RC23063 (W0401-073) January 14, 2004
Computer Science

IBM Research Report

Efficient Interval Indexing for Content-Based Subscription E-Commerce and Services

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

Efficient Interval Indexing for Content-Based Subscription E-Commerce and Services

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

Abstract

This paper presents a dynamic interval index that, with a moderate storage cost, performs fast event matching against a large number of predicate intervals specified by content-based subscriptions. A set of *virtual construct intervals* (VCIs) is predefined, each with a unique ID. Each predicate interval is decomposed into one or more VCIs, which become activated by the predicate. The predicate ID is then inserted into the ID lists associated with the decomposed VCIs. To facilitate fast search, we start with a bitmap vector to indicate the activation of VCIs that cover an attribute value. Then, we study various techniques to reduce the storage cost, including *logarithmic* construct intervals (LCI) which reduce the total number of VCIs, *bitmap clipping* which prunes certain positions of a bitmap vector, and *bitmap virtualization* which eliminates the bitmap. Simulations are conducted to evaluate and compare these techniques.

Keywords: Subscription Services, E-commerce Enabling Technologies, Pub/Sub, Interval Indexing, Virtual Construct Interval, Event Matching, and Event Monitoring.

1 Introduction

Content-based subscription e-commerce and services in a large distributed environment have become popular in the advent of the Web [1, 11, 2, 12, 16]. Clients can subscribe to various content-based services, usually expressed via predicates on a set of attributes, with a provider on the Web. For example, a security analyst would like to be notified if the reading on a certain sensor reaches a specific level. The service provider monitors these predicates against continuously changing event data, such as the current stock prices, sensor readings, interest rates, or business activities. Once an incoming event is matched with a subset of the subscriptions, proper actions can be taken automatically. For example, alerts can be sent to them via e-mails or cellphones. Specific actions, such as buying or selling stocks, can also be triggered.

One of the most critical components of supporting large-scale content-based subscription services is the fast matching of events against the predicates. A large number of events

can occur in a short period of time. Each event must be matched against a large number of predicates, perhaps in the hundreds of thousands or even millions. Hence, an efficient event matching algorithm is required. Usually a main memory-based predicate index is needed [1, 3, 7, 8, 9, 16]. The index must support dynamic insertions and deletions of predicates as client interests are intermittently added into or removed from the system. The search complexity and the storage cost must be minimized. Furthermore, predicates may contain non-equality clauses, such as intervals. Interval predicates are particularly difficult to index in the face of dynamic insertions and deletions.

In this paper, we study the problem of efficient interval predicate indexing for supporting content-based subscription services. Formally, the goal is to support the finding of all predicate intervals in a set $Q = \{I_1, I_2, \cdots, I_n\}$ which cover an event value. This was referred to as a *stabbing query* problem [13]. It corresponds to finding all predicate intervals that match an event value. Such an event value could be the rate of network attacks in a hosted data center, the reading of a sensor, the price of a stock or the interest rate of a government bond. Predicate intervals, such as $[12, 19]$, $(8, 45]$, $(6, 100)$, $(-10, 3]$ and $[45, 200)$, represent user interests on certain attribute. For example, a user may want to be notified if the reading of a sensor falls within $[12, 19]$.

We present a new and novel interval indexing method, called *Virtual Construct Interval* (VCI) indexing. It is based on the following observation. For simplicity, assume that an attribute A has R distinct values. All predicate intervals defined on attribute A are drawn horizontally from the left endpoint to the right endpoint, as in Fig. 1(a). The stabbing query problem can be solved by drawing a vertical line at an event value. All the predicate intervals intersecting with this vertical line match the event value. For example, predicates $q1, q3, q7$ and $q9$ match a_i while $q2, q4, q6, q8$ and $q9$ match a_j in Fig. 1(a).

Though conceptually simple, it is quite challenging to quickly identify those predicate intervals intersecting with the vertical line of an event value. Without the help of an interval index, one must perform event matching via linear search. Event matching can be unacceptably slow, especially for a large number of predicate intervals. On the other hand, one can pre-compute and maintain the IDs of predicate intervals intersecting with the vertical lines at all possible event values. This is referred to as the direct listing approach. Event matching becomes extremely fast via a direct lookup. However, the storage cost will be prohibitive because each predicate ID is replicated by the number of event values it covers.

VCI indexing is an attempt to implement the idea shown in Fig. 1(a) in a storage cost-effective manner. We predefine a set of virtual construct intervals, each with a unique ID and specific endpoints. A predicate interval is first decomposed into one or more VCIs. A VCI is activated when a predicate interval using it in the decomposition is added into the system. The predicate ID is then inserted into the predicate ID lists associated with the activated VCIs in the decomposition.

