A Predicate Matching Algorithm for Database Rule Systems*

Eric N Hanson[1,2] *Moez Chaabouni*[2]
Chang-Ho Kim[2] *Yu-Wang Wang*[2]

[1] USAF Wright R & D Center
WRDC/TXI
Dayton, OH 45433

[2] Wright State University
Dept of Computer Science
Dayton, OH 45435

Abstract

Forward-chaining rule systems must test each newly asserted fact against a collection of predicates to find those rules that match the fact Expert system rule engines use a simple combination of hashing and sequential search for this matching We introduce an algorithm for finding the matching predicates that is more efficient than the standard algorithm when the number of predicates is large We focus on equality and inequality predicates on totally ordered domains This algorithm is well-suited for database rule systems, where predicate-testing speed is critical A key component of the algorithm is the *interval binary search tree* (IBS-tree) The IBS-tree is designed to allow efficient retrieval of all intervals (e g range predicates) that overlap a point, while allowing dynamic insertion and deletion of intervals The algorithm could also be used to improve the performance of forward-chaining inference engines for large expert systems applications

1 Introduction

Efficient testing of rule predicates is critical for good performance of forward-chaining rule systems Extensive research has been done on processing rule conditions efficiently, including development of the Rete algorithm [For82], a modified version of Rete called TREAT [Mir87], and extensions to the Rete algorithm to exploit parallelism [KS89] In this paper, we investigate an important part of the rule condition testing problem testing a collection of predicates to see which of the predicates match a single fact In database terminology, which we will use hereafter, this is the problem of testing a tuple to see which of a collection of single-relation selection conditions match the tuple

This paper does not address the issue of how join predicates 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 We are given a database containing a set of n relations, R_1 R_n, and m production rules (triggers), r_1 r_m Rules are of the form

if *condition*
then *action*

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 $1 \leq j \leq n$ We assume 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 The general form of a predicate for purposes of this discussion is a conjunction of the following form

$$P_i \equiv (\text{the tuple } t \text{ is in relation } R_j) \wedge C_1 \wedge C_2 \wedge \quad C_q$$

where each C_j, $1 \leq j \leq q$, is one of the following

$C_j \equiv const_1\ \rho_1\ t$ attribute $\rho_2\ const_2$
$C_j \equiv t$ attribute $= const_1$
$C_j \equiv$ function(t attribute)

In addition, $const_1 \leq const_2$, both $const_1$ and $const_2$ are drawn from the domain of legal values for t attribute, and ρ_1 and ρ_2 are one of $\{<, \leq\}$ Equality predicates are a special case of interval predicates, but since they are so common, they are listed separately Open intervals are specified by setting $const_1$ or $const_2$ to $-\infty$ or $+\infty$, respectively For predicate clauses of the form "function(t attribute)," nothing is assumed about the function except that it returns true or false

Some example predicates will be shown below, based on this relation schema

EMP(name, age, salary, dept)

Here are some examples of predicates on tuples of the relation EMP

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 the collection of predicates described above, and a tuple t, the predicate testing problem is to determine exactly those P_i's that match t One approach to testing predicates is to use a predicate index Many approaches to the predicate indexing problem have been developed In Section 2 we discuss these methods in order of increasing complexity Section 3 covers pragmatic considerations for predicate indexing regarding characteristics of data in relational database applications, and the expected characteristics of rules Section 4 introduces an alternate predicate indexing algorithm called the *interval binary search tree* (IBS-tree) which we argue is

*This work was supported by the Air Force Office of Scientific Research under grant number AFOSR-89-0286

readily implementable, and should perform well in realistic database rule processing applications Section 5 presents an analysis that shows the performance characteristics of the method described in Section 4 Finally, Section 6 summarizes and presents conclusions

2 Review of Predicate Indexing Methods

Predicate indexing methods range from simple sequential testing to use of complex geometric data structures Below we list some alternatives that have been proposed for predicate indexing in a DBMS, in order of increasing complexity Parallel predicate indexing methods are not considered since our primary focus is a fast uniprocessor implementation For each method, we discuss what is done when a tuple is modified or inserted into the DBMS

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 [For81, Mir87] 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 [SSH86, SHP88], involves treating a predicate clause like a query, and running the standard query optimizer to produce an access plan for the query [S*79] 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 [Gut84] and R+-tree [SSH86] 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) normally refer to only one or two attributes, and rarely more than five [Col89] Collections of low dimension predicates like these are not small closed regions Rather, they are "slices" through space that overlap extensively Spatial data structures, particularly R-trees and R+-trees, index regions like these poorly, giving slow search performance

