

The Relational Interval Tree

Report by Daniel Bucher

May 30, 2010

Preface

This Report summarizes the main information of the paper [KPS 00] which was published in the year 2000 and tries to explain the idea of the *Relational Interval Tree* understandable. After reading this report you should know how an *RI-tree* looks like and how it works.

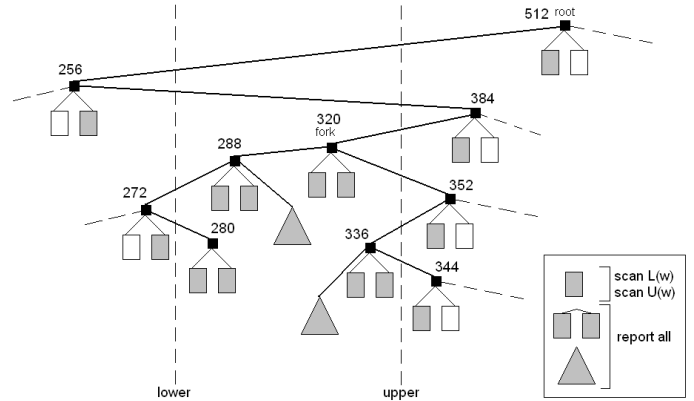


Figure 1: Typical structure of an RI-Tree

1 The Relational Interval Tree

The *Relational Interval Tree* (*RI-tree*) is an implementation which can be integrated in already existing relational database systems.

1.1 Introduction

A typical *Relational Interval Tree* looks like shown in figure 1. Important to recognize at this moment is that the *RI-Tree* has a balanced binary search tree as primary structure and the *root* is always chosen as 2^{h-1} for a range of $2^h - 1$. So in the shown example where the *root* is equal to 512 the range would therefore be 1023. So this means the nodes of the *RI-Tree* have the values from 1 to 1023.

The *RI-Tree* is always traversed by bisection. In the shown example above we can see that because the *root* is 512 its left child has to be 256. The left child of this node would be 128 and the right child is as shown 384. This is equal to $256 + 128$.

Every node has two sorted lists which are called *lowerBound* and *upperBound* and are abbreviated as $L(w)$ and $U(w)$ for the node w .

Every interval is registered at the first node from the *root* it intersects with and so every interval is only stored once and therefore we haven't any redundancy and so we get space complexity of $O(n)$.

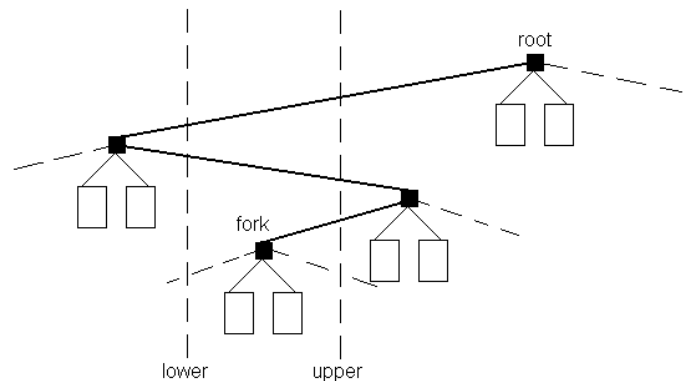


Figure 2: Where the *fork node* lies

1.2 The fork node

As mentioned above every interval (l, u) is registered at the first node w from the *root* which it intersects. This node where the interval is stored is called *fork node* (figure 2). The notation (l, u) means that the lower bound of the interval is equal to l and the upper bound of the interval is equal to u . So it's also possible to insert points by just choosing the lower bound equal to the upper bound which would result in (p, p) for the point p . So keep in mind that the first node w which fulfils the property $l \leq w \leq u$ becomes the *fork node* of the interval. Here's an example in figure 3 where we insert the interval $(300, 380)$ into our *RI-Tree* with *root* 512.