Event matching becomes finding all the activated VCIs that intersect with the vertical line at an event value. We predefine the VCIs in such a way that the candidate VCIs can be systematically computed. Once activated VCIs are identified, the IDs of predicates matching the event are stored in the associated ID lists. To make event matching fast, we start with a bitmap vector to indicate the activation of VCIs that cover an event value. Then, we explore a series of techniques to reduce the storage cost of VCI indexing while maintaining its fast search time. These techniques include (a) reducing the total number of VCIs; (b) clipping each bitmap vector, i.e., pruning certain bit positions from a bitmap vector; and (c) making

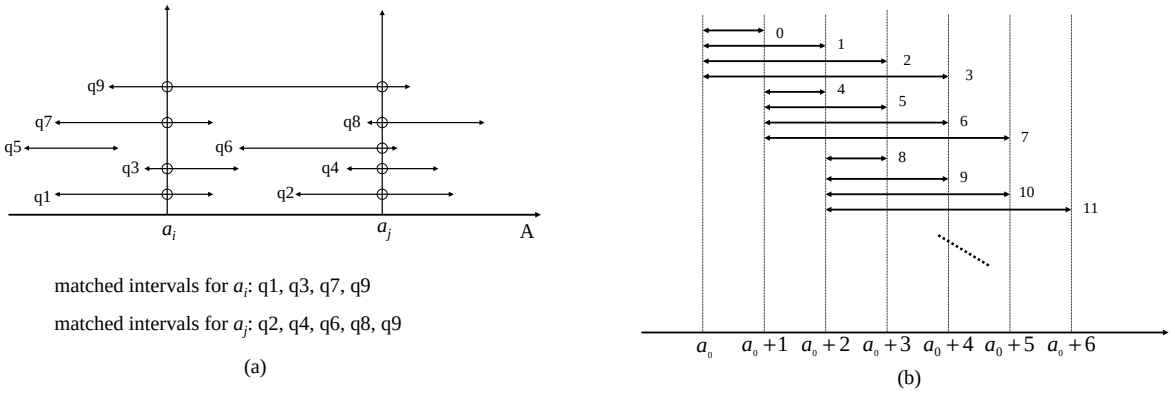

Figure 1: (a) Event matching against predicate intervals; (b) An example of simple construct intervals (SCI) with $L = 4$.

bitmap vectors virtual, i.e., eliminating them.

We conduct simulations to evaluate these different techniques for storage reduction. We also compare VCI indexing with a direct listing and a state-of-the-art IS-lists [9] approaches. The direct listing approach has $O(1)$ search time and $O(n\bar{w})$ storage cost, where $\bar{w}$ is the average interval length among the n predicate intervals. The IS-lists approach uses interval skip lists to implement a balanced dynamic search tree. It has $O(\log(n))$ search time and $O(n\log(n))$ storage cost. The results show that, compared with the IS-lists approach, VCI indexing with bitmap virtualization is better in both search time and storage cost for a large n, where n is the total number of predicates.

The paper is organized as follows. Section 2 presents two VCI indexing methods. Section 3 introduces bitmap clipping. Section 4 describes bitmap virtualization. Section 5 shows the performance studies. Related work is discussed in Section 6. Finally, Section 7 summarizes the paper.

2 VCI indexing

2.1 System model

For clarity, *predicate intervals* are user-defined intervals in the predicates of subscriptions. On the other hand, *virtual construct intervals* (VCIs) are virtual intervals that we use to decompose predicate intervals. Each VCI has a unique ID, called *interval coordinate* or simply *coordinate*.

We focus on simple predicates, where each contains a single range-based clause on an attribute.[1] The result is applicable to complex predicates, where each contains a conjunction of more than one clauses. For example, an efficient interval index can be maintained for each attribute in a two-phase event matching algorithm involving complex predicates, such as the ones presented in [3, 16].

We assume that all predicate intervals are defined on an attribute A. A can be of integer or non-integer data type. Predicate intervals include both endpoints, and the endpoints are

[1]We have also developed Virtual Construct Rectangles (VCR) for efficient predicate indexing for multi-dimensional range predicates [14].

integers.[2] Namely, we deal with the case that a predicate interval is represented as $p : [x, y]$, where p is the predicate ID, x and y are integer endpoints and $y > x$. Due to space limitation, event values are assumed to be integers in this paper. However, event values can be relaxed to be non-integers [15]. Moreover, the intervals can be open-ended or exclusive, and the interval length can be zero (i.e., $y = x$) [15]. We assume that predicate intervals fall between a_0 and $a_0 + R - 1$, i.e., the attribute range is R.

2.2 Simple construct interval (SCI) approach

In the SCI approach, we define L unique VCIs that start at each integer value and have respective lengths of $1, 2, \cdots, L$. The coordinates for the L VCIs that start at value $a_0 + j$ are $jL, jL + 1, \cdots, (j + 1)L - 1$, respectively, where $0 \leq j \leq R - 1$. Fig. 1(b) shows an example of VCIs and their unique IDs (coordinates) with $L = 4$ in an SCI approach.

To construct a VCI index, a bitmap vector B_j is allocated for each integer value $a_0 + j$, where $0 \leq j \leq R - 1$ (see an example in Fig. 3(a)). Each B_j has $N = RL$ bits, each representing a unique VCI. N is the total number of VCIs. Associated with each interval coordinate c, $0 \leq c < N$, we maintain a predicate ID list. An array H of size N is used to maintain a header pointer to each predicate ID list. H_c denotes the pointer to the predicate ID list for VCI c.

A predicate interval is first decomposed into one or more VCIs. These VCIs strictly cover the predicate interval. In other words, any attribute value covered by the predicate interval is also covered by at least one of the decomposed VCIs, and vice versa. The decomposition is very simple. We repeatedly use a VCI with length of L to cover a predicate interval from its left endpoint. Hence, a predicate $[x, y]$ is decomposed into $\lfloor \frac{y-x-1}{L} \rfloor$ VCIs with length L and a remnant.