3 Practical Considerations for Predicate Indexing in a DBMS

Numerous database rule systems have been proposed recently, including Ariel [Han89], RPL [DE88], the POSTGRES rules system [SHP88], HiPAC [DBB*88], and DIPS [SLR89] We envision that applications built using systems like these will be primarily data management applications, enhanced with rules to provide improved data integrity, monitoring 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 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 [BO89], 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 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 We discuss such a predicate index in the next section

4 A High-Performance Predicate Indexing Method

In this section we introduce 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, be able to return a list of all the predicates that match a tuple t from a relation R We want the algorithm to 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 online

In the algorithm we propose, the system builds an index which has at the top level a hash table, using relation names as keys, similar to high-performance implementations of production systems mentioned previously 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 a balanced IBS-tree which allows efficient searching to determine which interval and equality predicates match a value For predicates that are a conjunction of selection clauses, if there is an indexable clause, the most selective one is placed in the IBS-tree (selectivity estimates are obtained from the query optimizer) 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 Below, we focus in more detail on the IBS-tree

4.1 Dynamic Indexing of Intervals

The IBS-tree was motivated by the need to efficiently find all points, intervals, and open-ended intervals that match a particular query value in a dynamic environment where predicates can be added and deleted on-line Data structures for indexing intervals in a static environment where all intervals are known in advance include *segment trees* and *interval trees* [Sam88, Sam90] In the database rule system environment, segment trees and interval trees are not adequate because they do not allow dynamic insertion and deletion of predicates

A data structure that can index intervals dynamically is the *priority search tree* [McC85] An advantage of the priority search tree over the IBS-tree is the priority search tree requires only $O(N)$ space to index N intervals, while as we shall see in Section 4, the IBS-tree requires $O(N \log N)$ space in the worst case (but $O(N)$ in the best case) However, the priority search tree appears more complex to implement than the IBS-tree Moreover, the IBS-tree can directly accommodate multiple intervals with the same lower bound, which the interval tree cannot do To handle intervals with the same lower bound, priority search trees must use a special transformation from pairs with non-unique lower bounds to pairs with unique lower bounds This transformation is not trivial, and it must be created for each different data type to be indexed In contrast, IBS-trees work without modification on any totally ordered domain for which the comparison operators $\{<, =, >\}$ are defined — no additional code is needed

One-dimensional R-trees can also index intervals dynamically [Gut84] However, due to their generality, and the indexing heuristics required, R-trees are challenging to implement R-trees also require only $O(N)$ space Their performance should be good for intervals with low overlap, but when there is heavy overlap, search performance can worsen significantly Also, R-trees cannot accommodate open intervals

Below, we show how a binary search tree can be augmented to index intervals, resulting in the IBS-tree Then, we discuss extensions that allow the tree to remain dynamically balanced

4.2 Interval Binary Search Trees

In this section we introduce a method for augmenting a binary search tree with additional information to make it possible to rapidly find all intervals that overlap a point The IBS-tree can accommodate points and open intervals as well as closed intervals Nodes in the tree have the following form

Value	a data value representing the end point of an interval or the constant in an equality predicate
$>$	a set of interval identifiers
$=$	a set of interval identifiers
$<$	a set of interval identifiers
left	subtree holding all nodes with values less then Value
right	subtree holding all nodes with values greater than Value

