Selection Predicate Indexing for Active Databases Using Interval Skip Lists

Eric N. Hanson
Theodore Johnson
Computer and Information Sciences Department
University of Florida
Gainesville, FL 32611
{hanson,ted}@cis.ufl.edu

TR94-017
15 April 1994
(revised 13 October 1994)

Abstract

A new, efficient selection predicate indexing scheme for active database systems is introduced. The selection predicate index proposed uses an interval index on an attribute of a relation or object collection when one or more rule condition clauses are defined on that attribute. The selection predicate index uses a new type of interval index called the *interval skip list* (IS-list). The IS-list is designed to allow efficient retrieval of all intervals that overlap a point, while allowing dynamic insertion and deletion of intervals. IS-list algorithms are described in detail. The IS-list allows efficient on-line searches, insertions, and deletions, yet is much simpler to implement than other comparable interval index data structures such as the priority search tree and balanced interval binary search tree (IBS-tree). IS-lists require only one third as much code to implement as balanced IBS-trees. The combination of simplicity, performance, and dynamic updateability of the IS-list is unmatched by any other interval index data structure. This makes the IS-list a good interval index structure for implementation in an active database predicate index.*

1 Introduction

Efficient testing of rule predicates is critical for good performance of active database systems and other forward-chaining rule systems. This paper presents an investigation of an important part of the rule condition testing problem that many active database systems and inference engines face: testing a collection of predicates to see which of the predicates match a single data object. In database terminology, which is used hereafter, this is the problem of testing an object (or tuple) to see which members of a collection of single-object (single relation) selection conditions match it. Extensive research has been done on processing rule conditions efficiently in inference engines for developing knowledge-based systems, such as OPS5 [4], KEE [20] and ART [21], and in active databases, including Ariel [16], POSTGRES [28, 32, 29], DATEX [3], RPL [7], Starburst [33], HiPAC [6] and others. Some of the work on inference engines and active databases has

*The C++ source code for interval skip lists can be obtained by sending a request by e-mail to hanson@cis.ufl.edu, or from http://www.cis.ufl.edu/~hanson/IS-lists/

lead to the development of the Rete algorithm [12], a modified version of Rete called TREAT [24], extensions to the Rete algorithm to exploit parallelism [22], and an optimized generalization of Rete and TREAT called Gator [17]. All these algorithms can be enhanced by speeding up testing of selection predicates.

The problem of determining which of a set of predicates match an object arises independent of the type of data or knowledge model used. For example, facts in OPS5 are like relational tuples, whereas in KEE and ART, facts are "frames" which are similar to objects in an object-oriented database. Hereafter, relational database terminology will be used for uniformity.

The Rete, TREAT and Gator algorithms also provide ways to test join conditions of rules. This paper does not address the issue of how join conditions will be processed. Efficient ways to determine which single-relation selection predicates match every new and modified tuple are important because this match must be done regardless of how the join clauses of rule conditions are tested.

The predicate testing problem in a database rule system is defined as follows. A given database contains a set of n relations, R_1 ... R_n, and m active database rules (triggers), r_1 ... r_m. Rules are of the form

on *event*
if *condition*
then *action*

Rules of this type are called event-condition-action (ECA) rules [6], and are the most popular type of rules being proposed in active database research at the time of this writing. Consider this relation schema:

EMP(name, age, salary, dept, job)

An example of the event and condition parts of an ECA rule on this EMP relation in the Ariel rule language [16] is:

```
define rule r1
on append to emp
if 20000 < emp.salary and emp.salary <= 30000
then ...
```

In Ariel, the approach used is to transform the event and condition parts of the rule into a single logical "condition." Internally, the condition of such a rule is treated as:

```
emp.EVENT = "append" and 20000 < emp.salary and emp.salary <= 30000
```

Prior to testing rule conditions, Ariel tags tuples with an event identifier describing the event just performed on the tuple. Testing for occurrence of an event is equivalent to testing this event identifier's value with an ordinary condition test. This paper assumes that such a transformation is made (it is possible for many

types of ECA rules). The remainder of this paper focuses on efficient testing of the *condition* of an active database rule, but because a transformation like the one just described is possible in many cases, the results are applicable for a large class of ECA rules.

In general, a rule condition can be an expression containing a conjunction of selection conditions and joins (projection is not allowed in rule conditions). Considering only the selection conditions of the rules, there is a collection of k single-relation predicates, P_i, $1 \leq i \leq k$. Each predicate restricts one or more attributes of a tuple t from a relation R_j, where $1 \leq j \leq n$. It is assumed that any predicate containing a disjunction is broken up into two or more predicates that do not have disjunction, and these predicates are treated separately. For purposes of predicate indexing, predicates are thus conjunctions of a subset of the restrictive clauses in the set $\{C_1(t), C_2(t), ...C_j(t)\}$.

The general form of a predicate P defined on any tuple t from a relation R_j for purposes of this discussion is a conjunction of this form:

$$P(t) = C_{i_1}(t) \wedge C_{i_2}(t) \wedge ...C_{i_m}(t)$$

The number of conjunctive clauses in the predicate is m, and m can vary.

Let ρ_1 and ρ_2 be elements of the set $\{<, \leq\}$. Each conjunctive clause $C_k(t)$, where $1 \leq k \leq j$, has one of the following forms:

Interval: *constant*1 ρ_1 t.attribute ρ_2 *constant*2

Equality: t.attribute = *constant*

General function: function(t)

The values *constant*, *constant*1 and *constant*2 are drawn from the domain of legal values for t.attribute. Open intervals, or ranges, can be formed by letting *constant*1 be $-\infty$ or *constant*2 be $+\infty$, e.g., t.attribute ≥ 1000 is equivalent to $1000 \leq t$.attribute $\leq +\infty$. For predicate clauses that are general functions, nothing is assumed about the function except that it returns true or false.

Some database languages may not allow interval predicates. However, systems implementing these languages could easily construct interval predicates from range predicates or conjunctions of range predicates. So the techniques discussed here are relevant even for database languages that don't allow interval predicates to be specified explicitly.

Some example predicates are shown below:

EMP.salary $<$ 20000 and EMP.age $>$ 50

20000 $\leq$ EMP.salary $\leq$ 30000

EMP.job = "Salesperson"

IsOdd(EMP.age) and EMP.dept ="Shoe"

In the last predicate above, IsOdd is a function that returns true if its argument is an odd number, and false otherwise.

Given a collection of conjunctive predicates as described above, and a tuple t, the predicate testing problem is to determine exactly those predicates that match t. One approach to testing predicates is to use a predicate index. Several approaches to the predicate indexing problem have been developed. In what follows, these methods are discussed in order of increasing complexity, some pragmatic considerations regarding interval indexing in databases are given, and then the interval skip list is introduced, along with a new and efficient predicate indexing strategy based on the interval skip list.

2 Review of Predicate Indexing Methods

Predicate indexing methods range from simple sequential testing to use of complex geometric data structures. Below, some alternatives that have been proposed for predicate indexing in a DBMS are listed in order of increasing complexity. Parallel predicate indexing methods are not considered since the primary focus here is a fast uniprocessor implementation. For each method, a discussion is given regarding what is done when a database object is modified, inserted, or deleted.

2.1 Sequential Search

In this method, the system traverses a list of predicates sequentially, testing each against the tuple. This has low overhead and works well for small numbers of predicates, but clearly performs badly when the number of predicates is large.

2.2 Hash on Relation Name Plus Sequential Search

In this method, the system maintains one list of predicates for each relation, and for each tuple modified, hashes on relation name to locate the predicate list for the tuple. The predicates on the list are then tested against the tuple sequentially. This is essentially the algorithm used in many main-memory-based production rule systems including some implementations of OPS5 [13, 24]. The algorithm performs well when the average number of predicates per relation is small, and the predicates are distributed evenly over the relations.

2.3 Physical Locking

This method, discussed in [30, 31], involves treating a predicate clause like a query, and running the standard query optimizer to produce an access plan for the query. If the resulting access plan requires an index scan, then special persistent markers (locks) are placed on all tuples read during the scan, and all index intervals inspected during the scan. If the resulting access plan is a sequential search, then "lock escalation" is performed, and a relation-level lock is placed on the relation being scanned. When a tuple is modified or inserted, the system collects locks that conflict with the update (i.e. all relation level locks, any locks that conflict with any indexes that were updated, and any other locks previously on the tuple). For each of the locks collected, the system tests the tuple against the predicate associated with the lock.

This algorithm has the advantage that no main-memory is needed to hold a predicate index, so theoretically, a very large number of rules can be accommodated. In addition, the algorithm makes use of the standard indexes and query processor to index predicates. However, there are disadvantages to the approach. In particular, when there are no indexes, or a large number of predicate clauses lie on attributes which do not have an index, most predicates will have a relation-level lock. This degenerate case requires sequentially testing a new or modified tuple against all the predicates for a particular relation, resulting in bad worst-case performance when the number of predicates is large. Also, the set of predicates must be stored in main memory to avoid costly disk I/O to test a tuple against a predicate when a lock for that predicate is found. This negates some of the memory-saving advantages of the algorithm. In addition, the need to set locks on index intervals and on tuples complicates the implementation of storage structures.

2.4 Multi-dimensional indexing

This technique stores a collection of predicates in a multi-dimensional structure designed for indexing region data. Applicable indexes include the R-tree [15] and R+-tree [30] . Predicates are treated as regions in a k-dimensional space (where k is the number of attributes in the relation on which the predicates are defined), and inserted into a k-dimensional index. Each new or modified tuple is used as a key to search the index to find all predicates that "overlap" the tuple. This technique works well when most predicates are small closed regions in the space defined by the schema of the relation from which tuples are drawn. However, real relational database applications often involve relations with anywhere from one to over 100 attributes, with a large fraction of relations having from 5 to 25 attributes. Typical predicates on these relations (e.g. single-relation selection conditions in WHERE clauses of queries, and also trigger conditions) will normally refer to only one or two attributes, and rarely more than five. Low-dimension predicates like these often

will not be small closed regions. Rather, they will often be "slices" through space that overlap extensively. Spatial data structures such as R-trees and R+-trees cannot be expected to perform well when indexing overlapping and open-ended regions like these.