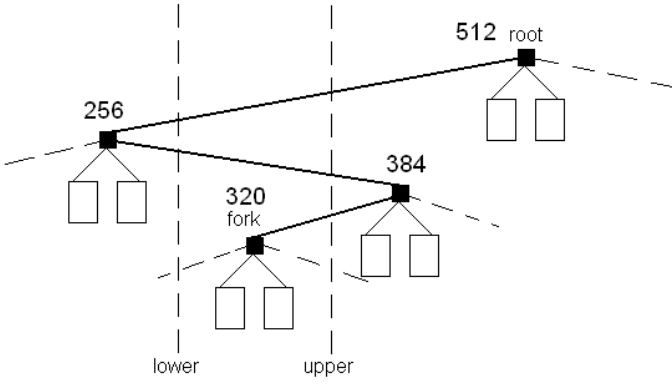


Figure 3: Inserting the interval (300, 380)

1.3 Structure of the Relational Interval Tree

The original *Interval Tree* has three structures. The primary structure is the binary search tree the secondary structure are the lists of each node containing the lower and upper bounds and the tertiary structure is a *linked list* of all the non empty lists.

The already built-in composite indexes of an RDBMS allow us the combine the secondary and tertiary structure in a relational representation which gives us the desired space complexity of $O(n/b)$ for n intervals and block size b .

To map the secondary structure to a relational schema we do it as follows: $L(w)$ is the list $\{l_1, \dots, l_{n_w}\}$ with all the lower bounds of the n_w intervals and so we can represent the same information by the tuple set $\{(w, l_1), \dots, (w, l_{n_w})\}$. Now we can take the union over all nodes w which gives us the relation $(node, lower)$. Now we do the same again for $U(w)$ which is the list $\{u_1, \dots, u_{n_w}\}$ holding all the upper bounds. Then we transform this into the tuple set $\{(w, u_1), \dots, (w, u_{n_w})\}$. That union gives us the relation $(node, upper)$. The two relations together give us the exact information that we had before in the secondary structure.

These two relations $(node, lower)$ and $(node, upper)$ can now be organized efficiently by the already built-in composite indexes of the RDBMS. This is a very good solution because it's very hard to outperform these indexes by user defined structures because these indexes typically are implemented very robust and highly tuned by using a *B+ -tree* and they already obey the transaction semantics so that we don't need to adapt anything. Also key compression techniques are used to avoid redundancy for equal node values w and as the indexes only manage the nonempty nodes we don't have to take care about the tertiary structure since it is already comprised in

the indexes.

To prepare a given interval relation for the *RI - tree* we just add an attribute node and two indexes. This results in a relational schema containing the attributes $(node, lower, upper, id)$ which can be done with the *data definition language (DDL)* statements in SQL as shown below:

```
CREATE TABLE Intervals
    (node int, lower int, upper int, id int);
CREATE INDEX lowerIndex
    ON Intervals (node, lower);
CREATE INDEX upperIndex
    ON Intervals (node, upper);
```

1.4 Updates in Relational Interval Trees

Now the registered intervals are managed by our relational schema but our remaining task of the primary structure is to organize the data space such that we can perform insertions and queries. The original *Interval Tree* is only able to deal with a static set of boundary points but we need to be able to manage also dynamic ones because we want to be able to insert and delete intervals when we need to and those bounding points aren't known in advance. Today's application also need the possibility to expand the data space. For all this we need a new general and adaptable technique to manage all of this.

The *RI - Tree* solves this by virtualizing the primary structure and so doesn't materialize any set of nodes. This allows to insert and delete any interval at anytime without the need to reorganize the structure because now the entire range from 1 to $2^h - 1$ for an in advance given $h \geq 0$ is guaranteed and we aren't restricted to any bounding points from now on.

The basic version chooses the *root* node as 2^{h-1} which in my opinion is a good choice for the range of 2^h and traverses the rest of the tree via bisection which does only use simple integer arithmetics but no I/O operations. Recall that an interval (l, u) is registered at the topmost node w for that it holds that $l \leq w \leq u$ and this node is called a *fork node*. If you can't remember where the *fork nodes* are located in the *RI - Tree* please have again a look at figure 2. The *RI - tree* allows in contrast to the original *Interval Tree* that intervals begin and end at inner nodes. They aren't forced to do so only at leaves. As mentioned before also points can be represented. This is done by degenerated intervals (p, p) where p is the point we want to represent.

