perfectly cover every range query. This leads to ambiguities on whether or not a query really covers a data point during a search. Search time suffers as a result. In [22], a variable-sized approach was proposed for query indexing. However, it fails in the third challenge because the number of VC's covering a given data point can be large, leading to increases in search time.

In this paper, we present a new range query index that simultaneously meets all three challenges. It belongs to the variable-sized category. It is based on a set of *containment-encoded squares*, CES's, predefined over the entire query space. There are multiple layers of virtual squares with different sizes. Virtual squares in two different layers exhibit containment relationships and such relationships are encoded in their ID's. Containment-encoding makes the search process extremely efficient because most of the operations can be carried out by a simple logical-shift instruction. Each range query is perfectly decomposed into one or more CES's. Hence, no ambiguity exists on whether or not a data point is covered by a range query. With variable-sized CES's, each query can be decomposed into a small number of CES's, reducing the index storage cost. Containment-encoded squares limit the number of VC's that can possibly cover any data point to a small number, effectively lowering the search time.

Note that the current paper is not the first one, or the only one, to use the concept of multi-layered, hierarchical squares. It has also been used for various other applications, such as spatial joins [11] and spatial data mining [20, 21]. However, to the best of our knowledge, the labeling of these multi-layered squares into containment-encoded squares, the use of CES's to decompose range queries for query indexing and the use of CES's to facilitate extremely fast search operations are new and unique. Our contributions can be summarized as follows:

- We presented a new CES-based range query indexing approach for efficient stream processing. It simultaneously meets all three challenges to effective VC-based query indexing. It is based on containment-encoded squares, which are virtual squares used to decompose range queries, maintain the query index and conduct efficient search operations.

- We compared qualitatively and quantitatively the CES-based query index with other alternatives. The results show that, with a small storage cost, the CES-based query index

significantly outperforms other alternatives.

The rest of the paper is organized as follows. Section 2 presents the details of the CES-based query index. Section 3 provides qualitative comparisons of the CES-based index with alternatives. Section 4 shows performance evaluation. Section 5 discusses related work. Finally, Section 6 summarizes our paper.

2 CES-based query indexing

The new CES-based query index aims to have both low storage cost and fast search time. It defines a set of containment-encoded squares to meet all three challenges. Here, we describe the system model, the definition and labeling of CES's, and the decomposition and the search algorithms.

2.1 System model

Individual data items in a stream are assumed to be relational tuples with well-defined attributes, e.g., network measurements, meta data records, sensor readings, and so on. Range queries are assumed to be specified as conjunctions of two intervals involving two attributes X and Y, i.e., a rectangle in a 2D space. For simplicity, assume that the attribute ranges are $0 \leq X < R$ and $0 \leq Y < R$, respectively. Note that the monitoring region need not be a square. It can be a rectangle. Query boundaries are assumed to be defined along the integer grid lines of X and Y [2]. However, data points from the stream can be any non-integer numbers. In this paper, we define a rectangle or a virtual construct with (a, b, w, h), where (a, b) is the bottom-left corner, w is the width and h is the height. To avoid duplication, we assume that a virtual construct (a, b, w, h) is defined as $\{(x, y)|(a \leq x < a + w) \wedge (b \leq y < b + h)\}$.

2.2 Containment-encoded squares (CES)

Fig. 1 shows an example of containment-encoded squares and their ID labeling. The CES's are defined as follows. First, we partition the entire $R \times R$ monitoring area into $(R/L)^2$ virtual square partitions, each of size $L \times L$. Here, we assume that $L = 2^k$ and L is the maximal side length of a CES. The $L \times L$ squares are called level-0 virtual squares. Then, we create k additional levels of virtual squares. Level-1 virtual squares are created by partitioning each level-0 virtual square into 4 equal-sized $L/2 \times L/2$ virtual squares. Level-2 virtual squares are

[2] If query boundaries are not integers, we can expand them to the nearest integers. The CES-based query index can still be used to first efficiently identify a set of candidates. Extra checking might be needed to find the final results from the candidates.

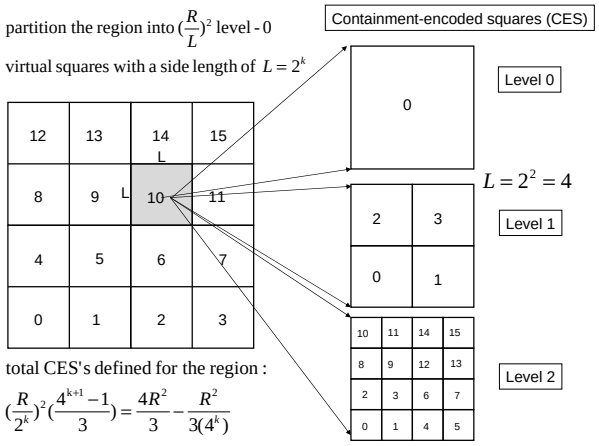

Figure 1: An example of containment-encoded squares (CES's).

created by partitioning each level-1 virtual squares into 4 equal-sized $L/4 \times L/4$ virtual squares. Level-k virtual squares have unit side length, i.e., 1×1.

The total number of CES's defined within each level-0 virtual square, including itself, is $\sum_{i=0}^{i=k} 4^i = (4^{k+1} - 1)/3$. These virtual squares are defined to have containment relationships among them in a special way. Every unit-sized CES is contained by a CES of size 2×2, which is in turn contained by a CES of size 4×4, which is in turn contained by a CES of size 8×8, $\cdots$, and so on.