3 Practical Considerations for Predicate Indexing in a DBMS

We envision that applications built using active database rule systems will be primarily data management applications, enhanced with rules to provide improved data integrity, monitoring and alerting capability, and some features similar to those found in expert systems. Database rule system applications will have to handle large volumes of data (perhaps millions of records). However, we expect that the number of rules in the majority of database rule system applications will be small enough that the set of rules and data structures for rule selection condition testing will be small enough to fit in main memory. We believe that this assumption is reasonable because rules are a form of intentional data (schema) as opposed to extensional data (contents). Moreover, the largest expert system applications built to date have on the order of 10,000 rules [2], which is few enough that data structures associated with the rules will fit in a few megabytes of main memory. More typical rule-based system applications have on the order of 50 to 1000 rules.

It is possible to concoct hypothetical applications where a tremendous number of rules are used, more than can fit in a main-memory data structure. Normally, rules in such applications have a very regular structure. This regular structure can be exploited to redesign the application so that only a few rules are used in conjunction with a much larger data table. The rules then use pattern matching to extract data from the table. For example, consider an application for stock reordering in a grocery store. The store might have 50,000 items for sale, with a relation ITEMS containing one tuple for each item. One way to implement the application would be to have one rule for each item to test whether the stock of the item is below a re-order threshold. An alternative way to implement the application would be to add a field to the ITEMS table containing the re-order threshold, and a single rule which compares the current stock level to the re-order stock level. This second implementation is clearly preferable.

It is standard practice in programming expert systems to put as much of the knowledge as possible into "facts" (e.g. frames, objects or tuples) and as little as possible into rules. This is done because knowledge structures are more regular and easier to understand than rules. This practice will be even more important in database rule system applications, where most of the "knowledge" should be stored in the database, with minimal use of rules.

The above discussion is a partial justification for building a carefully tuned main-memory predicate index

to test selection predicates of rules. Just such a predicate index is discussed in the next section. If the number of rules in an active database is extremely large, the predicate index presented here might not fit in main memory, and hence would not perform well. The selection predicate index presented here is not designed to handle such cases. But we believe it will perform well for the vast majority of foreseeable applications. Application developers will, of course, need to be aware of its limitations, i.e. the practical number of rules that can be handled is limited my the main-memory available to store the selection predicate index. This means that thousands of rules could be handled but not millions given current main memory sizes.

4 A High-Performance Predicate Indexing Method

This section introduces a predicate indexing method tailored to the problem of testing rule selection conditions in a database rule system. The task the algorithm must perform is, given a set of single-relation selection predicates as described earlier, return a list of all the predicates that match a tuple t from a specified relation R. The algorithm should have the following properties:

1. the ability to support general selection predicates composed of a conjunction of clauses on one or more attributes of a relation,
2. fast predicate matching performance,
3. the ability to rapidly insert and delete predicates on-line.

In the algorithm proposed, the system builds an index which has at the top level a hash table, using relation names as keys. Each entry in the table contains a pointer to a second-level index for each relation. This index maintains a list of non-indexable predicates. In addition, the second-level index contains a set of one-dimensional indexes, one for each attribute of the relation for which one or more indexable predicate clauses have been defined. This one-dimensional index is an interval skip list which allows efficient searching to determine which interval and equality predicates match an attribute value of a tuple. For predicates that are a conjunction of selection clauses, if there is an indexable clause, the most selective one is placed in the interval skip list (selectivity estimates are obtained using statistics from the system catalogs). A diagram for this indexing scheme is shown in Figure 1. In addition to this index, there is a main-memory table called PREDICATES that holds the predicates. When a partial match between a tuple t and a predicate P is found, P is retrieved from PREDICATES and tested against t to see if there is a complete match. Interval skip lists are discussed in detail below.

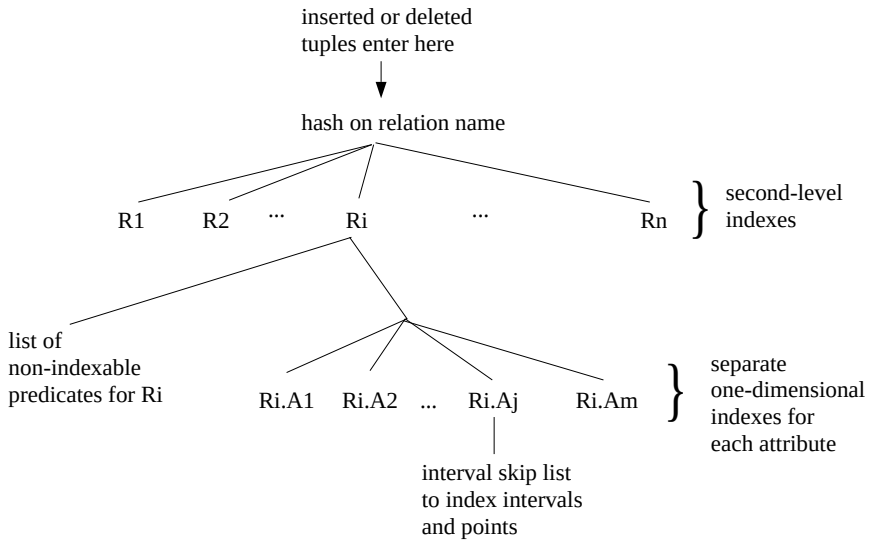


Figure 1: High-level diagram of predicate indexing scheme.

5 Interval Skip Lists

An important problem that arises in a number of computer applications, including active database rule systems, is the need to find all members of a set of intervals that overlap a particular point. Queries of this kind are called *stabbing queries* [27]. This section introduces the interval skip list (IS-list), which is designed to handle stabbing queries efficiently. The IS-list is an extension of the randomized list structure known as the *skip list* [26]. In what follows, alternative algorithms for solving stabbing queries are reviewed, and the IS-list algorithm and its performance analysis are given.

5.1 Review of Stabbing Query Solution Methods

Formally, the stabbing query problem is to find all intervals in the set $Q = \{i_1, i_2, ...i_n\}$ which overlap a query point X. Intervals are written using a comma-separated pair of values. Inclusive interval boundaries are indicated by square brackets, and non-inclusive boundaries by parentheses. Open intervals have one boundary at positive or negative infinity, and points have both boundaries equal. Examples of intervals are [17,19), [12,12], [-∞,22]. Several different approaches to solving the stabbing query problem have been developed. The simplest solution is to place all n intervals in Q in a list and traverse the list sequentially, checking each interval to see if it overlaps the query point. This algorithm has a search complexity of $O(n)$.

A more sophisticated approach is based on the *segment tree* [27]. To form a segment tree, the set of all endpoints of intervals in Q is formed, and an ordered complete binary tree is built that has the endpoints

as its leaves. To index an interval, the identifier of the interval is placed on the uppermost nodes in the tree such that all values in the subtrees rooted at those nodes lie completely within the interval. In this way, an interval of any length can be covered using $O(\log n)$ identifiers. Hence, the segment tree requires $O(n \log n)$ storage. In order to solve a stabbing query using a segment tree, the tree is traversed from the root to the location the query value X would occupy at the bottom of the tree. The interval identifiers on all nodes visited are returned as the answer to the query. A query takes $O(\log n)$ time. The segment tree works well in a static environment, but is not adequate when it is necessary to dynamically add and delete intervals, as it is in an active database predicate index.

Another data structure that can be used to process stabbing queries is the *interval tree* [8, 9]. Unfortunately, as with the segment tree, all the intervals must be known in advance to construct an interval tree.

R-trees can index intervals dynamically [15]. Subtrees of each R-tree index node contain only data that lies within a containing rectangle in the index node. Since rectangles in each node may overlap, on searching or updating the tree it may be necessary to examine more than one subtree of any node. An important part of the R-tree algorithm involves use of heuristics to decide how to partition the rectangles in a subtree to determine the best set of index rectangles for an index node. Due to its generality, and the indexing heuristics required, the R-tree is challenging to implement (the IS-list proposed in this paper is of relatively low complexity). A useful property of R-trees is that they require only $O(n)$ space. Their performance should be good for rectangles (or intervals in the 1-dimensional case) with low overlap, but when there is heavy overlap, search time can degenerate rapidly.

The *time index* [10] is designed to support efficient retrieval of versions of objects in a temporal database. Object versions are valid over time intervals. A time index can support stabbing queries (find all objects valid at a point in time) in $O(\log n)$ time given n unique time points in the index. Time indexes can also support finding all objects valid during non-empty interval [l,r] where l <r. Time indexes are an extension of B-trees. One difficulty with time indexes is that they can take $O(n^2)$ space in the worst case to index n time intervals, though a compression technique based on storing only incremental changes in leaf pages keeps the constant factor low. It is possible to achieve $O(\log n)$ search time for stabbing queries using $O(n \log n)$ space (the IS-lists presented here achieve this).

Another data structure which solves the stabbing query problem efficiently (among others), and does allow dynamic insertion and deletion of intervals is the *priority search tree* [23]. An advantage of the priority search tree is that it requires only $O(n)$ space to index n intervals. However, the priority search tree in its balanced form is very complex to implement [34]. In addition, for a priority search tree to handle a set

of intervals with non-unique lower bounds, a special transformation must be used to transform the set of intervals into one where the intervals have unique lower bounds. This transformation is not trivial, and it must be created for each different data type to be indexed.

The *interval binary search tree* (IBS-tree) can handle stabbing queries, and can be balanced more easily and is easier to implement than the priority search tree, although it requires $O(n \log n)$ storage [18, 19]. We conjecture that balanced IBS-trees require $O(\log n)$ time for searching and $O(\log^2 n)$ average time for insertion and deletion, though a definitive performance analysis has not been done. A data structure closely related to the IBS-tree called the *stabbing tree* has been developed to find the *stabbing number* for a point given a collection of intervals [14]. The stabbing number is the number of intervals that overlap a point. In contrast, the IBS-tree and the IS-list return a *stabbing set* containing all the intervals overlapping the query point, not just the number. The interval skip list is quite similar in principle to the IBS-tree, but it inherits the simplicity of skip lists, making it much easier to implement than the balanced IBS-tree.

One can envision a structure called an "interval B-tree," that could solve stabbing queries using a hierarchical mark-placement strategy similar to that used for IBS-trees and the IS-lists described here. To our knowledge, no description of such a structure has been published. We have not pursued development of such a structure because we felt that it would be more complex than IS-lists to implement and not perform better. An "interval B-tree" would be a close cousin of the time index [10].