To update the *RI-Tree* and to insert a new interval (l, u) all we have to do is to search for the *fork node* and to insert its l into $L(w)$ and its u into $U(w)$. We do this by inserting the relational indexes into the schema from before (the table we created in section 1.3).

The procedure to find the *fork node* is shown below. Notice that the recursion is controlled by a decreasing step and not by the depth of the tree itself. This however seems to work faster.

```
FUNCTION int forkNode (int lower, int upper){
    int node = root;
    for (int step = node/2; step >= 1;
        step /= 2)
        if (upper < node) node -= step;
        elseif (node < lower) node += step;
        else break;
    return node;
}
```

When the *fork node* is found the interval can be inserted efficiently into the relational indexes by the DBMS itself. We only need the SQL statement which is shown below and can also be used for deleting an interval from the database. This is very useful because today's RDBMS typically execute this two operations in $O(\log_b n)$ I/O operations when the database holds n intervals.

```
INSERT INTO Intervals
VALUES (forkNode(:lower, :upper),
       :lower, :upper, :id);
```

Again an example of inserting interval (300, 380) into an example tree with *root* 512 could be found in figure 3.

1.5 Dynamic Expansion of the Data Space

Because the basic version has a fixed range of $2^h - 1$ values and the root is chosen as 2^{h-1} the tree has always a height of h and so the CPU time complexity grows linearly with h .

The *RI-tree* allows to adjust the height of the tree according to the actual data distribution. Moreover it makes it possible to expand the upper bound dynamically which is a typical requirement for temporal applications because time also increases but also the expansion of the lower bound is possible as we will see.

The height of the tree is affected by the value we choose for the root and so affects directly the depth which all the algorithms have to go down to find the interval they are looking for because all of them have to start at the *root* value. So we want to be able to control the minimum tree height and that's why we need some more parameters which are *root*, *offset*, *leftRoot*, *rightRoot* and *minstep*. Lets see what they are good for.

Root: If it's possible to adapt the *root* dynamically we get two major advantages. Firstly we can keep the tree height as minimal as possible and secondly we are able to expand the tree at its upper bound as soon as new intervals arrive. When we choose the root value as 2^h we are able to manage efficiently all intervals which have $0 < \text{lower}$ and $\text{upper} < 2^{h+1}$ where $h = \lfloor \log_2(\max\{\text{upper}\}) \rfloor$. This we can achieve by adjusting the *root* parameter at every insertion which is possible in $O(1)$ and doesn't affect the existing entries.

Offset: We have the optimal *root* height when our data space starts at 1. But what when our intervals don't start at 1? It could be possible that our intervals span the range of $[x_1, x_N]$ where $x_1 \gg 1$ and so our intervals may start far away from the origin. So if we don't take this into account our left side of the tree may be nearly empty and we would get a tree height of $\lfloor \log_2(x_n) \rfloor$ whereas we would only need a height of $\lfloor \log_2(x_N - x_1) \rfloor$ because our data space only has a length of $x_N - x_1$. We can clearly optimize the tree height. So all we need to do is to shift the intervals until the lower bound x_1 of the first interval is equal to 1. This gives us the optimal *root* height back which is $h_{opt} = \lfloor \log_2(\max\{\text{upper}\} - \min\{\text{lower}\}) \rfloor$. The amount of shift operations needed to achieve this is stored in the parameter *offset*.

LeftRoot and RightRoot: If we would change the *offset* parameter when we receive a new interval which lower bound is smaller than the one we used to calculate the *offset* we would have to do a lot of recalculation for all node values that are already stored in the tree. This effort which would cost us I/O operations of $O(n/b)$ we absolutely want to avoid. That's why the *offset* is fixed after inserting the first interval into the database. So that seems to be a good idea but what if the first interval that's inserted into the database isn't the leftmost one? Then we need a possibility to expand the space at the lower bound and that's where the *leftRoot* and *rightRoot* come into play.