Property 1 *The total number of CES's defined in an $R \times R$ monitoring region is* $(\frac{R}{L})^2 \sum_{i=0}^{i=k} 4^i = \frac{4R^2}{3} - \frac{R^2}{3(4^k)}$.

The total number of CES's defined is very close to that of UGC's defined, which is R^2. However, CES-based indexing does not suffer from high storage cost as much as UGC-based indexing. This is because large-sized CES's, e.g., $L \times L$, are available for decomposition. In contrast, only unit grid cells of size 1×1 are available in a UGC-based query index, causing each query to be decomposed into a large number of UGC's.

Now, we describe the labeling of CES's with containment encoding. A separate pointer array is used to map the 2D virtual squares at each level into a linear order. Hence, there are $k + 1$ pointer arrays for a CES-based query index.

Within each level, the ID of a virtual square consists of two parts: a partition ID and the local ID within the partition. If a virtual square has a partition ID p and local ID z_i, then its unique ID c_i at level i, where $0 \leq i \leq k$, can be computed as follows:

$$c_i = 4^i p + z_i. \tag{1}$$

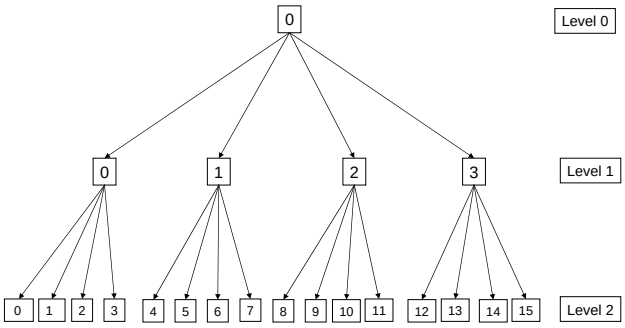

Figure 2: An example of a perfect quaternary tree and its containment-encoded labeling.

This is because there are 4^i CES's within each partition at level i. The partition ID can be computed as the row scanning order of the level-0 CES's starting from the bottom row and moving upwards. For example, for a level-0 CES (a, b, L, L), where (a, b) is the bottom-left corner and L is the side length, its partition ID can be computed as follows:

$$P(a, b, L, L) = \frac{a}{L} + (\frac{b}{L})(\frac{R}{L}). \tag{2}$$

The labeling of local CES ID's within a partition follows that of a perfect quaternary tree as shown in Fig. 2, where the ID's of the four child squares are $4s, 4s+1, 4s+2$ and $4s+3$ if the parent has a local ID s. In order to preserve containment relationships between virtual squares at different levels, the CES ID's within the same partition at each level follow the z-ordering space-filling curve, or Morton order [14, 13, 17]. For example, in Fig. 1, the ID's for the 16 level-2 virtual squares for partition 10 follow the z-ordering space-filling curve. In general, the local ID's of $4s, 4s + 1, 4s + 2$ and $4s + 3$ are assigned to the southwest, southeast, northwest and northeast children, respectively, of a parent virtual square with a local ID s.

Property 2 *For any CES at level i with a local ID z_i, where $0 < i \leq k$, the local ID of its parent can be computed by $\lfloor z_i/4 \rfloor$, or a logical right shift by 2 bits of the binary representation of z_i.*

Let $Z(a, b, 2^i)$ denote the local z-order of a CES $(a, b, 2^i, 2^i)$, where $0 \leq i < k$. $Z(a, b, 2^i)$ can be easily computed by a bit-shuffling procedure [14, 13]. We summarize this procedure as follows. Let $a_{k-1} \cdots a_1 a_0$ and $b_{k-1} \cdots b_1 b_0$ denote the least significant k bits in the binary representations of $\hat{a}$ and $\hat{b}$, respectively, where $\hat{a} = a - L\lfloor a/L \rfloor$ and $\hat{b} = b - L\lfloor b/L \rfloor$. Note that $\hat{a}$ and $\hat{b}$ are the local bottom-left corner of CES $(a, b, 2^i, 2^i)$ relative to that of its partition. $Z(a, b, 2^i)$ can be computed by interleaving the most significant $k - i$ bits of $a_{k-1} \cdots a_1 a_0$ and

```
Decomposition (a, b, w, h) {
  m = 1; Q = (a, b, w, h) ;
  while ((Q ≠ NULL) ∧ (m < L)) {
    remove leftmost column strip with width m, if any; split it into m × m CES's;
    remove topmost row strip with height m, if any; split it into m × m CES's;
    remove rightmost column strip with width m, if any; split it into m × m CES's;
    remove bottommost row strip with height m, if any; split it into m × m CES's;
    m = m × 2;
  }
  if (Q ≠ NULL) {
    divide Q into multiple L × L CES's;
  }
}
```

Figure 3: Pseudo code for decomposition algorithm with CES's.

$b_{k-1} \cdots b_1 b_0$ as follows: $b_{k-1} a_{k-1} \cdots b_i a_i$. As an example, assume $k = 2$, then $Z(10, 11, 2^0) = 14 = 1110_2$, the binary representation of 14. It is derived from bit shuffling of 10_2, the most significant 2 bits of 2 $(10 - 8 = 2)$, and 11_2, the most significant 2 bits of 3 $(11 - 8 = 3)$ (see Fig. 1). Similarly, $Z(10, 10, 2^1) = 3 = 11_2$.

Property 3 *The total number of CES's that can possibly cover/contain any given data point within the monitoring region is* $k + 1$.