5.2 IS-list Algorithms

The IS-list is formed by augmenting a skip list with additional information. IS-lists can accommodate points as well as open and closed intervals with inclusive and exclusive boundaries. The skip list data structure [26] is reviewed below and then extensions needed to index intervals are described.

5.2.1 Review of Skip Lists

Skip lists are a probabilistic alternative to balanced binary trees. They support search, insertion, deletion, and ordered scan operations. What is astonishing about skip lists is that they can do what balanced trees such as AVL trees [1] can do, but they are much simpler to implement [26]. Skip lists also have other advantages over balanced trees, including lower storage use, and smaller constant factor overhead in actual implementations.

The skip list is similar to a linked list, except that each node on the list can have one or more forward pointers instead of just one forward pointer. The number of forward pointers the node has is called the *level*

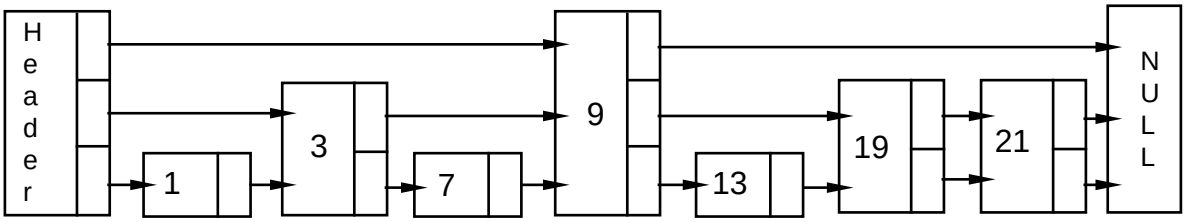


Figure 2: Example of a skip list.

of the node. When a new node is allocated during a list insertion, its level is chosen at random, independently of the levels of other nodes. The probability a new node has x levels is

$$P(x = k) = \begin{cases} 0 & \text{for } k < 1 \\ (1-p) \cdot p^{k-1} & \text{for } k \geq 1 \end{cases}$$

where p is a real number in the interval $(0, 1)$ that parameterizes the skip list. With $p = 1/2$, the distribution of node levels will allocate approximately $1/2$ the nodes with one forward pointer, $1/4$ with two forward pointers, $1/8$ with three forward pointers, and so on. In the algorithms given in this paper, the convention is that for a node with k levels, the bottom level is numbered 0 and the top level is numbered $k - 1$.

A skip list is normally organized with values in increasing order. A node's pointer at level l points to the next node with $l + 1$ or more forward pointers. An example of a skip list is shown in Figure 2. Searching in a skip list involves "stair-stepping" down from the beginning of the list to the location of the search key. The process of searching a skip list for a search key K begins at the list header at the level i equal to one less than the maximum level of a node in the list. Assume that the current node being visited is called y (y initially is the header). If the value of the key of the node pointed to by the level i pointer of y is $\geq K$, i is set to $i - 1$. Otherwise, y is set to be the node pointed to by the level i forward pointer of the current node. The search continues in this fashion until $i = 0$, at which point the node immediately after y either has a key equal to K, or else K is not present in the list and it would be located immediately after y.

Insertion and deletion in skip lists involves simply searching and splicing. The splicing operation is supported by maintaining an array of nodes whose forward pointers need to be adjusted. For a full description of the algorithms for maintaining skip lists and skip lists extended to support additional capabilities such as searching with fingers, efficient merging, finding the kth item in a list etc. the reader is referred to [26, 25].

The performance of skip lists is quite similar to that of balanced binary search trees. The expected value of times for searching, insertion and deletion in a skip list with n elements are all $O(\log n)$. The variance of

the search times is also quite low, making the probability that a search will take significantly longer than $\log n$ time vanishingly small. Comparing actual implementations of skip lists and AVL trees [1], skip lists perform as well as or better than highly-tuned non-recursive implementations of AVL trees, yet programmers tend to agree that skip lists are significantly easier to implement than AVL trees [26]. A discussion of extensions to skip lists to support interval indexing is given below.

5.2.2 Extending Skip Lists to Support Intervals

The basic idea behind the interval skip list is to build a skip list containing all the endpoints of a collection of intervals, and in addition, place markers on nodes and forward edges in the skip list to "cover" each interval. The placement of markers on edges and nodes in an interval skip list can be stated in terms of the following invariant:

Interval skip list marker invariant: Consider an interval $I = (A,B)$ to be indexed. End points A and B are already inserted into the list. Consider some forward edge in the skip list from a node with value X to another node with value Y. The interval represented by this edge is (X,Y). A marker containing the identifier of I will be placed on edge (X,Y) if and only if the following conditions hold:

1. **containment:** I contains the interval (X,Y).
2. **maximality:** There is no forward pointer in the list corresponding to an interval (X', Y') that lies within I and that contains (X,Y).

In addition, if a marker for I is placed on an edge, then the nodes that are the endpoints of that edge and have a value contained in I will also have a mark placed on them for I. Open intervals are represented as closed intervals with one inclusive boundary at infinity, e.g., $(7,\infty]$. Hence without loss of generality only closed intervals are considered. A diagram illustrating the application of the invariant is shown in Figure 3.

An example of a set of intervals and the IS-list for those intervals is shown in Figure 4. Searching an IS-list to find all intervals that overlap a search key can be done efficiently given a skip list with markers on it satisfying this invariant. The challenge in inserting and deleting intervals into an interval skip list is to perform the operations efficiently while maintaining the invariant. The remainder of this section describes the procedures for searching, insertion and deletion in IS-lists.

5.2.3 Searching

The procedure to search an IS-list L to find all intervals that overlap a search key K, and return those intervals in a set S, is to search along the same path that would be visited by the standard skip list search

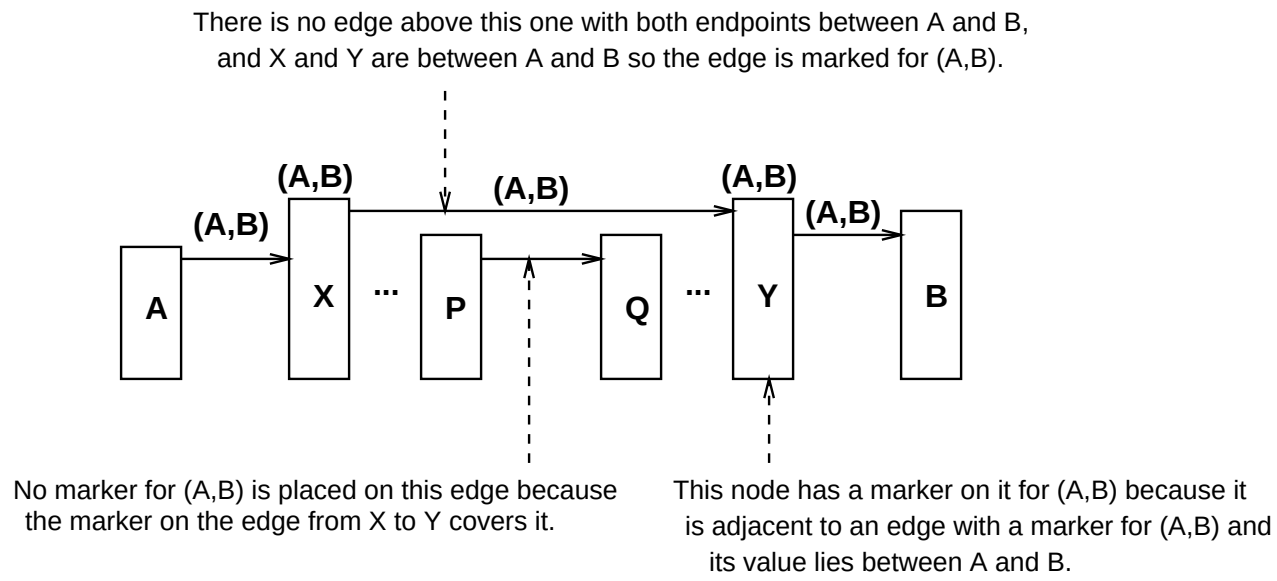


Figure 3: An example illustrating the interval skip list marker placement invariant.

procedure, and add markers to S as the search proceeds. Whenever the procedure drops from level i to $i-1$ during the search, it adds to S the markers on the forward pointer at level i of the current node. This is valid since markers on the forward pointer at level i must belong to an interval that contains K. At the final destination, if K is present in the list, the procedure adds the markers on node K to S. Otherwise, (when K is not present) it adds the markers on the lowest pointer of the current node to S. When the search terminates, exactly one marker for every interval that overlaps K will be in S. No duplicates will be found. Each node of level i in an interval skip list contains the following:

key: a key value,

forward: an array of forward pointers, indexed from 0 to $i-1$, as in a regular skip list,

markers: an array of sets of markers, indexed from 0 to $i-1$,

owners: a bag (multi-set) of identifiers of the intervals that have an endpoint equal to the key value of this node (one interval identifier can appear twice here if the interval is a point),

eqMarkers: a set of markers for intervals that have a marker on an edge that ends on this node, and which contain the key value of this node.

An outline of an implementation of this search algorithm is shown as the procedure findIntervals(K,L,S) below. In this algorithm and the other algorithms discussed later, the boolean operators **and** and **or** are conditional. In other words, in evaluation of p **and** q, if p is false, then q is never evaluated. Similarly, in evaluation of p **or** q, if p is true, then q is never evaluated. Comments are indicated by a '//' as in C++.