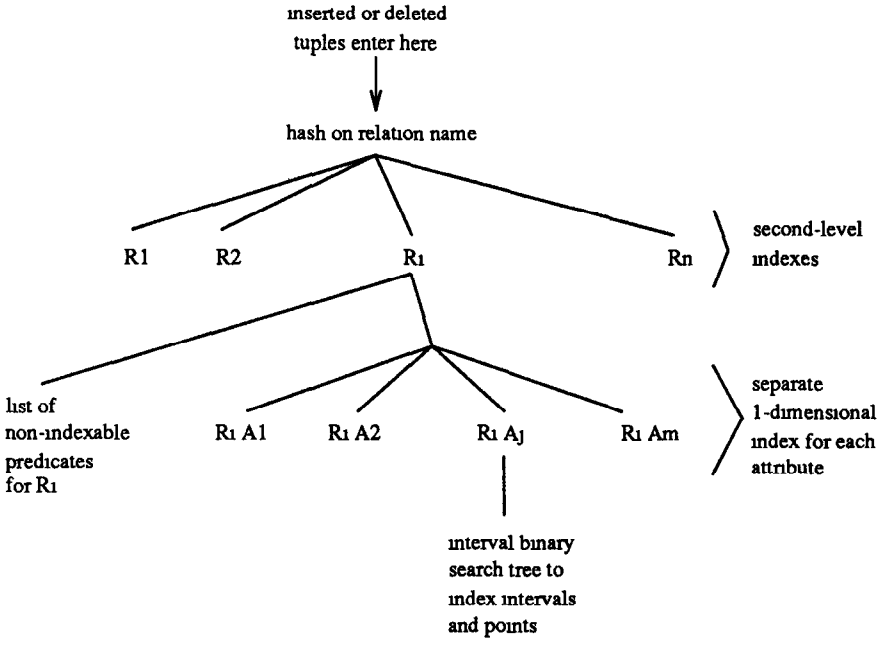

Figure 1 High-level diagram of predicate indexing scheme

The Value, left, and right entries of these nodes mean the same as they do in a standard binary search tree The "=" slot of each node contains a set of interval identifiers If an interval identifier I is contained in the = slot of a node with value V, that implies that I overlaps V The "<" slot also contains a set of interval identifiers If an interval identifier I appears in the < slot of a node with value V, that implies that any value X *that would be inserted into the left subtree of* V lies within interval I The meaning of the ">" slot is symmetric to that of the < slot

Nodes of an IBS-tree are represented graphicly using four boxes organized in the shape of an upside-down "T " The upper box contains the value for the node The lower three boxes contain the <, = and > sets, ordered from left to right An example set of intervals and the IBS-tree for those intervals is shown in Figure 2

Let P denote an interval predicate, and let PC_l and PC_r be the left and right boundaries of P, respectively Let $P\rho_l$ and $P\rho_r$ denote the comparison relations (one of <, $\leq$) for the boundaries of P To insert P into an IBS-tree with root R, the insertPredicate procedure is called which inserts the left end of the interval by calling addLeft(P, R) and the right end by calling addRight(P, R)

Below we will define addLeft and addRight These functions require the ability to determine whether everything in the right (or left) subtree of R will lie within P To help perform this test, we use functions rightUp(R) and leftUp(R) These functions are defined as follows

- rightUp(R) the lowest ancestor of R in the tree that contains R in its left subtree
- leftUp(R) the lowest ancestor of R in the tree that contains R in its right subtree

To find leftUp or rightUp for a node R, traverse upward from R and record leftUp or rightUp as necessary The procedure addLeft is shown in Figure 3 It recursively descends the tree, placing marks in the <, = and > slots of nodes as appropriate, and inserting a node in the tree if no node with the value of the interval's left boundary yet exists The procedure first tests to see if R is null, and if so makes R point to a new tree with one node containing a value equal to the left boundary of the interval Next, it checks for one of three possible cases

Case 1: If the value in R equals the left boundary of the interval, check to see if everything in the right subtree of R will lie within the interval If so, insert the identifier of the interval into the > slot of R Then, if the left boundary of the interval is defined using $\leq$, put the interval identifier in the = slot of R

Case 2: If the value of R is less than the interval's left boundary, call addLeft on the right subtree of R

Case 3: If the value of R is greater than the interval's left boundary, if the value lies in the interval, add the interval identifier to the = slot of R Next, if everything in the right subtree of R will lie in the interval, add the interval identifier to the > slot of R Finally, call addLeft on the left subtree of R

The procedure addRight(P, R) is symmetric to addLeft(P, R), so discussion of addRight is omitted

The above shows how to insert intervals into the tree In order to search the tree to find all intervals in the tree rooted at R that overlap a point X, and return a set of those predicates in S, the algorithm findIntervals(X,R,S) shown in Figure 4 is used The function findIntervals is called initially with arguments X, R and an initially empty set S If R is null, then no more matches are found and