2.3 Decomposition algorithm

Fig. 3 shows the pseudo code for decomposing a rectangle range query (a, b, w, h) into one or more CES's. It is a modification of a strip-splitting-based optimal algorithm for decomposing a query window into maximal quad-tree blocks [19]. The difference is that the algorithm in [19] allows m to be as large as R, assuming that $R = 2^r$, r is some integer, and R is the side length of the monitoring area. In contrast, we only allow m to be as large as $L = 2^k$, the maximal side length of a CES. The decomposition algorithm performs multiple iterations of 4 strip-splitting processes. During each iteration it tries, if possible, to strip away from Q a column strip or a row strip of width or height of $m = 2^i$, where $0 \leq i < k$, from each of the four outside layers of Q, starting with $i = 0$. The column strip or row strip is then split or decomposed into multiple $m \times m$ square blocks. The goal is to use minimal number of maximal-sized CES's to decompose Q. The entire strip-splitting process is like peeling a rectangular onion from the outside. The width of each layer at each successive iteration doubled until it reaches L. After that, it decomposes the remaining Q using $L \times L$ CES's.

During each iteration, the rule to determine if there is any strip of width or height 2^i that

can be removed from the remaining Q is based on the bottom-left corner, width and height of Q [19]. Assume that the current remaining Q is denoted as (a', b', w', h'), if $(a' \mod 2^{i+1}) \neq 0$, then a column strip of width 2^i, where $0 \leq i < k$, can be removed from the leftmost of Q. If $((b' + h') \mod 2^{i+1}) \neq 0$, then a row strip of height 2^i can be removed from the topmost of Q. If $((a' + w') \mod 2^{i+1}) \neq 0$, then a column strip of width 2^i can be stripped from the rightmost of Q. Finally, if $(b' \mod 2^{i+1}) \neq 0$, then a row strip of height 2^i can be removed from the bottommost of Q.

As an example to illustrate the strip-splitting-based decomposition, Fig. 4 shows the step-by-step decomposition of a range query Q defined as $(5, 2, 8, 12)$. (1) A column strip $(5, 2, 1, 12)$ of width 1 is stripped away from the leftmost outside of Q because $(5 \mod 2) \neq 0$. The remaining Q becomes $(6, 2, 7, 12)$. The column strip can be split into 12 CES's of size 1×1. (2) Another column strip $(12, 2, 1, 12)$ of width 1 can be stripped from the rightmost outside of the remaining Q because $((6 + 7) \mod 2) \neq 0$. This column strip again can be split into 12 CES's of size 1×1. After the stripping, the remaining Q becomes $(6, 2, 6, 12)$. (3) A column strip $(6, 2, 2, 12)$ of width 2 is removed from the leftmost outside of Q because $(6 \mod 4) \neq 0$. The remaining Q becomes $(8, 2, 4, 12)$. (4) A row strip $(8, 12, 4, 2)$ of height 2 is removed from the topmost outside of Q because $((2+12) \mod 4) \neq 0$. The remaining Q becomes $(8, 2, 4, 10)$. (5) A row strip $(8, 2, 4, 2)$ of height 2 is removed from the bottommost outside of Q because $(2 \mod 4) \neq 0$. The remaining Q becomes $(8, 4, 4, 8)$. (6) Finally, $(8, 4, 4, 8)$ is decomposed into two 4×4 CES's and the remaining Q becomes NULL. In total, 36 CES's are used to decompose the original Q. The query ID is inserted into the 36 associated query ID lists. In contrast, it would have required 96 unit grid cells to decompose the same range query.

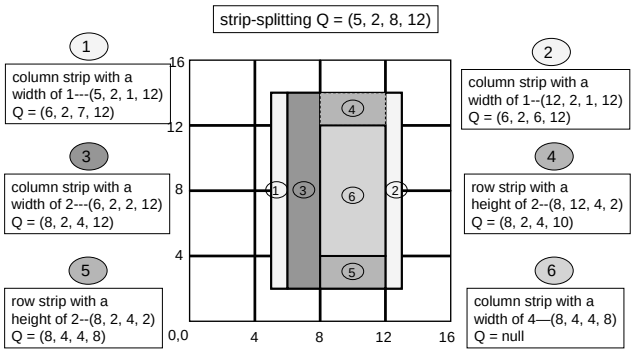

Figure 4: An example of strip-splitting-based decomposition with CES's.

```
Bottom-up Search(x, y) {
  I_x = ⌊x⌋; I_y = ⌊y⌋;
  P_x = ⌊I_x/L⌋; P_y = ⌊I_y/L⌋;
  // (LP_x, LP_y) is the partition bottom-left corner
  p = P_x + P_y(R/L); // partition ID
  z = Z(I_x − LP_x, I_y − LP_y, 2^0);
  // local ID of CES (I_x, I_y, 1, 1)
  for (l = k; l ≥ 0; l = l − 1) {
    c = 4^l p + z; // covering CES ID at level l
    if (IDlist(l, c) ≠ NULL) { output(IDList(l, c)); }
    z = z/4; // right shifts by 2 bits
  }
}
```

Figure 5: Pseudo code for a bottom-up search algorithm.

2.4 Search algorithm

For a given data point (x, y), the search algorithm finds the $k + 1$ CES's that contain or cover (x, y). Fig. 5 shows the pseudo code for a bottom-up search algorithm. It first finds the partition ID and the local ID of the level-k CES that contains (x, y). Let p denote the partition ID and z denote the local ID of the covering CES at level k. The unique ID of the covering CES at level k is $4^k p + z$. From Property 2, the local ID at level $k - 1$ can be easily computed by dividing z by 4 because of containment encoding. This can be implemented by a logical right shift by 2 bits. As a result, the entire search operation is extremely efficient. If the entire query index can be fully loaded into main memory, the CES-based query index can handle a very rapid stream.