Fig. 2 shows the algorithm for inserting a user-defined predicate interval $p : [x, y]$. Depending on the length of $p : [x, y]$, a simple decomposition may be needed. If $y - x > L$, we first decompose it into $m + 1$ VCIs, $c_0 : [x, x + L]$, $c_1 : [x + L, x + 2L]$, $\cdots$, $c_{m-1} : [x + (m - 1)L, x + mL]$ and $c_m : [x + mL, y]$, where $m = \lfloor \frac{y-x-1}{L} \rfloor$. Each of the first m VCIs has a length of L. On the other hand, if $y - x \leq L$, $p : [x, y]$ itself is one of the N VCIs and no decomposition is needed.

For each decomposed VCI, the coordinate c_j, $0 \leq j \leq m$, is computed via **find_coordinate()**. The coordinate c_j is then used to insert p into H_{c_j} via **insert_pid()** and to set the proper bits of certain bitmap vectors to 1 via **set_bitmap()**, if necessary.

The function **find_coordinate()** for the SCI approach is rather simple. Given any VCI $[a, b]$, its unique ID c is computed as follows:

$$c = (a - a_0)L + (b - a) - 1. \tag{1}$$

We can easily verify Eq. (1) using the example shown in Fig. 1(b). The function **insert_pid**(p, c_j) inserts p into the head of H_{c_j}. The function **set_bitmap**(c_j) sets proper bits to 1, if necessary. Assume $b_{i,j}$ represents the bit position j of B_i, where $0 \leq i < R$ and $0 \leq j < N$. It sets every b_{i,c_j} to 1 for $a \leq i \leq b$, where c_j is the coordinate of VCI $[a, b]$.

[2] 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 predicates. However, a final checking is needed to determine if the candidate predicates indeed match the event.

INSERT($p : [x, y]$, SCI) {
  if ($y - x > L$) {
    $m = \lfloor \frac{y-x-1}{L} \rfloor$;
    break $p : [x, y]$ down to $m+1$ VCIs, $c_0 : [x, x+L], \cdots, c_m : [x+mL, y]$;
    for ($j = 0$; $j < m$; j++)
      c_j = **find_coordinate**($[x + jL, x + (j + 1)L]$, SCI);
    c_m = **find_coordinate**($[x + mL, y]$, SCI);
  } else { $m = 0$; c_0 = **find_coordinate**($[x, y]$, SCI); }
  for ($j = 0$; $j \leq m$; j++) {
    if ($\mathrm{H}_{c_j} == \phi$) **set_bitmap**(c_j);
    insert_pid(p, c_j);
  }
}

Figure 2: Algorithm for inserting a predicate interval in SCI.

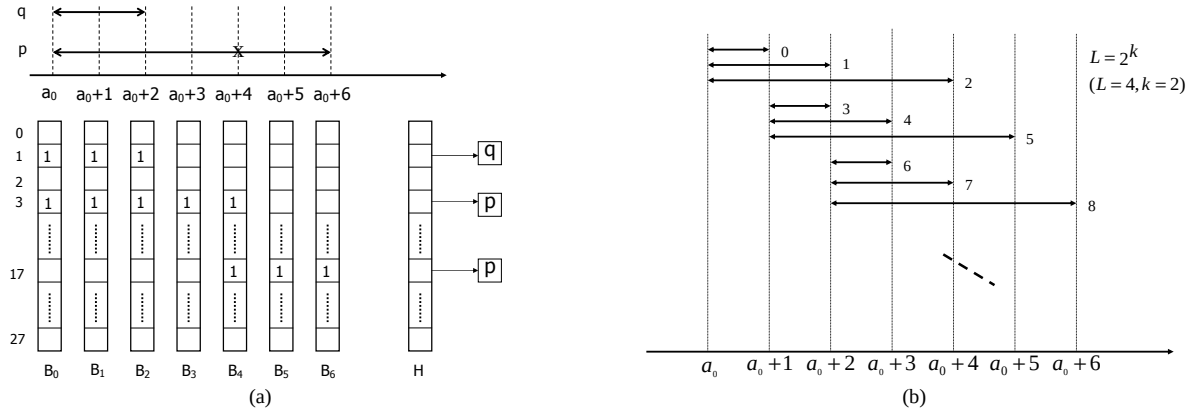

Figure 3: (a) An example of VCI indexing using SCIs with bitmap vectors; (b) An example of logarithmic construct intervals (LCI) with $L = 4$.

Note that **set_bitmap(c_j)** function needs to be executed only if H_{c_j} is ϕ before the insertion of p into H_{c_j}. Similarly, a corresponding **reset_bitmap(c_j)** needs to be executed only if H_{c_j} becomes ϕ after the removal of p from H_{c_j} in the deletion of a predicate interval from the interval index. This is due to the fact that b_{i,c_j} is used to indicate if construct interval c_j is activated by some predicate interval.