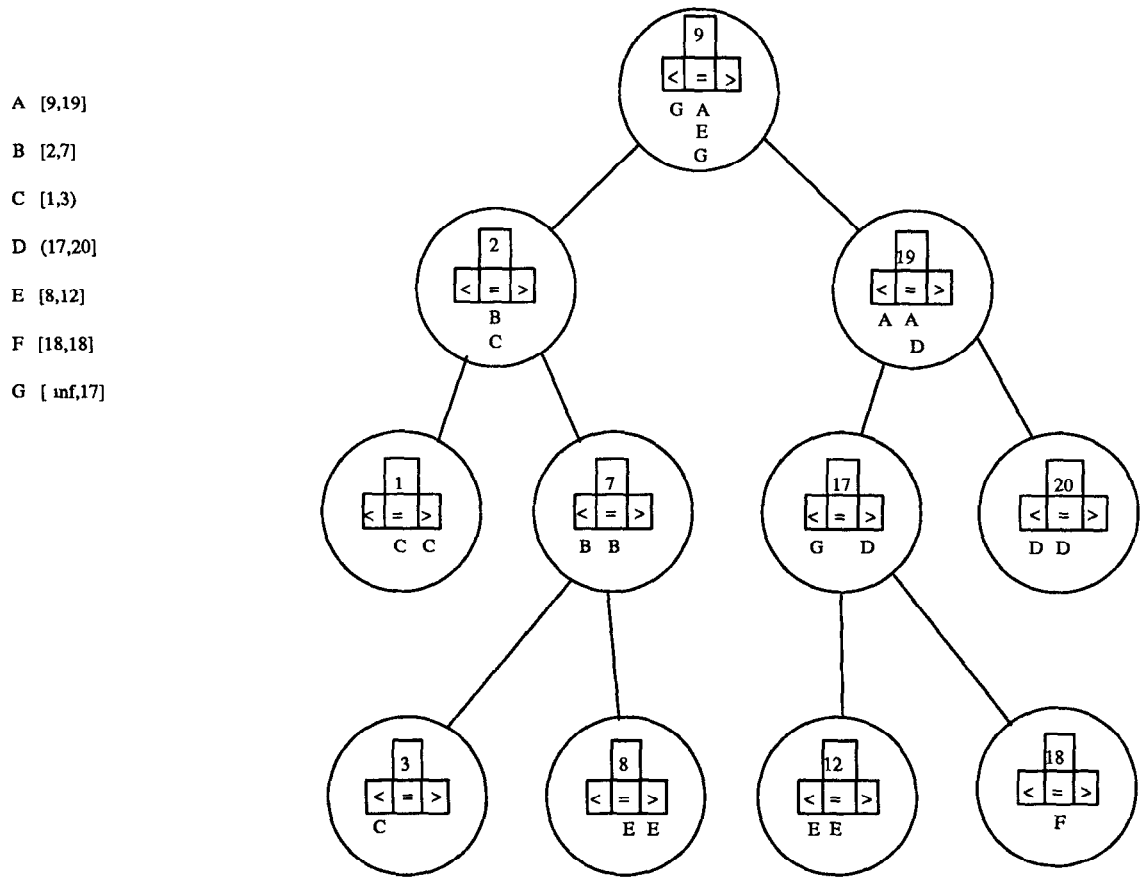

Figure 2 Example interval binary search tree for intervals shown

findIntervals returns If X is equal to the Value entry of node R, the elements of the $=$ set for R are added to the match set S If X is less than the Value of node R, the $<$ set of R is added to S, and the left subtree of R is searched If X is greater than the Value of node R, the $>$ set of R is added to S and the right subtree or R is searched

Deletion of nodes in an IBS-tree occurs only when an interval is deleted To delete an interval I, first delete all markers for I from the tree, using the reverse of the procedure for insertion Then delete the left and right endpoints of I if no other intervals have the same endpoint If an endpoint x is to be deleted, the following procedure is used

If x has no null child,

> Locate the predecessor of x by following right pointers from left(x) until finding a node with a null right child Call this predecessor node y Delete all markers set for intervals with end point y from the tree Store these intervals in temporary set T Swap the *values* of x and y, leaving the markers in their former locations

Make the pointer from the parent of x that points to x now point to either the non-null child of x, if there is one, or else to null Discard x Reinstall the markers for intervals in the set T

A justification of the correctness of this deletion procedure is given in [HC89] This concludes the discussion of basic operations on IBS-trees We now turn to methods for keeping IBS-trees balanced

4.3 Balancing Interval Binary Search Trees

The cost of the findIntervals algorithm depends on the height of the IBS-tree The algorithm for IBS-trees described above does not guarantee that the tree will be balanced Several balanced binary tree schemes have been proposed, including AVL trees [AL62], balanced binary trees (or red-black trees) [Bay72, GS78] and self-adjusting binary trees [Tar83] A common theme in these algorithms is the use of *rotations* to rebalance the tree In particular, during rebalancing operations, the balanced binary tree mechanisms cited make use of the single and double rotations shown in Figure 5 In the figure, lower case letters represent internal nodes, and upper case letters represent subtrees