Note that even though we partition each virtual square at level i into 4 equal-sized quadrants at level $i+1$, similar to the quad-tree space partition, the bottom-up search algorithm described in Fig. 5 makes the CES-based query index unique. Most of quad-tree-based search algorithms start from the root and hence are top-town approaches. The bottom-up search algorithm achieves efficient search by taking advantage of the containment encoding embedded in the local ID's of virtual squares at different levels.

3 Qualitative comparisons with alternative query indexes

Fig. 6 describes a taxonomy of alternative VC-based 2D range query indexes based on the sizes of VC's and whether or not containment encoding exists among the VC's. There are 3 categories: FS-NC, VS-NC and VS-CO. FS-NC uses fixed-sized VC's and no containment encoding exists among the VC's. Both unit grid cells and $L \times L$ cells [10], where $L > 1$,

Table 1: Qualitative comparisons of alternative VC-based 2D query indexes.

	Challenge 1	Challenge 2	Challenge 3	storage cost	search time
FS-NC(1); e.g., UGC	meets $\surd$	fails	meets $\surd$	high	fast
FS-NC(2); e.g., [10]	fails	meets $\surd$	meets $\surd$	low	slow
VS-NC; e.g., [22]	meets $\surd$	meets $\surd$	fails	low	slow
CES	meets $\surd$	meets $\surd$	meets $\surd$	low	fast

belong to the FS-NC category. VS-NC uses variable-sized VC's and no containment encoding exists among the VC's. The various VCR-based indexes presented in [22] belong to this VS-NC category. Finally, VS-CO uses variable-sized VC's and containment encoding exists among the VC's. The CES-based query index described in Section 2 belongs to the VS-CO category. Note that the FS-CO category is not applicable because containment relation can not exist among fixed-sized VC's.

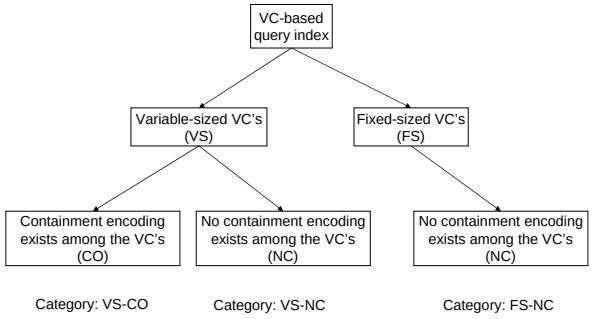

Figure 6: A taxonomy of alternative VC-based 2D range query indexes based on the sizes of VC's and whether or not containment encoding exists among the VC's.

In the following subsections, we describe examples of generic query indexes under the categories of FS-NC and VS-NC. For the FS-NC category, we show two alternatives: FS-NC(1) uses unit-sized, or 1×1, cells and FS-NC(2) uses $L \times L$, where $L > 1$, cells. Table 1 summarizes the qualitative comparisons of various alternative VC-based query indexes.

3.1 Fixed-sized and no containment encoding I (FS-NC(1))

The simplest VC-based range query index is based on *unit grid cells*, UGC's. The total number of virtual constructs defined is R^2. A range query (a, b, w, h) is decomposed into wh unit grid cells. This is referred to as an FS-NC(1) query index. In a UGC-based query index, there is no ambiguity on whether or not a query really covers a point. This is because every range query can be perfectly covered by one or more UGC's. However, the number of UGC's needed to

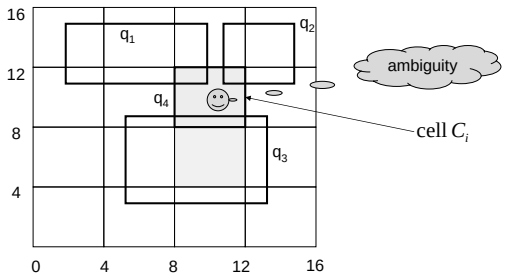

Figure 7: An example of a generic FS-NC(2) query index.

cover a range query can be large, especially if the area occupied by the range query is large. Hence, the storage cost for the query ID lists can be prohibitive. The number of covering VC's for any given data point (x, y) is exactly 1 and its ID can be computed as $\lfloor y \rfloor R + \lfloor x \rfloor$. The search performance hence is very fast if the entire query index can be loaded into main memory. If not, the search time can also be degraded due to page swapping effects.

3.2 Fixed-sized and no containment encoding II (FS-NC(2))

The high index storage cost of a UGC-based index can be reduced if $L \times L$ cells, where $L > 1$, are used. This is referred to as an FS-NC(2) query index. One of the query indexes proposed in [10] belongs to this category. Fig. 7 shows an example of a generic FS-NC(2) query index, where 4×4 fixed-sized cells are used. With larger-sized cells for decomposition, a range query requires fewer cells to cover it. Hence, storage cost is lowered. However, there is ambiguity on whether or not a query really covers a point. For example, in Fig. 7, queries q_1, q_2 and q_3 partially overlap with cell C_i. As a result, they may not cover a point inside cell C_i, creating ambiguities during a search with a data point. Even though the number of cells that cover a data point is only 1, boundary comparisons are needed to resolve the ambiguities, degrading the search performance. This is particularly true if the boundary definitions are stored in a file on disk.