We choose our global *root* as 0 which links us to the *leftRoot* and the *rightRoot* which can be managed

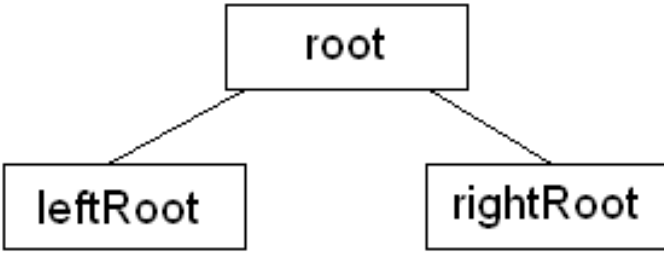


Figure 4: *root* with the child's *leftRoot* and *rightRoot*

independently. Now we have two *root* parameters instead of one. We assign all the negative values to the *leftRoot* and all the positive values to the *rightRoot*. Now we can also manage the expansion of the data space at the lower bound and upper bound separately. Figure 4 shows that now we have the *root* with the child's *leftRoot* and *rightRoot* which can be managed on there own.

Minstep: The parameter *minstep* stores the lowest level i_{min} at which insertions of intervals have occurred. The level 0 is defined as the leaf level. This parameter *minstep* is very useful because no query algorithm has to descend the tree deeper than to the level i_{min} because we know that there's nothing and so the secondary structure of all the lower nodes will be empty anyway.

Below we have the final insertion procedure which also handles the update of all the new introduced parameters. Notice that only the artificial node value is shifted by the *offset* and not the lower and upper bounds. This is very important because we only want to shift the artificial values to get a better tree height. We don't want to falsify the stored values themselves. Initially the *leftRoot* and also the *rightRoot* are set to 0. The *minstep* is set to infinity to make sure that it will be set correctly when we insert the first interval.

```

PROCEDURE insertInterval
  (int lower, int upper, int id){

    // initialize offset and shift interval
    if (offset = NULL) offset = lower;
    int l = lower-offset, u = upper-offset;

    // update leftRoot and rightRoot
    if (u<0 and l <= 2*leftRoot)
      leftRoot = -2^floor((log2(-l)));
    if (0<l and u >= 2*rightRoot)
      rightRoot = 2^floor((log2(u)));

```

```

// descend the tree down to the fork node
int node, step;
if (u<0) node = leftRoot;
elseif (0<l) node = rightRoot;
else /* 0 is fork node */ node = 0;

for (step = abs(node/2);
     step >= 1; step /= 2){
  if (u<node) node -= step;
  elseif (node<l) node += step;
  else /* fork reached */ break;
}

if (node != 0 and step < minstep)
  minstep = step;

INSERT INTO Intervals
  VALUES (:node, :lower, :upper, :id);
}

```

1.6 Analysis of the Tree Height

With the parameters *offset*, *leftRoot*, *rightRoot* and *minstep* we are able to represent a dynamically adjustable primary structure in respect to the cardinality m of the current data space and this in $O(1)$. The tree height of the resulting structure is $\log_2(m) + 1$ inclusive the global root 0. m is given by the following formula:

$$m = \max\{-leftRoot, rightRoot\} / minstep$$

The expansion of the data space is reflected by the range from the *leftRoot* to the *rightRoot* which means from $\min\{lower\}$ to $\max\{upper\}$. The parameter *minstep* shows the granularity of the data space by the smallest interval length $\min\{upper - lower\}$. This value is increased by 1 because we have also allowed degenerated intervals to represent points. But *minstep* can still be greater then $\min\{upper - lower + 1\}$ because also small intervals can be registered at high nodes like also at the *root* itself. Very important to notice here is that the tree height does never depend on the number of intervals. The tree height is always in $O(\log_2 m)$ where m has the following complexity:

$$m = O\left(\frac{\max\{upper\} - \min\{lower\}}{\min\{upper - lower + 1\}}\right)$$

2 Query Processing

Now that we know how the structure of the *RI-tree* looks like we also want to know how we can use query algorithms to get some information out of it. That's what we will have a look at in this chapter.