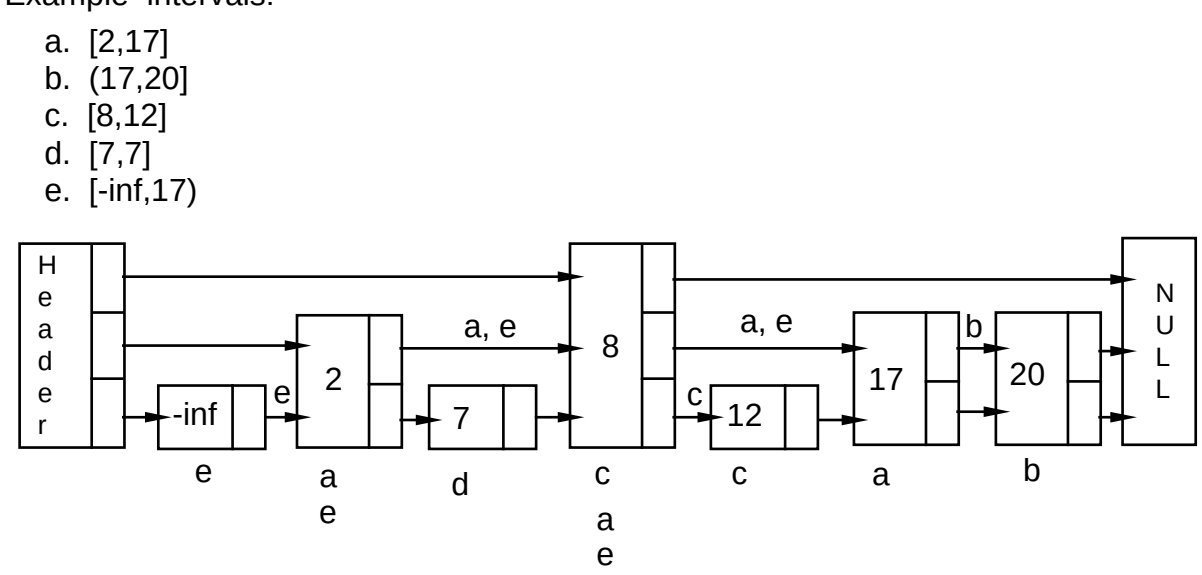


Figure 4: Example of an interval skip list for intervals shown.

```
procedure findIntervals(K,L,S)
    x := L.header; S := φ
    // Step down to bottom level.
    i := maxLevel
    while i ≥ 0 and (x is the header or x→key ≠ K) do
    begin
        // Search forward on current level as far as possible.
        while (x→forward[i] ≠ null and x→forward[i]→key < K) do
            x := x→forward[i]
        // Pick up interval markers on edge when dropping down a level,
        // unless you are at the node containing K already, in which case
        // you pick up the eqMarkers on that node just prior to exiting loop.
        if x is not the header and x→key ≠ K then
            S := S ∪ x→markers[i]
        else if x is not the header then
            S := S ∪ x→eqMarkers[i]
        i := i-1
    end
end findIntervals
```

In an actual implementation, the set S of matching intervals can be constructed by building a list of pointers to sets of markers that reside on

1. individual forward pointers and

2. perhaps the final node visited.

The union operations in findIntervals require simply appending a single value to the list representing S. This value is a pointer to a mark set being added to S. This operation requires only $O(1)$ time per level in the IS-list.[2] Duplicates do not have to be removed from S, because it is not possible to add a marker for the same interval to S more than once. This is true since descending past two edges with a marker for the same interval on them during a search would imply that the IS-list marker invariant was violated, which is a contradiction. The **if** statement in the search algorithm prevents adding any duplicate markers to S in the case that the search key value K is found in the list. Discussion now turns to the algorithm for inserting an interval into an IS-list.

5.2.4 Insertion

To insert an interval (A,B) into an IS-list, the first step is to insert A and B separately if they are not already in the list and adjust existing markers as necessary. The next step is to start at A, search for B, and place markers for (A,B) in a way that satisfies the marker invariant.

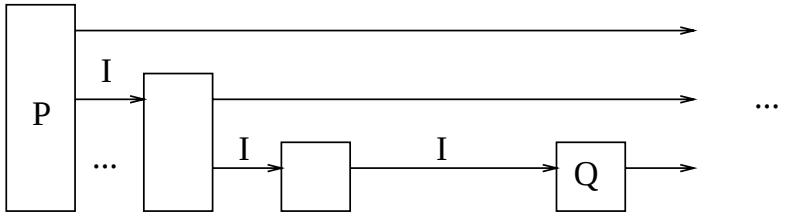
To place an interval end-point A into the list, the first step is to use the standard interval skip list insertion algorithm [26] to insert A. During this step, one must save a pointer to the new IS-list node containing A (call this N) and save the **updated** array which contains pointers to the nodes with pointers to N that had to be adjusted when A was inserted. The next step is to adjust the markers so that the IS-list marker invariant is maintained. An important observation is that markers can only stay at the same level or go up to a higher level after an insertion. They never move down. The procedure shown below adjusts markers to maintain the invariant after insertion of a node. It first places markers on the outgoing edges from N, raising them to higher levels as necessary. Then it raises markers on edges leading into N as necessary. In the procedure, the function level(x) returns the number of forward pointers of node x.

```
procedure adjustMarkersOnInsert(L,N,updated)
// Update the IS-list L to satisfy the marker invariant.

// Input: IS-list L, new node N, vector 'updated' of nodes with updated pointers.
// The value of updated[i] is a pointer to the node whose
// level i edge was changed to point to N.

// Phase 1: place markers on edges leading out of N as needed.
```

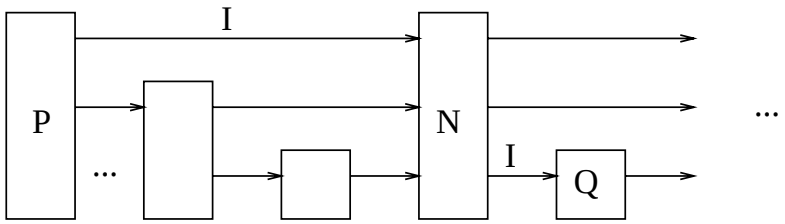
[2] If preferred, one may choose to implement the construction of S by traversing each set of markers added to S and adding the markers individually to a list of markers representing S. This method will add additional time $O(|S|)$ to the total search time. Since applications would have to traverse the list of markers returned anyway, constructing S this way would not affect the order of growth of the running time of the application, and it might be more convenient from a software engineering perspective.

```
// Starting at bottom level, place markers on outgoing level i edge of N.
// If a marker has to be promoted from level i to i+1 or higher, place
// it in the promoted set at each step.

promoted := φ // make set of promoted markers initially empty
newPromoted := φ // temporary set to hold newly promoted markers
removePromoted := φ // temporary set to hold markers to be removed from promoted

i := 0
while i ≤ level(N) - 2 and N→forward[i+1] ≠ null do
begin
    for m in updated[i]→markers[i] do begin
        if the interval of m contains (N→key,N→forward[i+1]→key)
        then // promote m
            remove m from the level i path from N→forward[i]
            to N→forward[i+1] and add m to newPromoted
        else place m on the level i edge out of N
    end

    for m in promoted do begin
        if the interval of m does not contain (N→key,N→forward[i+1]→key)
        then // m does not need to be promoted higher.
            // Place m on the level i edge out of N and remove m from promoted
            // using deferred update since promoted is being scanned.
            add m to N→markers[i]
            // mark N→forward[i] if needed
            if the interval of m contains N→forward[i]→key
                then add m to N→forward[i]→eqMarkers
            add m to removePromoted
        else // continue to promote m
            remove m from the level i path from N→forward[i] to N→forward[i+1]
    end

    remove all elements of removePromoted from promoted
    removePromoted := φ
    promoted := promoted ∪ newPromoted
    newPromoted := φ
    i := i+1
end

// Combine the promoted set and updated[i]→markers[i] and install them as
// the set of markers on the top edge out of N that is non-null.
N→markers[i] := promoted ∪ updated[i]→markers[i]
// Place promoted markers on N→forward[i] if needed.
for m in promoted do
    if the interval of m contains N→forward[i]→key
        then add m to N→forward[i]→eqMarkers

// Phase 2: adjust markers to the left of N as needed.

// Markers on edges leading into N may need to be promoted as
// high as the top edge coming into N, but never higher.
```

```
promoted := φ
newPromoted := φ
tempMarkList := φ
i := 0
while i ≤ level(N)-2 and updated[i+1] is not the header do begin
      copy update[i]→markers[i] into tempMarkList
      // scan tempMarkList instead of update[i]→markers[i] to avoid problem of
      // updating same set we're scanning
      for each mark m in tempMarkList do begin
            if m needs to be promoted (i.e. m's interval contains (updated[i+1]→key,N→ key))
            then begin
                  place m in newPromoted.
                  remove m from the path of level[i] edges between
                        updated[i+1] and N (it will be on all those edges
                        or else the invariant would have previously been violated).
            end
      tempMarkList := φ

      for each mark m in promoted do begin
            if m belongs at this level, (i.e. updated[i] is not the header, m's interval
                  covers (updated[i]→key,N→key),
                        updated[i+1] is not the header, and m's interval does not cover
                        (updated[i+1]→key,N→key))
            then begin // m doesn't need to be promoted higher.
                  add m to N→markers[i]
                  // Mark update[i] if needed.
                  if the interval of m contains update[i]→key
                        then add m to update[i]→eqMarkers
                  // Remove m from promoted (using deferred update).
                  add m to removePromoted
                  end
            else remove m from the level i path from updated[i+1] to N.
      end

      promoted := promoted − removePromoted
      removePromoted := φ
      promoted := promoted ∪ newPromoted
      newPromoted := φ
      i := i + 1
end

// Assertion: i=level(N)-1 or updated[i+1] is the header.
// Put all marks in the promoted set on level i edge coming into N.
updated[i]→markers[i] := updated[i]→markers[i] ∪ promoted
// Mark updated[i] as needed.
for each m in promoted do
      if the interval of m contains updated[i]→key
            then add m to updated[i]→eqMarkers

// Place markers on N for all intervals with markers that cross directly through N.
// (Since N is a new node, every marker coming into N must also leave N.)
for i:= 0 to level(N)-1 do
      N→eqMarkers := N→eqMarkers ∪ N→markers[i]
```



Markers placed for interval I=(P,Q) on sample IS-list.



Markers placed for I after insertion of node N and marker adjustment.

Figure 5: Example of node insertion and marker promotion.

end adjustMarkersOnInsert

An example of insertion of a node N and the corresponding promotion of markers in an example IS-list that would be accomplished by adjustMarkersOnInsert is shown in Figure 5. A key feature of this procedure is that the time taken to examine a marker that is not promoted is $O(1)$. This fact is important for the overall performance of the insertion operation, as will be discussed in section 5.3.