3.3 Variable-sized and no containment encoding (VS-NC)

In a VS-NC query index, variable-sized VC's are used. Fig. 8 shows an example of a generic VS-NC query index using virtual construct squares, VCS's, similar to one of the various query indexes proposed in [22] for locating moving objects in mobile computing. For each unit grid point, $k + 1$ virtual construct squares (VCS's) are defined. These $k + 1$ VCS's all have its bottom-left corners at the same grid point. But, they have side lengths of $2^0, 2^1, \cdots, 2^k$, where $k = \log(L)$ and L is the maximal side length of a VCS.

Each query can be perfectly covered by one or more VCS's. Hence, there is no ambiguity on whether or not a query really covers a point. Moreover, because VCS's of difference sizes are available for decomposition, each range query can be covered by a small number of VCS's. Hence, the storage cost is low.

However, each data point can be covered by a large number of VCS's, leading to increases in search time. From Fig. 9, it can be shown that there are $(4L^2 - 1)/3$ VCS's that can cover a data point (x, y). This can be derived as follows. Assume that (x, y) is inside the unit grid cell in Fig. 9. Consider the bottom-left VCS with size $L \times L$ that covers the unit grid cell. This $L \times L$ VCS can be moved eastwards along the X-axis and upwards along the Y-axis. There are a total of L^2 possible positions where the bottom-left corner of the $L \times L$ VCS can be placed such that it still covers the unit grid cell. Similarly, for the VCS with size $L/2 \times L/2$, the number of possible positions is $(L/2)^2$. Hence, the total number of possible covering VCS's for any data point (x, y) is $L^2 + (L/2)^2 + \cdots + 1 = \sum_{i=0}^{k} (L/2^i)^2 = (4L^2 - 1)/3$. In contrast, there are only $k + 1$ CES's that can cover a data point, where $k = \log(L)$.

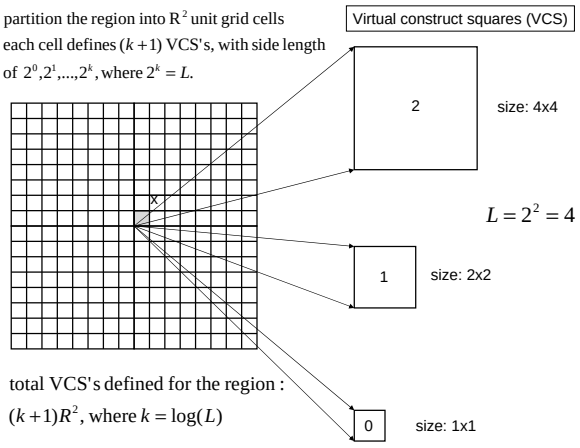

Figure 8: An example of a generic VS-NC query index using VCS's.

4 Performance evaluation

4.1 Simulation studies

Simulations were conducted to evaluate and compare quantitatively the CES-based index with generic indexes from the FS-NC(1), FS-NC(2), and VS-NC categories, respectively. We focused on the storage cost and search time because they are two of the most important metrics for a VC-based query index. A low storage cost is desirable so that the entire index can be loaded into

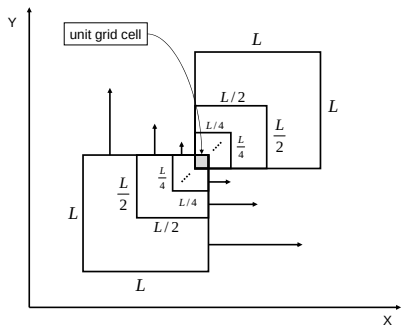

Figure 9: Many VCS's can cover a data point.

main memory. A fast search time is important so that a rapid stream can be properly handled. Note that, the search time, in terms of finding the ID lists from the covering VC's, of all the VC-based query indexes are independent of n, the total number of continual queries indexed. However, in our simulations, the search time included the report time, which involves reporting the query ID's stored in the associated query ID lists. As a result, it generally increases as n increases. This is because on average the ID lists contain more query ID's for a larger n. We implemented all four VC-based indexes. The simulations were conducted on an RS6000 model 43P machine running AIX 5.1.

The experiments were conducted under monitoring regions of different sizes. The bottom-left corner of a range query was randomly chosen from the monitoring region. Once the bottom-left corner was chosen, the width w and the height h of a range query were chosen uniformly and independently from $[1, W]$. Namely, $\bar{w} = W/2$ and $\bar{h} = W/2$. Various W's were used. The maximal side length L of a virtual construct was varied from 2 to 32. A total of n rectangle range queries were generated and inserted into the query index. In the experiments, n was varied from 1,000 to 64,000. After insertion, we performed about 50,000 to 100,000 random searches and computed the average search time. The storage cost included the pointer array(s) and the query ID lists maintained. The data points for search was a pair of floating point numbers chosen randomly and independently between 0 and R.

4.2 Impact of maximum side length (L)

We first examine the impact of the maximal side length L on the alternative query indexes in terms of storage cost and average search time. Note that, when $L = 1$, the FS-NC(2), VS-NC and CES-based indexes all degenerate into the UGC/FS-NC(1)-based index. Figs. 10(a) and (b) show the storage cost and average search time, respectively, of the four indexes when R^2 is relatively small ($R^2 = 65,536$). In contrast, Figs. 11(a) and (b) show the storage cost and

average search time, respectively, when R^2 is relatively large ($R^2 = 262,144$). For the entire set of experiments, a default W of 50 was used. The total number of range queries inserted was 16,000.