There are symmetric variants of both single and double rotations which are not shown A double rotation is merely two applications of a single rotation Hence, to balance IBS-trees, all we need is a method for adjusting the marks on an

```
addLeft(P, R)
    do
        if R = Null
            set R to point to a new node with Value = P C_l fi
        if R Value = P C_l
            if everything in the right subtree of R will
                lie within P (i e rightUp(R) ≤ P C_r)
                    add the identifier of P to the '>' slot of R fi
            if P ρ_l is '≤'
                add the identifier of P to the '=' slot of R fi
        else if R Value < P C_l
            addLeft(P, R right)
        else if R Value > P C_l
            if R Value < P C_r (i e R Value is between P C_l and P C_r)
                add the identifier of P to the '=' slot of R fi
            if everything in the right subtree of R will
                lie within P (i e rightUp(R) ≤ P C_r)
                    add the identifier of P to the '>' slot of R fi
            addLeft(P, R left)
        fi
    od
```

Figure 3 Procedure to add the left end of an interval to an IBS-tree

```
findIntervals(X,R,S)
    do
        if R = Null        return
        else if X = R Value
            add the '=' set of R to S
            return
        else if X < R Value
            add the '<' set of R to S
            findIntervals(X,R left,S)
        else (X is > R Value)
            add the '>' set of R to S
            findIntervals(X,R right,S)
        fi
    od
```

Figure 4 Procedure for finding intervals that overlap a point X

IBS-tree during a single rotation so that the resulting tree is also a correct IBS-tree

Consider the single rotate-right operation shown in Figure 5 (a) and (b) The subtrees C and D, and the subtree rooted at x are unaffected during the operation, so no adjustment to them is required However, nodes y and z are modified to have different subtrees, so we must consider how to adjust the marks in the $<$, $=$ and $>$ fields of both after the rotation to leave the IBS-tree in a correct state

The following modifications to the marks are required during a rotation

1 Copy every mark from the $<$ slot of z to the $<$ and $=$ slots of y (this is necessary since having a mark for a predicate P in the $<$ slot of z implies that P matches every value in the left subtree of y, as well as y itself)

2 If a mark is in the $>$ slot of y but not in the $>$ slot of z before the rotation, then move the mark to the $<$ slot of z after the rotation This is necessary because values in the subtree C are covered by marks in the $>$ slot of y before the rotation, and must be covered by marks in the $<$ slot of z afterwards if they cannot be covered by a mark on y

3 If a mark is in the $>$ slot of y and the $>$ slot of z before the rotation, then remove the mark from the $=$ slot and $>$ slot of z after the rotation (this is necessary to avoid redundant locks on the values in subtree D)

Operations for each mark slot $(<, =, >)$ on affected nodes y and z are summarized in Figure 6

In this section we have demonstrated that we can perform rotations about tree nodes and manipulate marks to restore the tree to be a correct IBS-tree The next section analyzes the performance of a balanced IBS-tree scheme that makes use of rotations

5 Performance Analysis

5.1 Analytical Performance Results

Assume that the AVL-tree scheme is used to maintain the balance of an IBS-tree [AL62] Each interval place $O(\log N)$ markers in the tree, for a worst-case storage requirement of $O(N \log N)$ Searching the tree to find all intervals that overlap a point X requires time $O(\log(N)+L)$ where N is the number of intervals indexed in the tree, and L is the number of intervals that overlap X This follows since traversing a path from root to leaf in the tree requires $O(\log N)$ time using a balanced tree scheme, and we must spend $O(1)$ time examining each of the L intervals retrieved

The cost of insertion and deletion of intervals in the tree is somewhat more difficult to calculate An important factor in

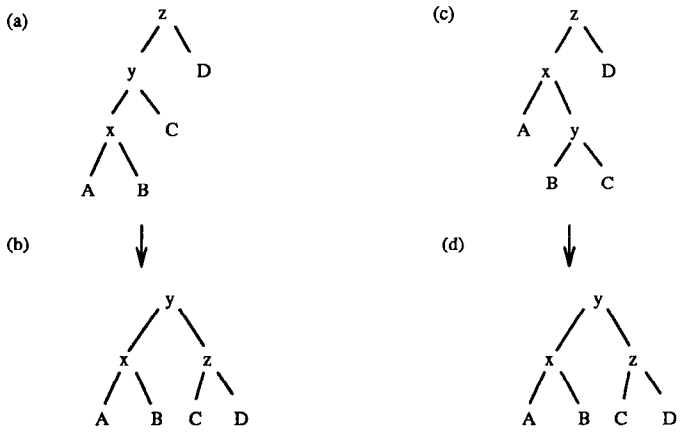

Figure 5 Single rotation (a,b) and double rotation (c,d)

	y	z
$<$	copy marks from $<$ of z	get marks from $>$ of y if necessary (see $> y$)
$=$	copy marks from $<$ of z	delete marks that are in both $>$ of y and $>$ of z
$>$	move marks to $<$ of z unless they appear in $>$ of z	delete marks that are in both $>$ of y and $>$ of z