To illustrate the simplicity of **insert_pid()** and **set_bitmap()**, Fig. 3(a) shows an example of inserting two predicate intervals $p : [a_0, a_0 + 6]$ and $q : [a_0, a_0 + 2]$ using the SCI example in Fig. 1(b). Each B_i has 28 bits ($R = 7$ and $L = 4$). To insert q, we set $b_{0,1}, b_{1,1}$ and $b_{2,1}$ to 1 and insert q into H_1. In contrast, the length of p is greater than 4. We first break it down to two VCIs: $3 : [a_0, a_0 + 4]$ and $17 : [a_0 + 4, a_0 + 6]$. Hence, we insert p to H_3 and H_{17} and set $b_{0,3}, b_{1,3}, b_{2,3}, b_{3,3}, b_{4,3}, b_{4,17}, b_{5,17}, b_{6,17}$ to 1.

To find all the intervals that cover an attribute value s, we simply find all the H_t, where $b_{s,t} = 1$ for $0 \leq t < N$. The matched predicate interval IDs are stored in these ID lists. Note that $b_{s,t}$ is the bit that indicates if VCI t is activated. The search time is independent of the number of predicate intervals maintained in the system.

The deletion algorithm is very similar to the insertion algorithm. It differs in (a) the removal of p from the corresponding predicate ID lists and (b) the reset of proper bits when the last predicate interval that uses a VCI is deleted. Because every new p is inserted at the head of H_j, we need to traverse 1/2 of H_j on average to find p.

2.3 Logarithmic construct interval (LCI) approach

In contrast to SCI, the LCI scheme uses $\log(L) + 1$ VCIs beginning at each integer value. Assume $L = 2^k$, the lengths of virtual construct intervals in LCI are $2^0, 2^1, \cdots, 2^k$. Fig. 3(b) shows an example of an LCI with $L = 4$ and $k = 2$. Compared with the SCI example in Fig. 1(b), LCI has in general less number of VCIs for the same L.

The insertion algorithm for LCI is mostly similar to that for SCI. The only exceptions are the **find_coordinate()** function and the decomposition process for the case where the length of a remainder interval is less than L.

The **find_coordinate()** function for LCI is also rather simple. The coordinate c for any VCI $[a, b]$, where $b - a = 2^l$, $0 \leq l \leq k$, is computed as follows:

$$c = (a - a_0)(\log(L) + 1) + \log(b - a). \tag{2}$$

This is because there are $\log(L) + 1$ VCIs for each attribute value and the coordinate starts from 0.

If $y - x \neq L$, we may need to find more than one VCIs that can fully cover an interval whose length is less than L. For example, if $L = 16$, we need $[a, a + 4]$, $[a + 4, a + 6]$ and $[a+6, a+7]$ to cover an interval $p : [a, a+7]$. This is different from SCI where we can always find a single VCI when $y - x < L$.

The search algorithm for LCI is exactly the same as that for SCI. The deletion algorithm of LCI is similar to the insertion algorithm of LCI, except for the removal of p and the reset of proper bits when the last predicate using a VCI is deleted.

3 Bitmap clipping

Some positions for each bitmap vector will never be used. This is because certain VCIs will never cover an event value. For example, [4 10] will never cover any event value less than 3 or greater than 11. Hence, we do not need N bits for each bitmap vector. In this section, we use *bitmap clipping* to reduce the storage requirement for each bitmap vector B_j for attribute value $a_0 + j$, $0 \leq j \leq R - 1$. Bitmap clipping cuts unnecessary bit positions from the top and/or bottom of a bitmap vector. However, we need to map a bit position from a clipped bitmap vector to the actual coordinate. To facilitate bitmap clipping, we introduce the concept of a *covering segment*.

A covering segment of an integer value is defined as the minimal set of contiguous VCIs that can possibly cover the value. Note that some of the VCIs within a covering segment may not cover the value. We could eliminate them from the covering segment to further reduce bitmap storage. However, in so doing we need a more complex function to map a bit position to the actual interval coordinate and vice versa. For simplicity, we chose to keep the covering segment as a contiguous VCI segment.

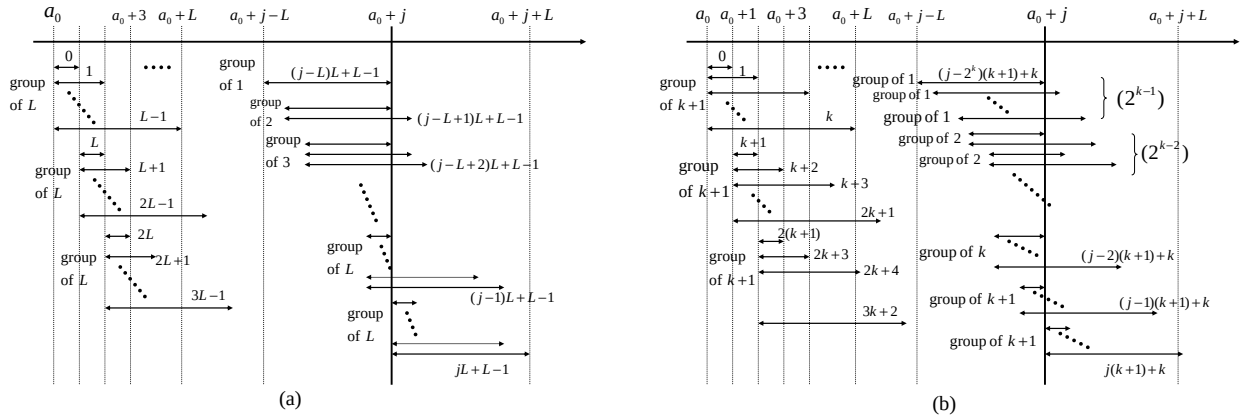

Figure 4: (a) Covering segment for an SCI scheme; (b) Covering segment for an LCI scheme.