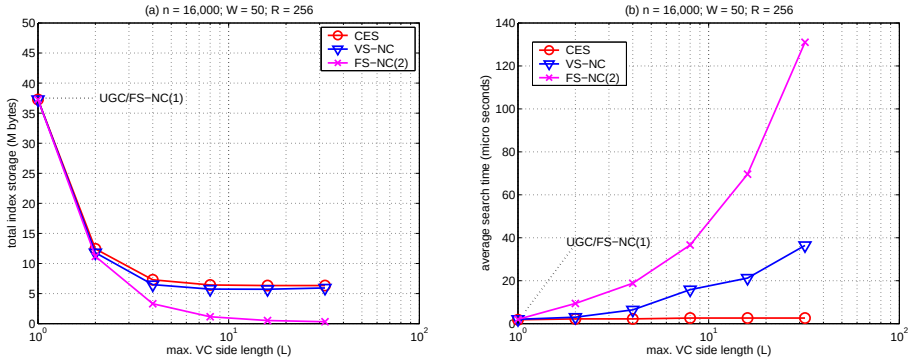

Figure 10: Impact of L, when R^2 is relatively small, on (a) total index storage cost; (b) average search time.

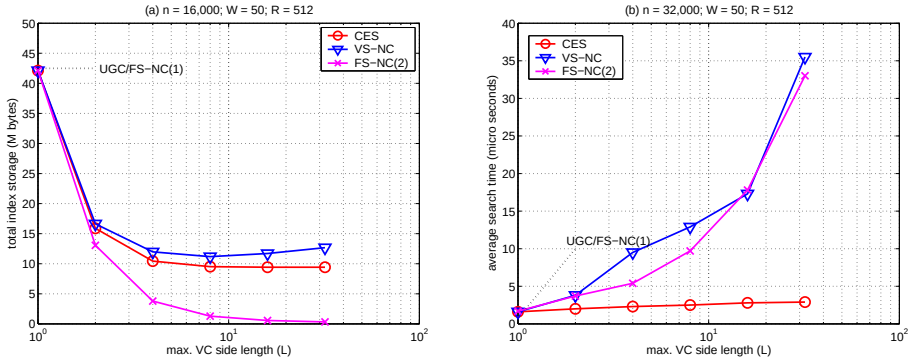

Figure 11: Impact of L, when R^2 is relatively large, on (a) total index storage cost; (b) average search time

As expected, the UGC/FS-NC(1) query index (the case with $L = 1$) has the fastest search time and the highest storage cost. The high storage cost can be effectively lowered by all three alternative indexes with a larger L. The lowest storage cost can be achieved by the FS-NC(2)-based index with a large L. Note that, in the computation of index storage costs in this paper, the query boundary definitions were not included for the FS-NC(2) index. Moreover, we assumed that these definitions are stored in memory as well for fast search time. The search time quickly increases for the FS-NC(2)-based index as L increases. In contrast, from $L = 2$ to $L = 32$, the CES-based index has a comparable search time performance as the UGC/FS-NC(1) index while incurring only a small fraction of the index storage cost. For example, for $L = 16$, the storage cost of the CES-based index is less than 14% and 24% of that of the UGC/FS-NC(1)

index in Fig. 10(a) and Fig. 11(a), respectively. These represent improvements of 86% and 76%, respectively.

In general, the average search time increases as L increases. This is particularly true for the FS-NC(2) and VS-NC query indexes because their search times increases far more quickly than the CES-based index as L increases. This is due to the failures of challenge 1 and challenge 3 by the FS-NC(2) and VS-NC query indexes, respectively (see Table 1). For the FS-NC(2) index, boundary comparisons dominate the search time even though there is only a single covering cell for any data point. As discussed in Section 3.3, there are $(4^{k+1}-1)/3$ VC's that can potentially cover a data point for the VS-NC index. In contrast, there are only $k+1$ covering VC's for the CES-based index.

Under all cases, the CES-based index performs significantly better than the FS-NC(2) and VS-NC query indexes in terms of search time. The advantage in average search time is particularly dramatic when L is large (see Figs. 10(b) and 11(b)). Note that the storage cost of the CES-based index generally decreases as L increases while the search time increases only in proportion to $\log(L)$. Hence, we can choose a large L, such as $L \approx W/2$ and $L = 2^k$, to lower the storage cost without a noticeable increase in search time.

4.3 The impact of search result size

From now on, we will focus on the comparisons among FS-NC(2), VS-NC and CES-based indexes. We assume that all three indexes use the same maximal side length L for their VC's.

Depending on the number of range queries and the distribution of these queries, the search result sizes can vary substantially. Search time is generally larger if a search result contains more query ID's because more ID's need to be reported. Fig. 12 shows the impact of search result size on search time. The search result size varies from about 120 to 200. For this experiment, $n = 16,000, L = 16, W = 50$ and $R = 256$. Because most of the search times for this experiment are less than 70μ seconds, we repeated the same point search for 100,000 times and compute the average. Fig. 12 plots the search times of 2000 random point searches for the FS-NC(2), VS-NC and CES-based indexes. It also plots the corresponding average search times from Fig. 10(b) as three different horizontal lines. In general, the search time is higher if the search result size is larger for all three indexes. Moreover, if the search result is spread over many query ID lists and they are far apart in the ID's of the covering VC's, the search time is higher for the same search result size. This is due to the effect of paging swap.