Figure 6 Modifications to marks of each slot on affected nodes y and z required during a single rotation

the cost of insertions and deletions is the cost of doing a rotation Each insertion requires $O(1)$ rotations to rebalance the tree, and a deletion requires $O(\log N)$ rotations In [HC89] it is shown that the average cost of a rotation in an IBS tree is $O(\log N)$ Inserting an interval into an IBS tree requires $O(1)$ rotations for time $O(\log N)$, plus $O(\log N)$ insertions of markers into mark sets, each of which costs $O(\log N)$ if mark sets are maintained using auxiliary binary search trees, for a total cost of $O(\log^2 N)$ per insertion Deletion of an interval from an IBS tree also requires time $O(\log^2 N)$ for removing the markers from the tree, and time $O(log^2 N)$ for doing $O(\log N)$ rotations at a cost of $O(\log N)$ each, for a total time of $O(\log^2 N)$

The above discussion is an average case analysis of the cost of updating an IBS-tree with no restrictions on the width of intervals, or the extent to which intervals overlap An intriguing phenomenon is that when intervals in the tree do not overlap, only $O(N)$ markers are placed in the tree for a storage requirement of $O(N)$, and significantly reduced update cost Derivation of this result is left to the interested reader Since in many practical applications intervals have limited overlap, this gives hope that the actual time and space requirements for IBS-trees will be somewhat lower in practice than indicated by the analysis in this section

5.2 Empirical Performance Results

To get empirical figures on the performance of IBS-trees, the algorithm was implemented in C++ on a Sun SPARCstation 1 computer The balancing scheme using rotations was not implemented, but as with ordinary binary search trees, the tree is normally balanced if data is inserted in random order A series of IBS trees were created which contained N predicates for N between 0 and 1,000 A fraction a of predicates were simple points of the form *attribute = constant*, and the remaining fraction $1 - a$ were closed intervals The points and interval boundaries were drawn randomly from a uniform distribution of integers between 1 and 10,000 The length of the intervals was drawn randomly from a uniform distribution of integers between 1 and 1,000 The average times to insert a predicate for values of a=0, 5 and 1, and increasing values of N are shown in Figure 7 The average insertion cost was measured as the time to insert N predicates in an initially empty index, divided by N Since the test does not reflect any balancing cost, insertion times for balanced IBS-trees will be higher than shown in Figure 7 The average search time to find all predicates that match a value is plotted in Figure 8 for a=0, 5 and 1, and increasing values of N

As a basis of comparison for the IBS-tree algorithm, the cost of finding the predicates that match a value by traversing a linked list of predicates and testing each one against the value is shown in Figure 9 The cost curve for sequential search is always higher than for the IBS-tree, showing that the IBS-tree has quite low overhead

As expected, the insertion and search time curves for the IBS-tree both show logarithmic increase in search time as the number of intervals increases The difference between

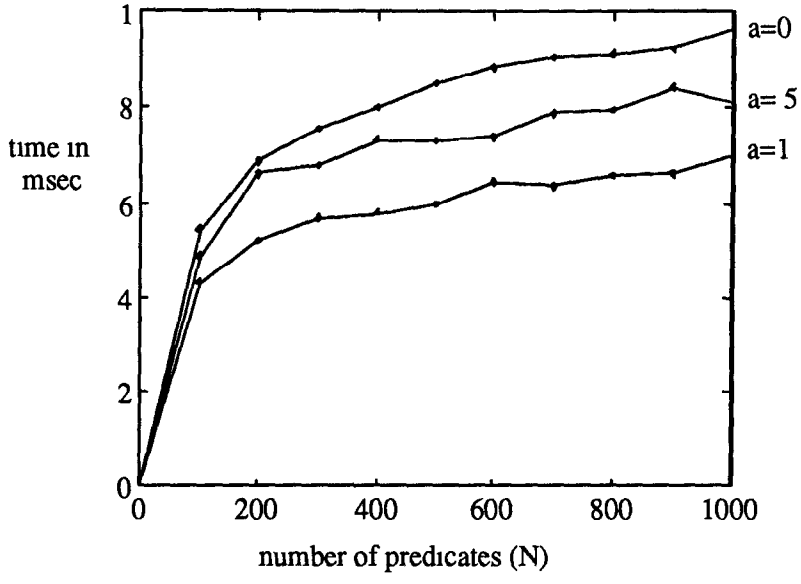

Figure 7 Average IBS-tree insertion times for a=0, 5 and 1

the curves for the different values of a (0, 5 and 1) are small, particularly for search time