Placing markers to cover the inserted interval (A,B) is accomplished by following forward pointers from A to B along the path defined by the IS-list marker invariant. In general this will involve stepping up several levels in the list from A and then stepping back down to B. An example of the general case is shown in Figure 6. There are also special cases in which it is only necessary to step down or up or proceed on the same level from A to B. Let I be an interval with lower and upper endpoints A and B respectively. The procedure to place markers for I on an interval skip list L that already contains endpoints A and B is shown below.

procedure placeMarkers(L,A,B,I)
begin
    // A and B are the IS-list nodes containing the left and right endpoints of I, respectively.
    // First mark non-descending path.
    x := A
    if I contains x→key then add I to x→eqMarkers

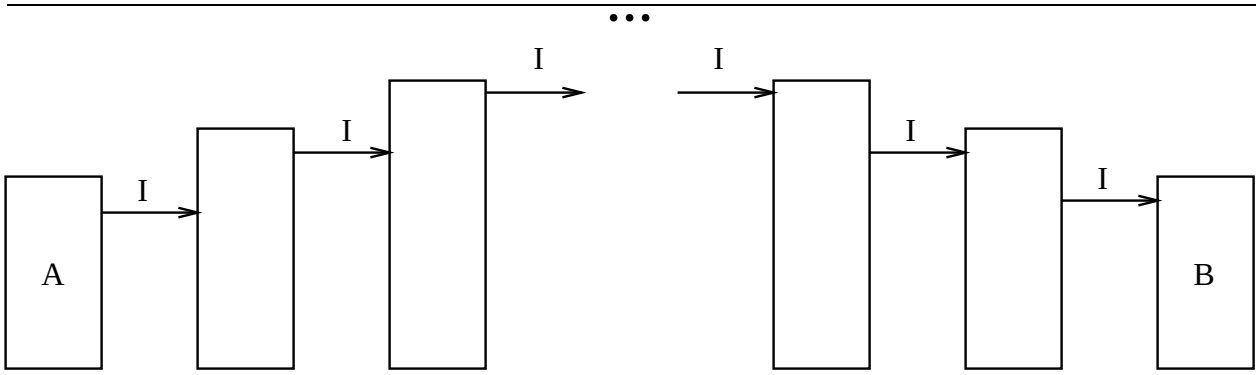


Figure 6: Placement of markers in IS-list to cover interval (A,B).

```
        i := 0    // start at level 0 and go up
        while x→forward[i] ≠ null and I contains (x→key,x→forward[i]→key) do
        begin
              // find level to put mark on
              while(i ≠ level(x)-1 and x→forward[i+1] ≠ null and
                    I contains (x→key,x→forward[i+1]→key) do
                          i := i + 1
              // Mark current level i edge since it is the highest edge out of x that contains I,
              // except in the case where the current level i edge is null, in which case it should
              // never be marked.
              if x→forward[i] ≠ null then begin
                    add I to x→markers[i]
                    x := x→forward[i]
                    // Add I to eqMarkers set on node unless currently
                    // at right endpoint of I and I doesn't contain right endpoint.
                    if I contains x→key then add I to x→eqMarkers
              end
        end

        // mark non-ascending path
        while x→key ≠ B →key do begin
              // find level to put mark on
              while(i ≠ 0 and (x→forward[i]=null or I does not contain (x→key,x→forward[i]→key)) do
                    i := i - 1
              // At this point, we can assert that i=0 or x→forward[i]≠null and
              // I contains (x→key,x→forward[i]→key).
              // In addition, x is between A and B so i=0 implies I contains
              // (x→key,x→forward[i]→key). Hence, level i edge out of x must be marked.
              add I to x→markers[i]
              x := x→forward[i]
              if I contains x→key then add I to x→eqMarkers
        end
  end placeMarkers
```

The procedure for deleting an interval from an IS-list discussed below is analogous to the insertion procedure.

5.2.5 Deletion

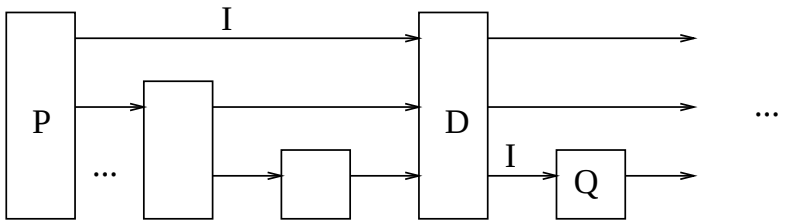
To delete an interval (A,B) the first step is to remove its markers. This is done by searching for the node containing A, and then scanning forward in the list for B, following a staircase pattern, which in general will contain an ascending path followed by a descending path. The approach used is very similar to that of the placeMarkers procedure for placing markers for a new interval, so a detailed algorithm for removing markers for (A,B) is not shown.

The next step in deletion of (A,B) is to remove the IS-list nodes containing the endpoints A and B, and to adjust any markers affected so that the IS-list marker invariant is still satisfied. Affected markers will always either stay at the same level or move down. They will never move up. (If deletion of a node would make them go up, they would have already been placed higher, contradicting the IS-list marker invariant.) This forms the basis for an incremental algorithm for adjusting markers after deletion of a node that is similar to the one used after insertion of a node. The algorithm, which will be called adjustMarkersOnDelete, is implemented by the procedure below. The parameters to the procedure are the IS-list L, the node to be deleted D, and a vector **updated** that contains pointers to the nodes with pointers into D that must be updated after the deletion. The **updated** vector can be constructed during a standard IS-list search for D.

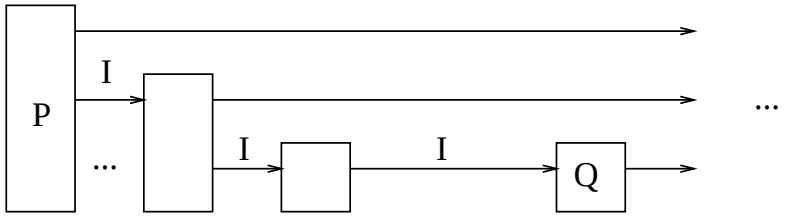
```
procedure adjustMarkersOnDelete(L,D,updated)
demoted := φ
newDemoted := φ

// Phase 1: lower markers on edges to the left of D as needed.
for i := level(D)-1 down to 0 do begin
    // Find marks on edge into D at level i to be demoted.
    for m in updated[i]→markers[i] do
        if D→forward[i] = null or the interval of m contains (updated[i]→key,D→forward[i]→key)
            then insert m into newDemoted

    // Remove newly demoted marks from edge.
    updated[i]→markers[i] := updated[i]→markers[i] − newDemoted
    // Note: updated[i]→eqMarkers is left unchanged because any
    // markers there before demotion must be there afterwards.

    // Place previously demoted marks on this level as needed.
    for m in demoted do begin
        // Place mark on level i from updated[i+1] to updated[i], not including
        // updated[i+1] itself, since it already has a mark if it needs one.
        y := updated[i+1]
        while y ≠ null and y ≠ updated[i] do begin
            if y ≠ updated[i+1] and the interval of m contains y→key
                then add m to y→eqMarkers
            add m to y→markers[i]
```

```
                y := y→forward[i]
            end
            if y ≠ null and y ≠ updated[i+1] and the interval of m contains y→key
                then add m to y→eqMarkers

            // If this is the lowest level m needs to be placed on, then place m on the level i
            // edge out of updated[i] and remove m from the demoted set.
            if D→forward[i] ≠ null and the interval of m contains (updated[i]→key,D→forward[i]→key)
            then begin
                add m to updated[i]→markers[i]
                add m to tempRemoved
            end
        end
        demoted := demoted − tempRemoved
        tempRemoved := φ
        demoted := demoted ∪ newDemoted
        newDemoted := φ
    end

    // Phase 2: lower markers on edges to the right of D as needed.
    demoted := φ
    // newDemoted is already empty
    for i := level(D)-1 down to 0 do begin
        for each marker m in D→markers[i] do
            if D→forward[i] ≠ null and (updated[i] is the header or
                the interval of m does not contain (updated[i]→key,D→forward[i]→key))
                    then add m to newDemoted.

        for each marker m in demoted do
        begin
            // Place mark on level i from D→forward[i] to D→forward[i+1]. Don't
            // place a mark directly on D→forward[i+1] since it is already marked.
            y := D→forward[i]
            while y ≠ D→forward[i+1] do begin
                add m to y→eqMarkers
                add m to y→markers[i]
                y := y→forward[i]
            end
            if D→forward[i] ≠ null and update[i] is not the header
                and the interval of m contains (update[i]→key,D→forward[i]→key)
                    then add m to tempRemoved
        end

        demoted := demoted − tempRemoved
        demoted := demoted ∪ newDemoted
        newDemoted := φ
    end
    end adjustMarkersOnDelete
```

An example showing the demotion of markers for an interval $I = (P, Q)$ that would be done for one possible IS-list after deletion of a node D appears in Figure 7. The incremental marker adjustment algorithms discussed above are important to the overall performance of insertion and deletion operations,



Markers for I=(P,Q) before deletion of node D.



Markers for I=(P,Q) after marker demotion & removal of D.

Figure 7: Example of node deletion and marker demotion.

which is analyzed in the next section. To facilitate the performance analysis, it is assumed there that mark sets are maintained using an $O(\log n)$ search structure (such as a skip list or AVL tree).

5.3 Performance Analysis

The expected time to search an IS-list to find all intervals that overlap a key K is $O(\log n)$, since $O(1)$ time is spent per level and there are $O(\log n)$ levels in the list. The cost of insertion and deletion are determined below.

Components of the insertion cost include the time required to

- insert the left and right endpoints of the interval,
- adjust markers for intervals already in the IS-list, and
- place markers for the new interval.

At the end of the operation the IS-list marker invariant must again hold. Inserting the left and right endpoints into the skip list requires $O(\log n)$ time. It is assumed that the interval markers on the edges are stored in a data structure that allows $O(\log n)$ inserts and deletes. This assumption implies that the time required to place markers for the new interval is $O(\log^2 n)$, because there are $O(\log n)$ levels and markers are placed on $O(1)$ edges of a level, at a cost of $O(\log n)$ per edge. The remaining cost that must be calculated is that of

promoting markers due to the insertion of the endpoints of the new interval. It is said that a node *disturbs* an interval if the node cuts an edge that contains a marker for the the interval. For convenience, in the analysis it is assumed that the forward pointer levels are numbered $1...k$ for a level k node, rather than $0...k-1$ as was used in the description of the algorithms. To facilitate the analysis, the following are defined:

$P(i)$: probability that the inserted node has i levels,

$D(i,o)$: set of intervals disturbed (with markers on levels 1 through i) by operation o which inserts an i level endpoint,

$R(i,o,s)$: cost to promote the markers for interval s when operation o inserts a level i endpoint,

$A(i,o)$: cost to adjust the markers for operation o which inserts a level i endpoint.