Note that there are gaps between the horizontal lines and the corresponding search times for the three indexes. This is due to the fact that the 100,000 repeated searches for a given data

point might cause the query ID lists containing the result to be cached in memory in Fig. 12. On the other hand, each data point was only searched once and no caching benefits can be realized in Fig. 10(b). Moreover, the caching benefit is most significant, i.e., the gap is widest, for the VS-NC index among the three because it has the largest number of VC's defined and hence the covering VC's may be spread most widely.

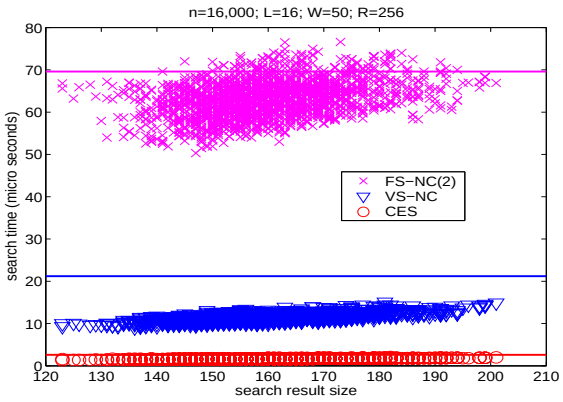

Figure 12: Impact of search result size on search time.

4.4 The impact of monitoring area

As R increases, the storage cost for the pointer array(s) becomes dominant in determining the total storage cost. This is because more virtual constructs will be defined. Figs. 13(a) and 13(b) show the total index storage and average response time, respectively, for the FS-NC(2), VS-NC and CES-based query indexes as R varies from 128 to 2,048. The storage cost of the VS-NC index becomes greater than that of the CES-based index when R becomes large because more VC's are defined. The storage cost of the FS-NC(2) index is the smallest. However, the search time is substantially higher than those of the VS-NC and CES-based indexes, respectively, especially when R is small. This is because, with the same number of queries, there are far fewer ambiguities to be resolved during a search for a large R than for a small R. Namely, the same number of queries is spread over a larger region and hence the number of queries that can partially overlap with a cell is smaller. More importantly, the average search time of the CES-based index remains very low for most of R values.

4.5 The impact of number of range queries (n)

Figs. 14(a) and 14(b) show the index storage cost and average search time, respectively, when R^2 is relatively large. For this set of experiments, W was 60, n was varied from 1,000 to 64,000 and $L = 8$.

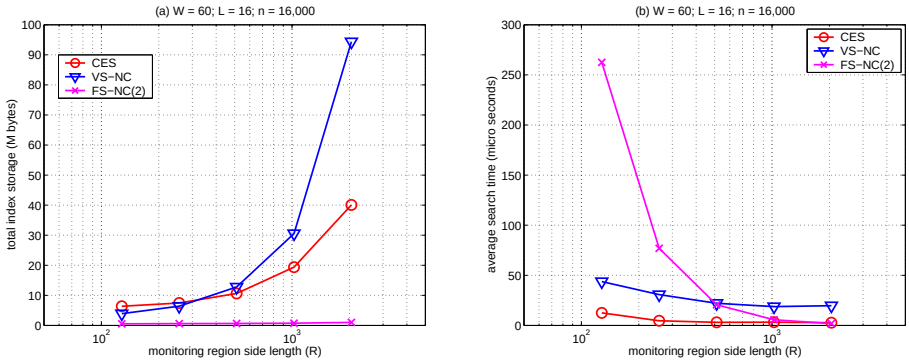

Figure 13: Impact of R on FS-NC(2), VS-NC and CES-based indexes in terms of (a) index storage cost and (b) average search time.

Once again, these figures show that, as n increases, both the storage costs and the search times of all three indexes tend to increase. The FS-NC(2) index has the lowest storage cost while the CES-based index has the fastest search time. Moreover, the search time of the CES-based index remains low even as n increases dramatically. From Fig 14(b), the average search time of the CES-based index outperforms the FS-NC(3) and VS-NC indexes by a wide margin, especially for a large n.

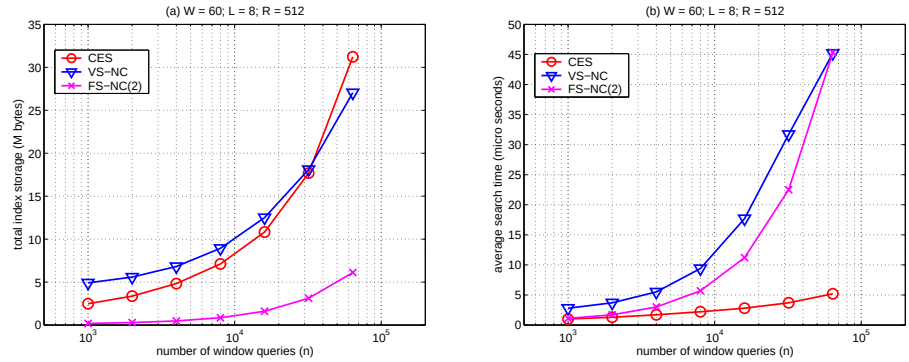

Figure 14: Impact of n, when R^2 is relatively large, on (a) index storage cost; (b) average search time.

5 Related work

Query indexing was also used in mobile computing to locate moving objects [5, 10, 12, 15, 22]. Some of the methods can be applied to the query indexing problem discussed in this paper. For example, the FS-NC(2) and VS-NC indexes were adapted from the cell-based [10] and the VCR-based [22] schemes, respectively, for moving objects. However, in addition to fast search

time, query indexing in mobile computing needs to address other challenges, such as allowing query processing to take advantage of incremental updates in object locations.