2.1 Original Intersection Search

For a better understanding we first have a look at the original query process of the *Interval Tree*. If we are looking for an interval $(lower, upper)$ we first descend from the *root* node down to the *fork* node of the query interval. This means we look for the first node which has $lower \leq w \leq upper$. All the nodes w we traverse on this path lie on the left or on the right side of our query interval as can be seen in figure 5. If $w < lower$ (the node w lies on the left side of the query interval) then all the intervals (l, u) which are registered at this node and also have $lower < u$ intersect with our query interval and need to be reported. This is done by scanning the sorted list $U(w)$ which holds the upper bounds of the registered intervals. Reporting r_w intervals is possible in $O(r_w)$. If we traverse a node w with $upper < w$ (the node w lies on the right side of the query interval) then we do the same thing with $L(w)$. Means we scan the sorted list $L(w)$ and report all the intervals with $l < upper$.

When we have found the *fork* node we continue to descend to the node w which is closest to *lower* we are looking for. When descending from here on only the two cases are possible. Either $w < lower$ as before and so we proceed as before or $lower \leq w$ and then we know that our query interval intersects all intervals which are registered at this node w . Moreover we can immediately report the whole right subtree of node w . Except for the fork node itself because then it would still be possible to have intervals stored there which don't intersect with our query.

Now we are finished with the left half of the tree and need to have a look at the right half. So we descend from the *fork* node down to the node w which is closest to *upper*. Now we can again proceed as before. If $upper < w$ we have to scan $L(w)$ as in the first step and else if $upper \leq w$ we know that we can report the whole left subtree if it isn't the fork node itself.

There is no reason to not use this algorithm for points which are only degenerated intervals with $lower = upper$. This just works fine and is as

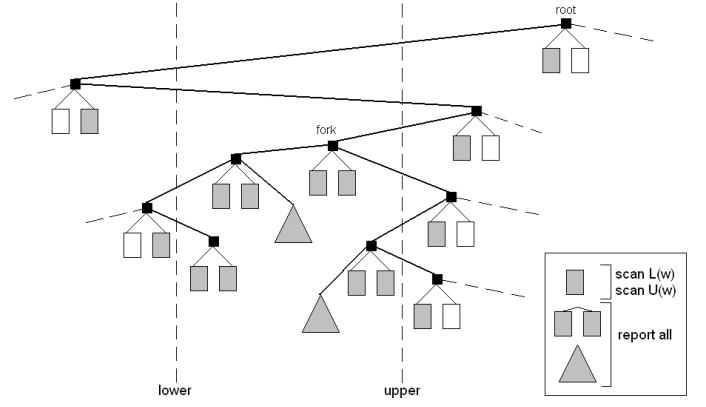


Figure 5: Which intervals are reported

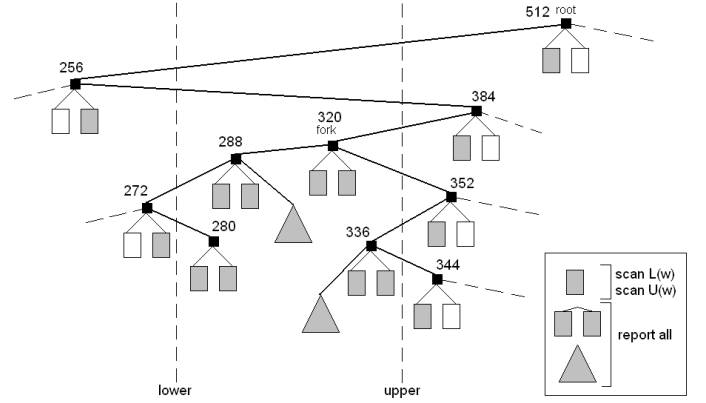


Figure 6: Example for our interval (274, 340)

effective as for interval queries. Figure 5 shows which intervals are reported by the query algorithm. Also it shows which lists $L(w)$ and $U(w)$ have to be scanned and which subtrees can be reported immediately.