In addition, using $E[v]$ to indicate the expected value of v, we define $D(i) = \mathrm{E}[|D(i,o)|]$, $A(i) = \mathrm{E}[A(i,o)]$, and $A = \mathrm{E}[A(i)]$.

Theorem 1 *If $D(i) = O(p^{-i})$, then the expected time required to adjust the existing markers in an IS-list when an endpoint is inserted is $O(\log^2(n))$.*

Proof: The value that needs to be calculated is A. If the expected adjustment cost is known for an operation that inserts a level i endpoint, $A(i)$, then A can be calculated by taking expectations. If the underlying skip list is parameterized by p, then the probability that a node has i levels is:

$$P(i) = (1-p)p^{i-1} \tag{1}$$

Therefore,

$$A = \sum_{i=1}^{\infty} (1-p)p^{i-1}A(i) \tag{2}$$

In order to find $A(i)$, the first step is to examine the cost to adjust the markers for the intervals that operation o disturbs when it inserts a level i endpoint. Operation o disturbs intervals in $D(i,o)$, and each of these intervals requires $R(i,o,s)$ time to adjust its markers. Therefore

$$A(i,o) = \sum_{s \in D(i,o)} R(i,o,s) \tag{3}$$

We consider the algorithm for updating the markers of the disturbed intervals. If no marker for that interval is promoted to a higher level, then processing that interval requires $O(1)$ time. If a marker for the

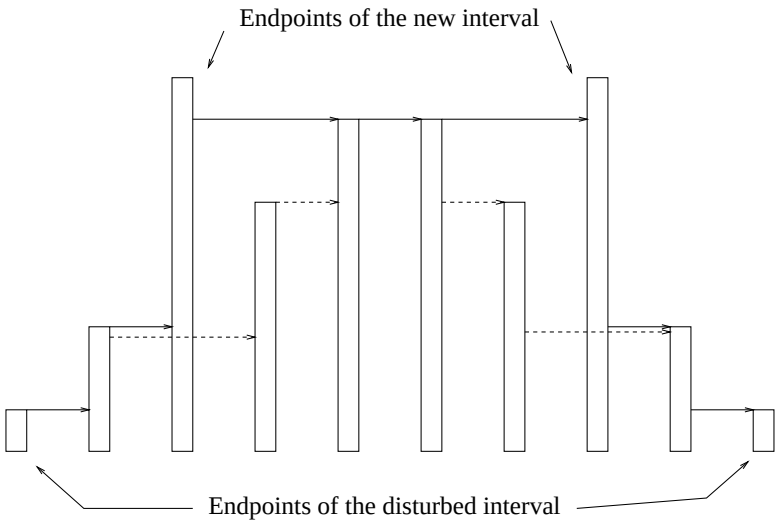


Figure 8: The dashed lines represent edges from which markers for the old interval are removed when the new interval is inserted.

interval is promoted from level j to level k, then all markers for that interval between the new node and the previous (next) k-level node must be deleted, (see Figure 8). There are $O(1)$ consecutive s-level nodes between $(s+1)$-level nodes, so promoting a marker by l levels requires that $O(l)$ markers be deleted, which requires $O(l \log n)$ time.

We consider an interval that is disturbed when the new endpoint is inserted. The number of nodes that the level j markers for the interval cover is $O((1/p)^j)$. If an i-level node is inserted and disturbs the interval, then the marker for the interval is on a level $i-l$ edge with probability $O(p^l)$. We assume that if a marker is promoted when an i-level node is inserted, it is promoted to level i. Therefore, a marker for a disturbed interval is promoted an expected $\sum_{l=1}^{i-1} l \cdot O(p^l) = O(1)$ levels.

The expected amount of work to adjust markers for a disturbed interval when an i-level endpoint is inserted is therefore

$$\begin{aligned} E[R(i,o,s)] = &\quad O(1) + O(1) \cdot O(\log n) \\ = &\quad O(\log n) \end{aligned} \tag{4}$$

Therefore,

$$\begin{aligned} A(i) = &\quad E[A(i,o)] \\ = &\quad E[\textstyle\sum_{s \in D(i,o)} O(\log(n))] \end{aligned}$$

$$
\begin{aligned}
&= && O(\log(n) \cdot E[|D(i,o)|]) \\
&= && O(\log(n) \cdot D(i)) \\
&= && O(min(\log(n)p^{-i}, \log(n) \cdot n)) \qquad (5)
\end{aligned}
$$

In the last step, we use the assumption that $D(i) = O(p^{-i})$, and the fact that no more than n intervals can be disturbed. Putting equation (5) into equation (2) yields

$$
\begin{aligned}
A = & && \textstyle\sum_{i=1}^{\infty} P(i)A(i) \\
&= && \textstyle\sum_{i=1}^{\infty} p^i \cdot min(\log(n)p^{-i}, \log(n) \cdot n) \\
&= && \textstyle O(\sum_{i=1}^{\lceil \log n \rceil} \log(n) + \log(n) \sum_{i=\lceil \log n \rceil + 1}^{\infty} p^i \cdot n) \\
&= && \textstyle O(\log^2 n + \log(n) \sum_{i=\lceil \log n \rceil + 1}^{\infty} p^i p^{-\log n})
\end{aligned}
$$

To finish, the substitution $r = i - \lceil \log n \rceil$ is made,

$$
\begin{aligned}
A = & && \textstyle O(\log^2 n + c \log(n) \sum_{r=1}^{\infty} p^r) \\
&= && O(\log^2 n) \qquad (6)
\end{aligned}
$$

The last step follows because $p < 1$, so the sum is $O(1)$. •

Corollary 1 *If $D(i) = O(p^{-i})$, then the expected time to perform an insert is $O(\log^2 n)$.*

Proof: The time to to insert both endpoints is $O(\log^2 n)$. The remaining work is to add the markers for the new interval, which is $O(\log^2 n)$. •

Corollary 2 *If $D(i) = O(p^{-i})$, then the expected time to perform a delete is $O(\log^2 n)$.*

Proof: the analysis is the same as for the insert operation. •

This analysis of the IS-list assumes that an operation that inserts a level i node disturbs $O(p^{-i})$ intervals. We feel that most interesting distributions that we encounter will satisfy this assumption.

We consider the following arguments: An interval in the IS-list follows a staircase pattern, and so places at most $O(1)$ markers on every level. Therefore, there are at most $O(n)$ markers placed on every level. The probability that a node has i or more pointers is $\sum_{j \geq i} P(j) = \sum_{j \geq i} (1-p)p^{j-1} = p^{i-1}$, so the expected number of forward level i edges is np^{i-1}. If every level i forward edge is equally likely to be cut when a node with at least i levels is inserted, then $D(i) = O(p^{-i})$.

The assumption that the expected number of level i forward edges is np^{i-1} is safe, because the skip list algorithm explicitly randomizes the node levels. The assumption that we are making about the underlying distribution is that every level j edge is equally likely to span the next insertion of a level i node, $j \leq i$. We next show that a large class of distributions are actually biased towards choosing edges with few markers on them.

Theorem 2 *If the endpoints of the distribution are chosen independently and identically distributed (iid) from a continuous distribution, then* $D(i) = O(p^{-i})$.

Proof: We consider, without loss of generality, that the endpoints are chosen iid from the uniform random distribution on $[0, 1]$ (other continuous distributions can be mapped to a uniform $[0, 1]$ distribution). Let us count $M(w)$, the expected number of markers placed on a level i edge of length w (the distance between the endpoints is w). We will call this edge e_w, and the level $i + 1$ edge that covers e_w will be called e_s. We will say that e_s is of length s. A marker for the interval (a, b) is placed on e_w if and only if the endpoints of e_w are contained within (a, b), but the endpoints of e_s are not.

Let us first determine the probability that a marker would be placed on e_w if e_s does not exist. We consider a randomly chosen interval, (a, b), and the probability, M_w, that its marker is placed on e_w. The endpoints of the interval are uniformly randomly chosen, so that the joint distribution of (a, b) has the distribution of a two element order statistic. The theory of order statistics [11] tells us that the density of the joint distribution $g(a, b)$ is a constant 2 in the region $b \in [0, 1]$, $a \in [0, b]$. Let us define w_1 and w_2 to be the lower and higher endpoints of e_w. We can also show that the density function for the w_1, $h(w_1)$ is a constant $1/(1 - w)$ in the region $[0, 1 - w]$. Let $f(a, b, w_1)$ be the joint density of a, b, and w_1. Since w_1 is chosen independently of (a, b), $f(a, b, w_1) = g(a, b)h(w_1)$. A marker for (a, b) is placed on e_w iff. $a \leq w_1$ and $b \geq w_2 = w_1 + w$, so that:

$$\begin{aligned} M_w &= \int_{w_1=0}^{1-w} \int_{a=0}^{w_1} \int_{b=w_1+w}^{1} f(w_1, a, b)\, db\, da\, dw_1 \\ &= \int_{w_1=0}^{1-w} \int_{a=0}^{w_1} \int_{b=w_1+w}^{1} 2/(1-w)\, db\, da\, dw_1 \\ &= (1-w)^2/3 \end{aligned} \tag{7}$$

We see that longer edges (in terms of the difference in the endpoint keys) are less likely be spanned by a random interval, and so are less likely to carry a marker for that interval (see Figure 9). Similarly, the number of intervals that cover e_s and so are not placed on e_w is $(1-s)^2/3$. Let $M_{w,s}$ be the probability that