Although range queries can be treated as rectangles, traditional spatial indexing methods [17], such as an R-tree or any of its variants [4, 8], are not effective because they are mostly disk-based indexing methods. As shown in [10], R-trees is not as effective as the FS-NC(2) query index. Moreover, the performance of an R-tree quickly degenerates when the regions of range queries start to overlap with one another [4, 9].

Quad-tree decomposition has been studied for many applications, such as image databases, geographic information systems, and so on [1, 16, 17, 19]. A quad-tree index usually involves mapping the quadrants into a linear order, such as the z-order [17], and storing the quadrants in a B-tree. Because a B-tree is normally disk-based, a quad-tree-based query index is generally not as efficient for stream processing.

The decomposition algorithm of the CES-based query index is adapted, with some modifications, from the optimal quad-tree decomposition as described in [19]. This is possible because the definition of $\log(L) + 1$ levels of CES's within each square partition follows the quad-tree space partition. However, the CES-based approach is more like a combination of grid partition and quad-tree partition. First, grid partition is used to create level-0 CES's of size $L \times L$. Then, quad-tree partition is used to create $\log(L)$ additional levels of CES's. Because of grid partition, we can also handle a rectangular monitoring region. In contrast, a quad-tree partition always starts with a square region.

Multi-layered, hierarchical squares have also been used for other applications, such as spatial joins [11] and spatial data mining [20, 21]. However, they are used in different manners and for different purposes. Specifically, they are not labeled as containment-encoded squares to decompose range queries, to maintain a query index, or to facililate extremely fast search, as they are in this paper.

Classifying network packets based on a set of rules, referred to as the packet classification problem, has been extensively studied for Internet routing [3, 7]. Each rule specifies some criteria applied to the packet header. Typical rules involve longest prefix matching or range matching. Range matching on two header fields, such as source IP and destination IP, in packet classification is similar, to some extent, to the stream processing problem discussed in this paper. However, range matching in packet classification usually finds the top-priority rule matching a packet. In contrast, we return all the range queries that cover a data point. Moreover, each range rule is usually converted into a set of prefixes first and then a longest prefix matching algorithm is used to solve the range matching problem in packet classification [7].

Finally, a query index based on containment-encoded intervals, CEI's, has been proposed for efficient stream processing [23]. However, it deals with one dimensional interval queries. The labeling of containment-encoded intervals is derived from a perfect binary tree. In contrast, we discuss two dimensional rectangle queries in this paper. The labeling of containment-encoded squares is derived from a perfect quaternary tree.

6 Summary

We have presented a new CES-based query index for efficient processing of continual range queries against a rapid stream. Ideally suited for stream processing, it has two desirable properties: low storage cost and fast search time.

The CES-based range query index centers around a set of virtual containment-encoded squares. The entire monitoring area is first divided into equal-sized partitions of size $L \times L$, where $L = 2^k$ and L is the maximal side length of a CES. These square partitions are called the level-0 virtual squares. For each level-0 virtual square, we then define an additional k levels of virtual squares by successively subdividing each virtual square into 4 equal-sized virtual squares. The virtual squares at level k are similar to unit grid cells. These virtual squares are defined and labeled such that containment relationships are encoded in their ID's. Containment encoding makes the search process extremely fast because most of the operations can be carried out by a simple logical shift instruction.

Simulations have been conducted to evaluate the performance of the CES-based query index and compare it with other alternative query indexes. The results show that (1) The UGC/FS-NC(1) query index has a high storage cost but a very fast search time; (2) The CES-based query index substantially outperforms the FS-NC and VS-NC query indexes in search time, especially when n or W is large.

References

[1] W. G. Aref and H. Samet. Decomposing a window into maximal quadtree blocks. *Acta Informatica*, 30:425–439, 1993.

[2] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of IEEE ICDE*, 2004.

[3] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to CAMs? In *Proc. of IEEE INFOCOM*, 2003.

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

[5] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Motion adaptive indexing for moving continual queries over moving objects. In *Proc. of ACM CIKM*, 2004.

[6] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *Proc. of ACM CIKM*, 2005.

[7] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, pages 24–32, Mar./Apr. 2001.

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

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

[10] D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch. Efficient evaluation of continuous range queries on moving objects. In *Proc. of DEXA*, 2002.

[11] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. of ACM SIGMOD*, 1997.

[12] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. of ACM SIGMOD*, 2004.

[13] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. of ACM SIGMOD*, 1986.

[14] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. of ACM PODS*, April 1984.

[15] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. on Computers*, 51:1124–1140, Oct. 2002.

[16] G. Proietti. An optimal algorithm for decomposing a window into maximal quadtree blocks. *Acta Informatica*, 36:257–266, 1999.

[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 VLDB*, 2003.

[19] Y.-H. Tsai, K.-L. Chung, and W.-Y. Chen. A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks. *IEEE TKDE*, 16(4):519–523, Apr. 2004.

[20] W. Wang, J. Yang, and R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proc. of VLDB*, 1997.

[21] W. Wang, J. Yang, and R. Muntz. STING+: An approach to active spatial data mining. In *Proc. of IEEE ICDE*, 1999.

[22] K.-L. Wu, S.-K. Chen, and P. S. Yu. Processing continual range queries over moving objects using VCR-based query indexes. In *Proc. of IEEE MobiQuitous*, Aug. 2004.

[23] K.-L. Wu, S.-K. Chen, and P. S. Yu. Query indexing with containment-encoded intervals for efficient stream processing. *Knowledge and Information Systems*, to appear (online version available in May 2005, a preliminary version appeared in ACM CIKM 2004).