Fig. 4(a) shows an example of the covering segment for the SCI scheme. On the left-hand side of Fig. 4(a), VCIs are grouped by their starting points. The set of VCIs that share the same starting point is viewed as a group of L intervals. Their coordinates are contiguous.

On the right-hand side of Fig. 4(a), we show all the VCIs that cover an attribute value $a_0 + j$ for $j \geq L$. At the top, there is a group of 1 VCI that begins at $a_0 + j - L$ and ends at $a_0 + j$. At the bottom, there is a group of L VCIs that all begin at $a_0 + j$ but end at different points. Note that the coordinates of the VCIs that do cover $a_0 + j$ may not be contiguous. However, all the IDs of VCIs that can possibly cover $a_0 + j$ fall into this covering segment range. The minimum VCI ID is $(j - L)L + L - 1$ and the maximum VCI ID is $jL + L - 1$.

Let $C_j^{\min}$ ($C_j^{\max}$) denote the minimum (maximum) ID of a VCI that covers attribute value $a_0 + j$. The covering segment of an attribute value $a_0 + j$, where $j \geq L$, is defined as the $(C_j^{\max} - C_j^{\min} + 1) = L^2 + 1$ consecutive VCI IDs beginning at $C_j^{\min}$. Note that the covering segment size is a constant and is independent of j for $j \geq L$.

The same covering segment concept can be applied to the LCI scheme. Fig. 4(b) shows the VCIs and their IDs that cover attribute $a_0 + j$ in an LCI scheme. Assuming $L = 2^k$, $C_j^{\min}$ is $(j - L)(k + 1) + k$ and $C_j^{\max}$ is $j(k + 1) + k$, for $j \geq L$. These two can be easily derived from Eq. (2). Hence, the size of the covering segment is $C_j^{\max} - C_j^{\min} + 1 = L(k + 1) + 1$, for $j \geq L$.

4 Bitmap virtualization

In VCI indexing, a bitmap vector B_j for attribute value $a_0 + j$ provides one major function. If a bit $b_{j,c}$ is 1, it indicates that all the predicate IDs pointed to by H_c cover attribute value $a_0 + j$ (see Fig. 3(a)). Without the bitmap vector, this piece of information is lost.

Fortunately, the activation of the VCI with coordinate c can be derived from the fact that H_c is not NULL. Remember that a predicate ID p is inserted into the list pointed to by H_c if the VCI with coordinate c is one of its decomposed VCIs. Hence, we can examine all H_c, for $C_j^{\min} \leq c \leq C_j^{\max}$, to see if a VCI within the covering segment of $a_0 + j$ is activated.

However, some of the VCIs whose coordinates fall within the covering segment of attribute value $a_0 + j$ may not cover $a_0 + j$. Hence, we need to know the right end point of a VCI from its coordinate to see if the VCI does cover the attribute value. This can be done by

a few computations. For example, to compute the right end point of a VCI $[a, b]$ from its coordinate c for LCI, we can use Eq. (2). First, we can compute a as $a_0 + \lfloor \frac{c}{k+1} \rfloor$, which involves an integer division and an addition. Then, we can compute b as $a + 2^l$, where l is an integer remainder of $c \div (k+1)$.

Note that the search time complexity remains at $O(1)$ with bitmap virtualization. For an event value j, we need to examine $C_j^{\max} - C_j^{\min} + 1$ header pointers. For SCI, this is $L^2 + 1$. For LCI, it is $L(\log(L) + 1) + 1$. Both are independent of n. If a pointer header is not NULL, then it needs extra computations to find the right end point of a VCI.

4.1 Minimizing computation

Let us define those VCIs that do cover $a_0 + j$ as the *covering VCIs* of $a_0 + j$. Let us also define a distance table, called DT_j, as the differences between all the coordinates of the covering VCIs of $a_0 + j$ and $C_j^{\min}$. Namely, with $C_j^{\min}$ and DT_j, we can reconstruct the coordinates of all the covering VCIs of $a_0 + j$.

There is an important property among the distance tables for all the attribute values $(a_0 + j)$'s where $L \leq j$. That is DT_i and DT_j are exactly the same even if $i \neq j$. This property can be easily observed from Figs. 4(a) and 4(b). Essentially, we can shift the entire drawings on the right-hand side of those two figures and the drawings 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.

Since DT_i and DT_j are exactly the same even if $i \neq j$, we can pre-compute, store and use a single DT to enumerate the coordinates of all the covering VCIs for any attribute value. In so doing, we can replace the more complex division operations with simple additions in bitmap virtualization. Instead of computing the ranges of a VCI from its coordinate and check if it covers an attribute value, we can now check if a covering VCI is activated. As will be shown in Section 5.4, the use of DT under bitmap virtualization can reduce the search time quite substantially. From Figs. 4(a) and 4(b), the size of the distance table, i.e., $\mid DT \mid$, can be derived as follows:

$$\mid DT \mid = \begin{cases} \sum_{i=1}^{L} i + L = \dfrac{L(L+3)}{2}, & \text{for SCI} \\ \sum_{i=1}^{k} i2^{k-i} + 2(k+1) = 2L + k, & \text{for LCI} \end{cases} \tag{3}$$

5 Performance evaluation

5.1 A direct listing approach

For comparison purpose, let us also look at a simple interval indexing scheme, called direct listing *Dlist*, that can achieve $O(1)$ search time. For each attribute value z, we simply maintain a direct ID list DL_z that contains all the IDs of predicate intervals that cover z.