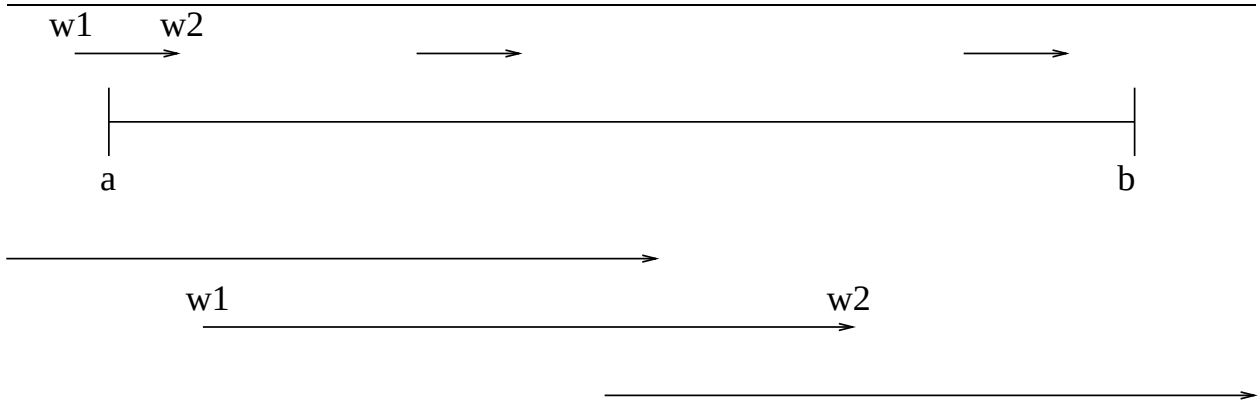


Figure 9: A long edge is less likely to fit in an interval (a, b) than is a short edge.

a marker is placed on an edge of length w whose parent edge is of length s. Then:

$$M_{w,s} = [(1-w)^2 - (1-s)^2]/3 \tag{8}$$

The lengths of e_w and e_s are not independent. However, since endpoints and node levels are chosen iid, the length of e_s is w plus an increment, a_i, that depends only on the level, i. So, the expected number of markers placed on e_w is n times the probability that an interval's marker is placed on the edge:

$$M(w) = nM_{w,w+a_i} \tag{9}$$

$$= n[(1-w)^2 - (1-w-a_i)^2]/3 \tag{10}$$

$$= na_i(2-2w-a_i)/3 \tag{11}$$

When a new endpoint is chosen, the probability that it is covered by the level i edge e is proportional to the length of e. But the expected number of markers on edge e decreases with the length of e. Therefore, when an i-level endpoint is inserted, the level i edge that spans the endpoint is likely to have fewer than average markers on it, so that $D(i) = O(p^{-i})$. •

This leads to

Theorem 3 *If the endpoints of the intervals are chosen iid from a continuous distribution, then the time required to perform an insert or a delete is* $O(\log^2 n)$.

In summary, the IS-list allows stabbing queries to be done in $O(\log n)$ time, and updates in $O(\log^2 n)$ time, while using $O(n \log n)$ storage.

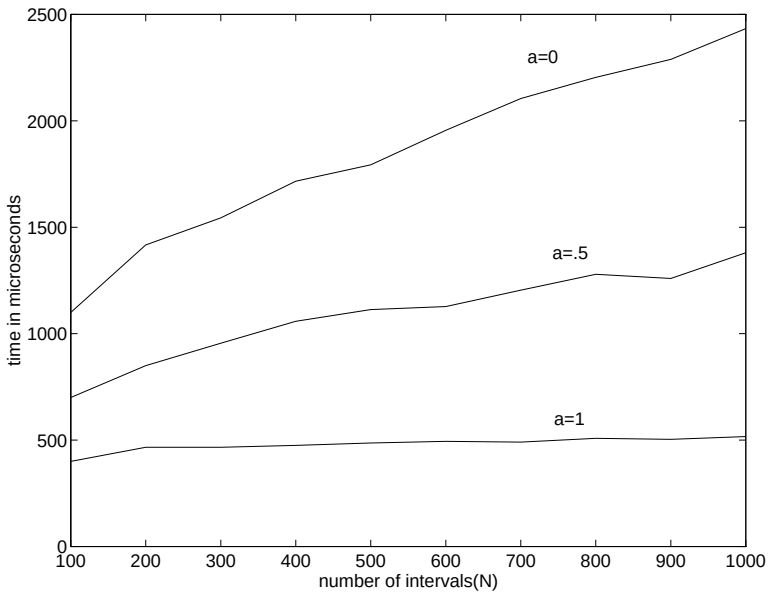


Figure 10: Average time of interval skip list insertion vs. number of intervals.

6 Performance Measurements

Some measurements of IS-list performance are given here. All measurements were done on a Sun SPARC-station ELC running at approximately 22 MIPS. IS-lists were implemented in C++. In the actual implementation, for which performance measurements are discussed in section 6, a singly linked list is used to represent mark sets. Adding a set of markers S1 to a set S2 is accomplished in the implementation by simply inserting each member of S1 at the head of S2. Using lists to represent mark sets yielded good results, so for simplicity using lists is recommended instead of an $O(\log n)$ search structure.

In the program runs conducted to take these measurements, intervals have integer endpoints. The left endpoint of each interval is drawn from a uniform random distribution between 1 and 10,000. Those intervals with non-zero width have their width drawn from a uniform random distribution between 1 and 1000. The parameter a is the fraction of intervals that have width zero. Hence, $1 - a$ is the fraction of intervals that are true intervals, not points.

Average interval insertion time for IS-list sizes up to 1000 intervals are shown in Figure 10 for a=0, .5, and 1. Average insertion time increases as a decreases since additional overhead is required to place and adjust markers. The curve for $a = 0$ contains only intervals, the curve for $a = .5$ has a 50/50 mix of intervals and points, and the curve for $a = 1$ contains only points.

Average IS-list search times for a=0, .5, and 1 are shown in Figure 11. To generate Figure 11, IS-lists

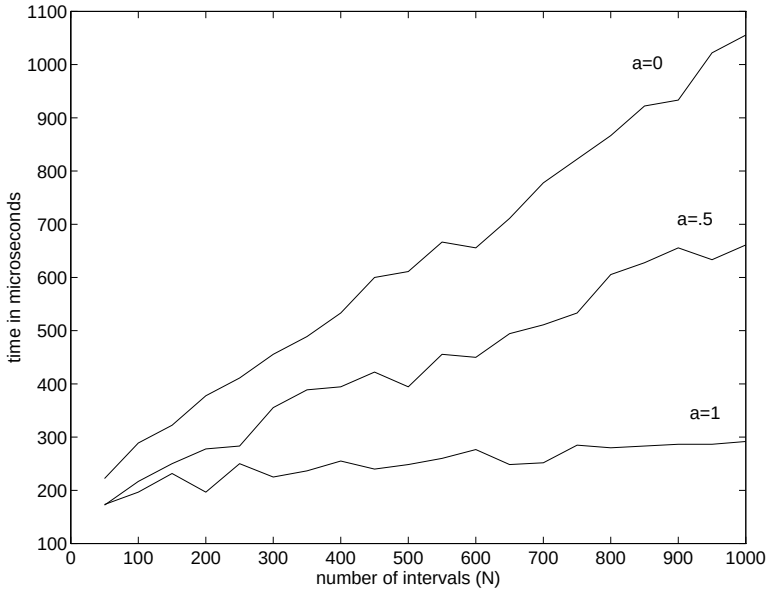


Figure 11: Average IS-list search time for a =0, .5 and 1.

of a fixed length were generated and then searched repeatedly using a randomly generated integer value between 1 and 10,000 as a search key. A new key was generated for each search. When $a = 1$, the intervals all have zero width (they are points) so usually no intervals will match during a search, and normally at most one will match. Thus, the curve for $a = 1$ has a logarithmic shape. When $a = .5$, half the intervals are points and half have non-zero width uniformly distributed in the range 1 to 1000. Hence, on average a fraction of roughly (.5*1000)/10,000, or five percent, of the intervals with non-zero width match each search. The same is true when $a = 0$ and all intervals have non-zero width. The search time shown includes the cost to insert the matching intervals into a list, i.e., $O(1)$ time per matching interval is incurred during each search. This is why the curves for $a = .5$ and $a = 0$ appear to have a linear as well as logarithmic component. As previously noted, it would be possible to have the search procedure build a "list of lists" of markers during the search instead of a sequential list, avoiding the $O(1)$ cost per match to insert matching markers into the answer list. However, $O(1)$ time per match would still be required eventually since the application using the IS-list would have to examine each matching interval. The fact that the curves in Figure 11 are not strictly increasing, but rather just follow an increasing trend, is due to the random nature of the IS-list. For example, the particular IS-list with 500 elements used to generate the curve for a=.5 in the figure might by chance have had a slightly better structure than the one with 450 elements.

A comparison of the cost of IS-list search vs. sequential search for a=.5 is given in Figure 12. The curves cross when the number of intervals N=6. This indicates that the IS-list has low overhead, and thus

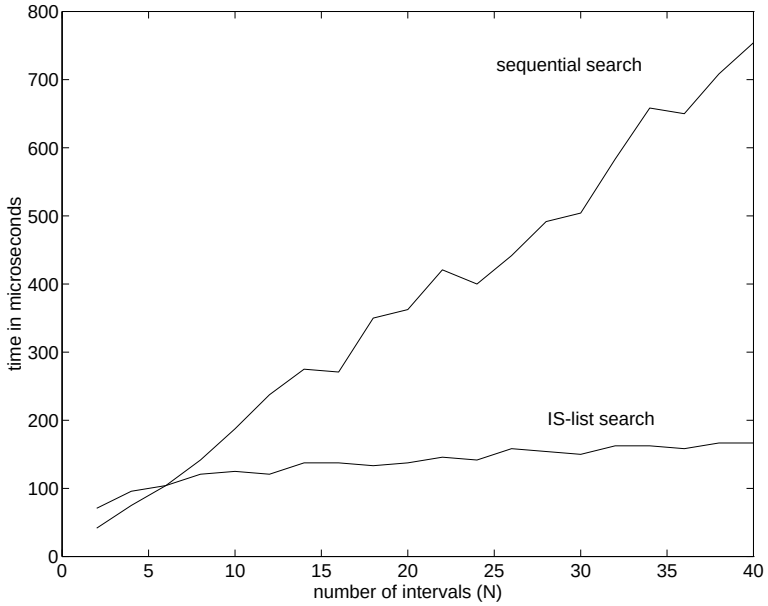


Figure 12: Average time of interval skip list search vs. sequential list search.

using an IS-list in a selection predicate index is worthwhile even for relatively small numbers of predicates per attribute.