When the IBS-tree 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, we can get an estimate for the time required to find matching predicates 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 IBS-tree for one attribute is approximately 13 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 1 In this scenario, the cost to search to find the partially matching predicates is the following

cost = hash cost
+ number of attributes searched
IBS-tree search cost
+ non-indexable predicate test cost

This yields the following numeric expression for cost

$$\begin{aligned} \text{cost} &= 1 + 15\tfrac{1}{3} \quad 13 + (1 - 9) \quad 02 \quad 200 \\ &= 1 + 5 \quad 13 + 4 = 1\ 1 \text{ msec} \end{aligned}$$

Since there are 200 predicates per relation, and the selectivity of the predicate clauses is 1, that means that 1 200 = 20 predicates must be tested after the initial search The time to test these is 05 20 = 1 msec Thus, the total time for predicate testing is 1 1 + 1 = 2 1 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 1 Given that this is a per-tuple CPU cost, the time is substantial, but should not be prohibitive Of course, these are CPU-only costs, and any increase in CPU speed will cause the predicate testing time to scale down accordingly

6 Conclusion

In this paper we have introduced a discrimination network structure for finding all members of a set of single-relation selection predicates that match a tuple 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 to 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 systems applications that appear to require a very large number of rules can be redesigned to use a small number of rules plus additional

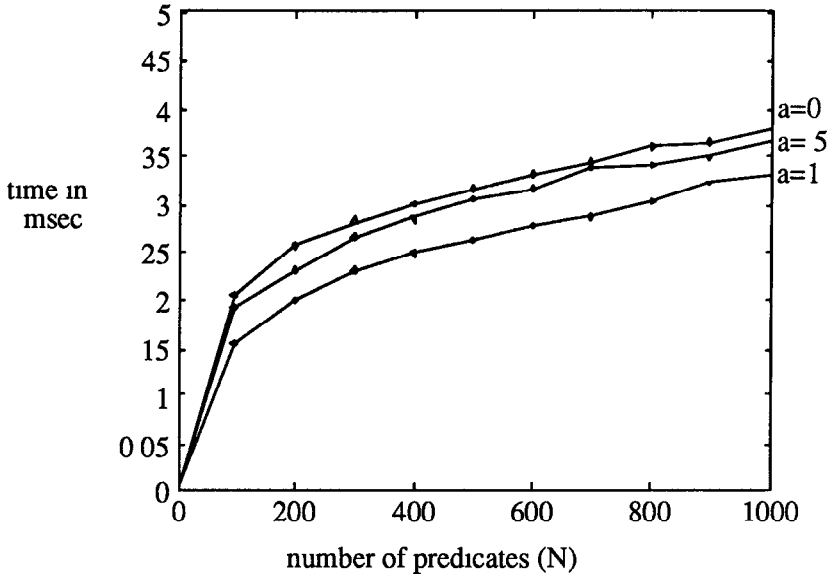

Figure 8 Average IBS-tree search times for a=0, 5 and 1

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 binary search tree, an extended binary search tree for indexing both interval and point data The IBS tree is related to the segment tree and interval tree, but in addition allows dynamic insertion and deletion of intervals and points while remaining balanced Analytical and empirical results show that the insertion and search performance of the IBS tree is almost as good as for an ordinary binary search tree The IBS tree or variations of it may be useful for other applications besides testing predicates, including VLSI CAD tools, geographic information systems, and other applications that deal with geometric data The IBS tree is useful anywhere an index for intervals is required which must be dynamically updatable

Although the intent of this paper was not to investigate parallelism, the algorithm proposed can easily be made to run significantly faster on a course-grain parallel machine such as a shared-memory multi-processor Parallelism can be achieved by searching the second-level index on each attribute of a tuple simultaneously, devoting a processor per attribute 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 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)

Considering topics for further research, an interesting area to investigate would be to implement several different techniques for dynamically indexing intervals, including 1-dimensional R-trees, IBS-trees, and priority search trees, and then compare their implementation complexity and time and space requirements Also, in the future we plan to work on developing an efficient structure to handle the join portion of rule predicates The discrimination network described in this paper will be used as the first layer of a two-layer network which will test both the selection and the join conditions of rules This two-layer approach is being implemented in the rule processing engine of the Ariel database system

References

[AL62] G M Adel'son-Vel'skii and E M Landis An algorithm for the organization of information *Soviet Math Dokl*, 3, 1962