To insert a predicate interval $p : [x, y]$, we insert p at the head of DL_z, for all $x \leq z \leq y$. The insertion time is proportional to the length of the interval $y - x + 1$, but it is independent of the number of intervals maintained. To search for all the intervals that cover any given data point z, we immediately have the answer and they are all linked in the list DL_z.

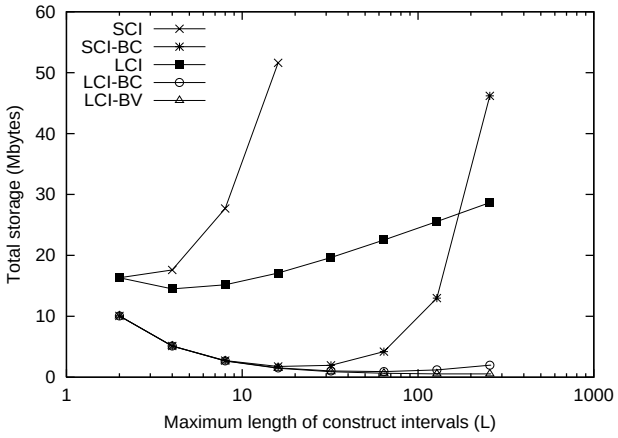

Figure 5: The impact of L on the total storage requirement.

To delete an interval $p : [x, y]$, we have to search each DL_z, where $x \leq z \leq y$, and then remove p from the lists. The delete time is proportional to the product of the interval length and the average size of the DL lists.

5.2 Simulation studies

Simulations were conducted to evaluate and compare various VCI indexing schemes. In the simulations, we implemented the different VCI indexing methods and the Dlist scheme. We also compared our schemes with a previous IS-lists approach [9]. The IS-lists approach has $O(\log(n))$ search time and $O(n \log(n))$ storage cost. For the IS-lists approach, we downloaded an implementation from the Web [6] and used it to run our input data.

Attribute values ranged from 1 to R with 5,000 as the default value of R for a moderate R. We also used a large R of 250,000 in the comparison of IS-lists and our approaches. The starting point of a predicate interval was chosen randomly between 1 and $R-1$ with a length w randomly chosen between 1 and W. Namely, $\bar{w} = \frac{W}{2}$. Different values of W were used. Since the maximum attribute value was R, any predicate interval $p : [x, y]$ where $y > R$ was reduced to $p : [x, R]$.

A total of n predicate intervals were generated and inserted. In the experiments, n was varied from 1,000 to 1,024,000. After insertion, we performed 10,000 random searches and computed the average search time and storage cost.[3]

5.3 Comparison of various VCI indexing schemes

We first compare the different VCI indexing schemes with bitmap clipping and bitmap virtualization. Namely, we compare SCI, SCI-BC, LCI, LCI-BC and LCI-BV schemes. In particular, we show the storage requirements of these five schemes with various Ls. For simplification, we only chose those Ls such that $L = 2^k$, starting from $L = 2$. Note that if $L = 1$ the five schemes degenerate into one that is similar to the Dlist scheme.

[3]Note that, the covering segment size and the distance table size are smaller for a non-integer event than for an integer event [15]. Search time for a non-integer event is in fact smaller than that for an integer event. Hence, we conducted our simulations based on integer event values.

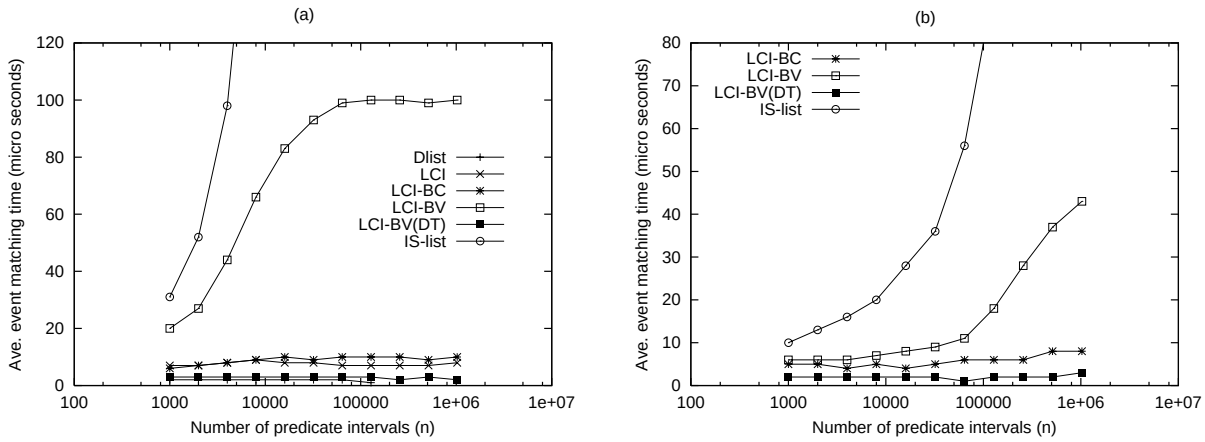

Figure 6: Comparison of LCI schemes with Dlist and IS-lists in event matching time with: (a) a moderate R (5000); (b) a large R (250,000).