One might ask how the IS-list insertion, deletion and search times compare with those for regular skip lists, and other ordered search structures like binary search trees and in-memory B-trees. IS-lists can do more than these structures, but a comparison is worthwhile to see what kind of overhead IS-lists introduce. Figure 10 sheds some light on this comparison. There, the curve for a=1 shows the average insert time for intervals of length 0, which are points. Hence, the order of execution of the IS-list insertion operation observed for a=1 is $O(\log n)$, the same for insertion into a standard skip list. The IS-list C++ code measured is implemented in a highly generic style, where the key values are members of a "Value" class, which has subclasses for integer, floating point, and string values. The same code will work for sets of intervals with endpoints of any type that is a subtype of "Value" as long as all the inserted endpoints have types that can be compared. This implementation makes the code general-purpose, but involves significant constant-factor overhead. When used for points, our IS-list code also has constant-factor overhead for examining mark sets on edges, which will always be empty. A highly tuned non-generic C implementation of skip lists tailored to integer keys can insert an integer in a list of 1000 elements in an average time of around 20 microseconds on a SPARCstation ELC. This is about a constant factor of 25 times faster than our IS-list code running to create the a=1 curve in Figure 10. Comparable results can be observed for deletion and search. One would expect that highly tuned implementations of other ordered search structures such as binary-search trees or

B-trees would have search, insert, and delete times close to, but slightly slower than those for the C skip list implementation measured.

When the IS-list is integrated into the overall predicate indexing scheme shown in Figure 1, predicate matching performance will depend on several factors, including:

- the fraction of predicates that are non-indexable,
- the number of attributes per relation,
- the fraction of attributes that have one or more predicate clauses,
- the number of indexable predicate clauses per attribute.

However, the time required to find matching predicates can be estimated using the following assumptions:

- hash search cost = .1 msec,
- fraction of predicates that are indexable = 90%,
- cost to test a predicate against a point in sequential search = .02 msec,
- average number of attributes per relation = 15,
- fraction of attributes per relation with 1 or more predicate clauses = 1/3,
- number of predicates per relation (N) = 200 (assuming that there are 200/5 = 40 predicates per attribute, the search cost in an IS-list for one attribute is approximately .17 msec),
- cost to test an entire predicate against a tuple when a partial match is found = .05 msec,
- number of clauses per predicate = 2,
- average selectivity of each predicate clause = .1.

The CPU usage times for operations shown above are reasonably close to the actual times for a Sun SPARCstation ELC. In this scenario, the cost to search to find the partially matching predicates is the following:

cost = hash cost
+ number of attributes searched ·
IS-list search cost
+ non-indexable predicate test cost

This yields the following numeric expression for cost:

$$\text{cost} = .1 + 15\tfrac{1}{3} \cdot .17 + (1 - .9) \cdot .02 \cdot 200$$
$$= .1 + 5 \cdot .17 + .4 = 1.35 \text{ msec}$$

Since there are 200 predicates per relation, and the selectivity of the predicate clauses is .1, that means that $.1 \cdot 200 = 20$ predicates must be tested after the initial search. The time to test these is $.05 \cdot 20 = 1$ msec. Thus, the total time for predicate testing is $1.35 + 1 = 2.35$ msec. This is a fairly realistic number for the cost of finding all predicates that match a tuple using the algorithm presented in this paper with a moderate to large number of rules on a machine the speed of a SPARCstation ELC. Measurements of the actual Ariel IBS-tree-based selection predicate index [5] show results quite close to those estimated here for a IS-list-based selection predicate index. Given that the 2.35 msec figure is a per-tuple CPU cost, the time is significant, but not prohibitive. Of course, this is a CPU-only cost, and any increase in CPU speed will cause selection predicate testing time to scale down accordingly.

7 Summary and Conclusion

In this paper we have introduced a discrimination network structure for finding all members of a set of selection predicates that match a database object. It was argued that the structure will be small enough to fit in main memory for three reasons. First, rules are a form of database schema, not data, and the size of the schema is normally relatively small. Second, the largest rule-based expert systems built contain on the order of 10,000 rules, which is small enough so the description and index structures for all rules will fit in main memory. One would expect that the number of rules in a large database rules system application would be of comparable size. Third, most applications that appear to require a very large number of rules can be redesigned to use a small number of rules plus additional tuples or fields in the database. Since the number of rules is expected to be small enough to fit in memory, a main-memory data structure was designed to take advantage of this.

The key component of the algorithm proposed is the interval skip list, an extended skip list for indexing both interval and point data. The IS-list is an efficient and relatively simple *dynamic* data structure for indexing intervals to handle stabbing queries efficiently. Unlike the commonly used segment tree, it can process insertions and deletions efficiently on-line, requiring $O(\log^2 n)$ time for each operation. We have implemented IS-lists in less than 1000 lines of C++ code, which is about one-third the amount of C++ code required in our implementation of interval binary search trees [5, 18]. Though implementation of IS-lists is not trivial, no other known interval index that is based on a self-balancing data structure and supports

both stabbing queries and dynamic updates can match the simplicity of implementation of the IS-list. This simplicity is in large part inherited from the skip list used as the basis for the IS-list. The main drawback of IS-lists is their potentially large storage utilization of $O(n \log n)$. If interval overlap is very low, less space is required. In fact, if intervals do not overlap, only $O(n)$ storage is needed. Thus, the IS-list may have advantages for applications that have intervals with limited overlap. The IS-list's storage cost may be well worth paying for applications that require a simple, efficient interval index that can be updated dynamically. Selection predicate indexing in active databases is such an application.

Though the IS-list was developed to solve the predicate indexing problem in active databases, it may be useful in predicate indexes for production rule systems for AI programming, such as implementations of OPS5. IS-lists could also be useful in VLSI CAD tools, geographic information systems, and other applications that deal with geometric data. The IS-list is useful anywhere an index for intervals is required which must be dynamically updatable.

IS-lists offer a unique blend of *simplicity* and *performance*. Another dynamic data structure, the IBS-tree, has been developed that can match the performance of IS-lists (at least empirically) [19, 18]. But a definitive analytical performance analysis of IBS-tree has not been done. More importantly, IBS-trees require *three times as many lines of code* to implement as IS-lists. We conjecture that any interval index based on a balanced-tree scheme would require substantially more code than IS-lists. Though the asymptotic performance of IS-lists is not always better than that of other known equivalent schemes for interval indexing, IS-lists perform at least as well, and the combination of simplicity, performance, and analytical rigor of IS-lists is unmatched.

Although the intent of this paper was not to investigate parallelism, the algorithm proposed can be made to run significantly faster on a course-grain parallel machine such as a shared-memory multi-processor. Data-level parallelism can be achieved by searching for predicates matching multiple tuples in parallel. Parallelism for matching a single tuple can be achieved by searching the second-level index on each attribute of the tuple simultaneously, devoting a processor per attribute. This could improve the performance of the algorithm by a factor nearly equal to the number of attributes searched in parallel (the initial hash cost is a per-tuple cost, and does not scale). In addition, when brute force search is required, as in the case of non-indexable predicates and when doing the final predicate test, the set of predicates to be checked can be divided evenly among the available processors.

In summary, the selection predicate indexing strategy presented here can improve the performance of active database systems. The data structures and algorithms presented, while not trivial, are simple enough to be feasible to implement in an active database system. The algorithm presented in this paper, using the

IBS-tree instead of the IS-list, has been implemented in the Ariel active DBMS [19, 16]. This implementation was done before the IS-list was discovered. If we were to implement the selection predicate index again, we would use the IS-list rather than the IBS-tree because the IS-list is simpler.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1262, 1962.

[2] Virginia E. Barker and Dennis E. O'Connor. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, 32(3):298–318, March 1989.

[3] David A. Brant and Daniel P. Miranker. Index support for rule activation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 42–48, May 1993.

[4] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison Wesley, 1985.

[5] Moez Chaabouni. A top-level discrimination network for database rule systems. Master's thesis, Dept. of Computer Science and Eng., Wright State Univ., December 1990.

[6] S. Chakravarthy et al. HiPAC: A research project in active, time-constrained database management, Final Technical Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[7] Lois M. L. Delcambre and James N. Etheredge. The relational production language: A production language for relational databases. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 153–162, April 1988.

[8] H. Edelsbrunner. A new approach to rectangle intersections: Part I. *International Journal of Computer Mathematics*, 13(3-4):209–219, 1983.

[9] H. Edelsbrunner. A new approach to rectangle intersections: Part II. *International Journal of Computer Mathematics*, 13(3-4):221–229, 1983.

[10] Ramez Elmasri, Gene T. J. Wuu, and Yeong-Joon Kim. The time index: An access structure for temporal data. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 1–12, August 1990.

[11] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. II.* John Wiley, 1970.

[12] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[13] Charles L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA 15213, July 1981.

[14] Gaston H. Gonnet, J. Ian Munro, and Derick Wood. Direct dynamic structures for some line segment problems. *Computer Vision, Graphics, and Image Processing*, 23:178–186, 1983.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, June 1984.

[16] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[17] Eric N. Hanson. Gator: A generalized discrimination network for production rule matching. In *Proceedings of the IJCAI Workshop on Production Systems and Their Innovative Applications*, August 1993.

[18] Eric N. Hanson and Moez Chaabouni. The IBS tree: A data structure for finding all intervals that overlap a point. Technical Report WSU-CS-90-11, Wright State University, April 1990. Also appears as Univ. of Florida CIS-TR-94-040, available at `http://www.cis.ufl.edu:80/cis/tech-reports/`.

[19] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–280, May 1990.

[20] Sara Hedberg and Marilyn Steizner. *Knowledge Engineering Environment (KEE) System: Summary of Release 3.1.* Intellicorp Inc., Mountain View, CA, July 1987.

[21] Inference Corporation, Los Angeles, CA. *ART Reference Manual*, 1990.

[22] Michael A. Kelly and Rudolph E. Seviora. An evaluation of DRete on CUPID for OPS5. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[23] Edward M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–278, 1985.

[24] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proc. AAAI National Conference on Artificial Intelligence*, pages 42–47, August 1987.

[25] William Pugh. A skip list cookbook. Technical Report CS-TR-2286, Dept. of Computer Science, Univ. of Maryland, July 1989.

[26] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[27] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addision Wesley, Reading, Mass., 1990.

[28] M. Stonebraker, M. Hearst, and S. Potaminos. A commentary on the POSTGRES rules system. *SIGMOD Record*, 18(3):5–11, September 1989.

[29] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[30] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First Annual Conference on Expert Database Systems*, pages 353–364, April 1986.

[31] Michael Stonebraker, Eric Hanson, and Spiros Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[32] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):125–142, March 1990.

[33] Jennifer Widom, Roberta J. Cochrane, and Bruce G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 1991.

[34] Nicklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1986.