[Bay72] R Bayer Symetric binary B-trees data structure and maintenance algorithms *Acta Informatica*, 1, 1972

[BO89] Virginia E Barker and Dennis E O'Connor Expert systems for configuration at Digital XCON and beyond *CACM*, 32(3), March 1989

[Col89] Larry Collins Informal survey of relational database applications at Wright-Patterson AFB 1989 (personal communication)

[DBB*88] U Dayal, B Blaustein, A Buchmann, et al The HiPAC project Combining active databases and timing constraints *SIGMOD Record*, 17(1) 51–70, March 1988

[DE88] 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

[For81] Charles L Forgy *OPS5 User's Manual* Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA 15213, July 1981

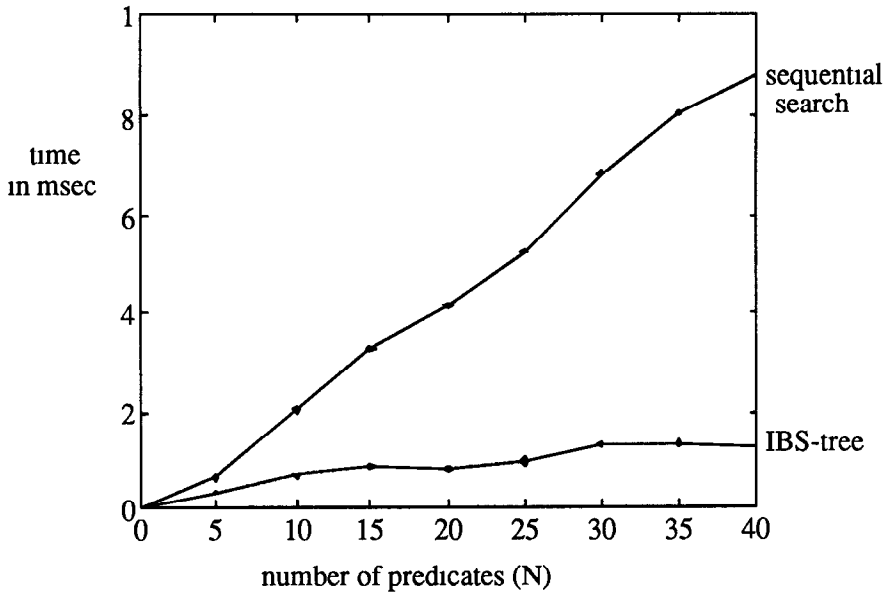

Figure 9 Predicate test cost for IBS-tree and sequential search

[For82] C L Forgy Rete A fast algorithm for the many pattern/many object pattern match problem *Artificial Intelligence*, 19 17–37, 1982

[GS78] L J Guibas and R Sedgewick A dichromatic framework for balanced binary trees In *Proc 19th Annual IEEE Symposium on Foundations of Computer Science*, 1978

[Gut84] A Guttman R-trees A dynamic index structure for spatial searching In *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, June 1984

[Han89] Eric N Hanson An initial report on the design of Ariel a DBMS with an integrated production rule system *SIGMOD Record*, 18(3), September 1989

[HC89] Eric N Hanson and Moez Chaabouni *The IBS Tree A Data Structure for Finding All Intervals That Overlap a Point* Technical Report, Wright State University, March 1989

[KS89] 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

[McC85] Edward M McCreight Priority search trees *SIAM Journal of Computing*, 14(2) 257–278, 1985

[Mir87] Daniel P Miranker TREAT a better match algorithm for AI production systems In *Proceedings of AAAI 87 Conference on Artificial Intelligence*, pages 42–47, August 1987

[S*79] P Selinger et al Access path selection in a relational database management system In *Proceedings of the 1979 ACM-SIGMOD International Conference on Management of Data*, June 1979

[Sam88] Hanan Samet Hierarchical representations of collections of small rectangles *ACM Computing Surveys*, 20(4) 271–309, December 1988

[Sam90] Hanan Samet *The Design and Analysis of Spatial Data Structures* Addision Wesley, 1990

[SHP88] Michael Stonebraker, Eric Hanson, and Spiros Potamianos The POSTGRES rule manager *IEEE Transactions on Software Engineering*, 14(7) 897–907, July 1988

[SLR89] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid Data intesive production systems the DIPS approach *SIGMOD Record*, September 1989

[SSH86] 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*, April 1986

[Tar83] Robert Endre Tarjan *Data Structures and Network Algorithms* Society for Industrial and Applied Mathematics, 1983