Fig. 5 shows the total storage costs for the five different schemes. For this experiment, we chose $R = 5,000$, $W = 500$ and $n = 10,000$. We varied L from $2, 4, 8, \cdots, 256$. Logarithmic scale was used for the x-axis. Fig. 5 clearly demonstrates a few important results. (i) Bitmap clipping is critically important in lowering the storage requirement for both SCI and LCI schemes. (ii) The LCI approach is generally more effective than the SCI approach. (iii) With a relatively small L, e.g., ≤ 128, SCI-BC requires less storage than LCI. (iv) For a small or moderate R, LCI-BC and LCI-BV are hardly distinguishable and both are the best among the various schemes.

5.4 Comparison of LCI with IS-lists in search time

In this section, we compare the average event matching times of LCI, LCI-BC, LCI-BV and LCI-BV(DT) schemes with Dlist and IS-lists. In LCI-BV(DT), we pre-computed the distance table DT and used simple additions to locate the activated covering VCIs under bitmap virtualization. Fig. 6(a) we show the event matching times for the six different schemes. For this experiment, R was a moderate 5,000, W was 200 and L was 128. We varied n from 1,000 to 1,024,000.

As expected, Dlist is the best in terms of event matching time (very close to the x-axis). On the other hand, the event matching time for the IS-lists approach degrades quickly as n increases. In fact, the average search time exceeds 100 μ seconds when n is greater than 4,000. In contrast, the average event matching time remains less than 10 μ seconds for the LCI or LCI-BC schemes even when n is as large as 1 million.

The search time for LCI-BC is only a little bit higher than that for LCI. However, the average search time for LCI-BV can be much higher than that of LCI. This is because, in LCI-BC, we only need to perform a simple addition of $C_j^{\min}$ to the clipped bitmap index to obtain the coordinate of an activated VCI. In contrast, from Section 4, we need to perform division operations in LCI-BV, which takes more CPU time. As n increases, more and more VCIs are likely to be activated. Hence, there is a higher probability that H_c is not empty for a VCI c in the covering segment and the endpoint of c needs to be calculated as described in Section 4. Note that the search time for LCI-BV is bounded by the size of the covering

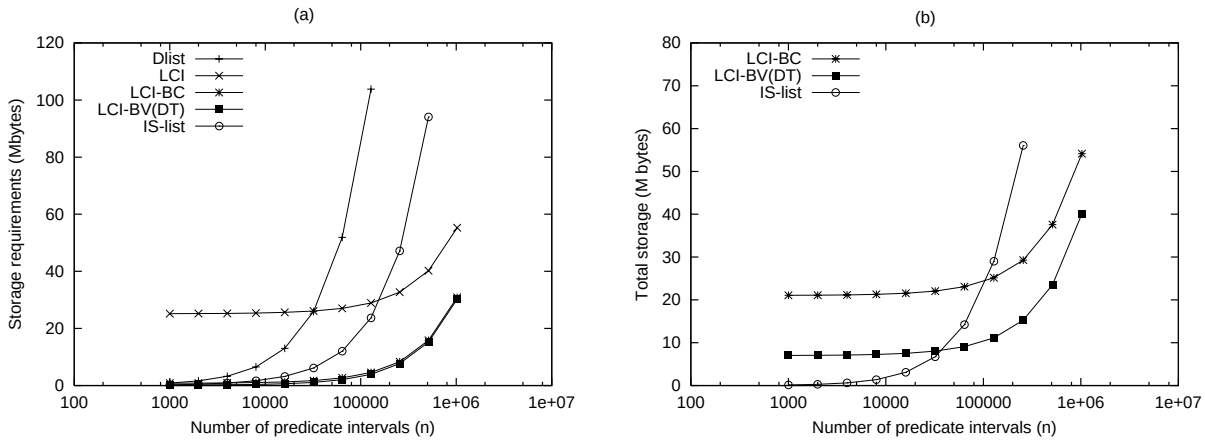

Figure 7: Comparison of LCI schemes with Dlist and IS-lists in storage cost with: (a) a moderate R (5000); (b) a large R (250,000).

segment (as shown in Fig. 6(a)). However, with a pre-computed distance table, the search time of LCI-BV(DT) becomes even better than that of LCI. This is due to the fact that bitmap manipulations required in LCI are more complex than the simple additions needed in LCI-BV(DT).

Fig. 6(b) shows the average event matching of LCI-BC, LCI-BV, LCI-BV(DT) and IS-lists approaches when R is a rather large 250,000. For this experiment, we did not show the results from Dlist and LCI because the storage costs are too high for them under most cases (see Section 5.5). Similar to Fig. 6(a), the average search time for IS-lists degrades quickly as n increases, albeit at a lower rate. Note that, in our experiment, the predicate intervals are randomly chosen within the attribute range. Hence, the search time is larger for a smaller R because more predicates need to be examined for a given attribute value. Fig. 6(b) also clearly shows that LCI-BV(DT) substantially reduces the search time for bitmap virtualization, especially for a large n.

5.5 Comparison of LCI with IS-lists in storage cost

In this section, we compare the storage costs of LCI, LCI-BC and LCI-BV schemes with Dlist and IS-lists. Fig. 7(a) shows the storage requirements for the five schemes as n increases from 1,000 to 1,024,000 for a moderate R (5,000). This was the same experiment we used to obtain the average event matching times in Fig. 6(a). The storage cost of LCI-BV (not shown explicitly) is similar to that of LCI-BV(DT) because the size of the distance table DT is negligibly small when compared with that of LCI-BV.

As expected, the storage cost of LCI is higher than the other four schemes for a smaller n because of the bitmap vectors. However, the storage cost of Dlist crosses over that of LCI when n is about 32,000. The storage cost of IS-lists also crosses over that of LCI when n is about 128,000. For the Dlist case, we can only run our experiment up to $n = 128,000$. Beyond that, our system (an RS 6000 system running AIX 4.3.3) ran out of memory. Similarly, we can only run our experiment up to $n = 512,000$ for the IS-lists case. However, we did not encounter the similar problem for the three LCI cases.

Fig. 7(a) shows that, without bitmap clipping or bitmap virtualization, the benefits of fast event matching time of LCI (see Fig. 6(a)) is achieved at the expense of more storage overheads, especially for a small n. However, with bitmap clipping or bitmap virtualization, the storage overhead is no worse than that of the IS-lists approach. In fact, for a moderate R, the storage overhead of LCI-BC or LCI-BV(DT) is smaller than that of IS-lists even for a small n. Moreover, LCI-BC and LCI-BV(DT) have a much better performance in average search time than IS-lists. (see Fig. 6(a)).

Fig. 7(b) shows the storage costs of LCI-BC, LCI-BV(DT) and IS-lists when R is a rather large 250,000. We did not show the case of LCI because the storage cost is too high for most of the cases. With a large R, the benefits of fast search time in LCI-BC and LCI-BV(DT) are achieved at the expense of higher storage overhead (see Figs. 6(b) and 7(b)). For example, the storage cost of IS-lists is lower than that of LCI-BC and LCI-BV(DT) for a small n in Fig. 7(b). However, as n increases, the storage cost of IS-lists crosses over that of LCI-BC or LCI-BV(DT).

Finally, bitmap virtualization combined with a pre-computed distance table, or LCI-BV(DT), is the most effective among the VCI indexing schemes. Compared with the IS-lists approach, LCI-BV(DT) is very effective, especially for a large number of predicates. This is true even if R is relatively large.

6 Related work

Various interval indexing approaches have been proposed, including segment trees, interval trees [13], R-trees [5], interval binary search trees (IBS-trees) [8] and interval skip lists (IS-lists) [9]. 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 [9], 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 [8, 9]. They were the first dynamic approaches that can handle a large number of overlapping predicate 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 predicate intervals. Moreover, as pointed out in [9], 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 [9] 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 other spatial indexing methods that can be used to handle one dimensional intervals [10, 4, 13]. 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 needed for the monitoring of events for content-based pub/sub, continual queries, or profile-based applications.

7 Summary

In this paper, we have studied a virtual construct interval (VCI) indexing approach for efficient interval predicate indexing, which is critical to large-scale content-based subscription services. Its goal is to perform fast event matching against a large number of interval predicates with a moderate storage overhead.

We introduced a new concept called virtual construct intervals. Each predicate interval is first decomposed into one or more VCIs. A VCI is activated when a predicate interval using it in the decomposition is added to the system. The predicate ID is then stored in the ID lists associated with the activated VCIs. Event matching is simple. The matched predicates are stored in the ID lists of the activated VCIs that cover the event value.

To facilitate fast search, we started with a bitmap vector for each attribute value to indicate the activation of VCIs that cover the value. Various techniques have been examined to reduce the storage overhead while maintaining the fast search time. First, the number of VCIs is reduced via the LCI approach. It reduces the bitmap size. Second, the bitmap is clipped, further reducing the bitmap size. Third, the bitmap is virtualized, i.e., eliminated. Finally, a pre-computed distance table is used to reduce the computation overhead needed after bitmap virtualization.

Simulations were conducted to compare the performance of various VCI indexing schemes and a state-of-the-art IS-lists approach. The results show that (1) LCI is generally more effective than SCI; (2) bitmap clipping is effective in reducing the storage cost when R is moderate; (3) the best VCI indexing scheme is to combine LCI with bitmap virtualization and a pre-computed distance table; (4) compared with the IS-lists approach, LCI with bitmap virtualization and a pre-computed distance table is better in storage cost and search time for a large number of predicates.

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] 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 applications. In *Proc. of VLDB*, 2002.

[3] 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 the ACM SIGMOD*, 2001.

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

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

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

[7] E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proc. of ICDE*, pages 266–275, 1999.

[8] 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.

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

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

[11] L. Liu, C. Pu, and W. Tang. Continual queries for Internet scale event-driven information delivery. *IEEE TKDE*, 11(4):610–628, July/Aug. 1999.

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

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

[14] 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.

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

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