Again an example in our example tree for the query interval (274, 340) which is shown in figure 6.

2.2 Translation into a Single SQL Query

The main goal we are trying to achieve is to make use of the built-in relational indexes. We can exactly do this for scanning the lists $L(w)$ and $U(w)$ because this is exactly a range scan over the attributes $(node, lower)$ and $(node, upper)$ which we had defined before as *lowerIndex* and *upperIndex*. We can use both indexes to scan the nodes w between *lower* and *upper*.

We don't scan the lists $L(w)$ and $U(w)$ immediately but we collect the lists of the traversed nodes in temporary lists *leftNodes* and *rightNodes*. In the example in figure 6 the *leftNodes* would be the

nodes 256 and 272 and the *rightNodes* would be 512, 384, 352, and 344. These temporary lists can be managed in the temporary session state and so don't cause any I/O effort which is exactly what we want as few as possible I/O operations. Now when we have those two lists all we need to do is to read out the intersected intervals and that can be done by a single SQL query as shown below:

```
SELECT id FROM Intervals i,
       leftNodes left, rightNodes right
WHERE (i.node = left.node
       AND i.upper >= :lower)
OR (i.node = right.node
    AND i.lower <= :upper)
OR (i.node BETWEEN :lower - offset
    AND :upper - offset);
```

As can be seen in figure 5 the nodes *leftNodes*, *rightNodes* and the nodes between *lower* and *upper* are distinct from each other. That's why we don't need the DISTINCT option in the SELECT clause because all the conditions connected by OR in the code specify disjoint interval sets and therefore there aren't any duplicates.

2.3 Simplified SQL Query

Typically the relational optimizer of a RDBMS splits the OR-query into a set of simpler queries which then are connected by UNION ALL. So in our case this would mean that it would split our query into three new queries. One for *leftNodes*, one for *rightNodes* and one for the nodes between. The query for *leftNodes* is supported by our index *upperIndex* and the query for *rightNodes* is supported by our index *lowerIndex* and so they can't be mixed. But the third query can be supported by either of them and so we can combine this query with the query of the *leftNodes*. It would also be possible to combine it with the *rightNodes* query. After having modified the query our temporary relation *leftNodes* now holds the node *w* which is inserted as a pair (*w*, *w*) and the result of the BETWEEN query is stored as a pair (*lower* - *offset*, *upper* - *offset*). The query for the *rightNodes* remains unchanged. The resulting SQL statement is shown below:

```
SELECT id FROM Intervals i, leftNodes left
```

```
WHERE (i.node BETWEEN left.min AND left.max
       AND i.upper >= :lower)
UNION ALL
SELECT id FROM Intervals i, rightNodes right
WHERE i.node = right.node
       AND i.lower <= :upper;
```

The general execution plan which is for example used by Oracle looks like the following:

```
SELECT STATEMENT
  UNION-ALL
    NESTED LOOPS
      COLLECTION ITERATOR
      INDEX RANGE SCAN UPPER_INDEX
    NESTED LOOPS
      COLLECTION ITERATOR
      INDEX RANGE SCAN LOWER_INDEX
```

2.4 Analysis of the Algorithm

We already observed that the tree height h which is in $O(\log m)$ only depends on the two parameters *leftRoot* and *rightRoot* which determine the quotient m . Now we know that the height is an upper bound for the number of entries in the relations *leftNode* and *rightNode* and is independent of the number n . We have to do an index range scan on *upperIndex* or *lowerIndex* for all the entries in the relations which are $O(\log m)$ entries. The index range scan first searches for the beginning of the range ρ and then scans from there on to report the r_ρ resulting objects from the range. Typical the search takes $O(\log_b n)$ I/O operations for example in a $B + -tree$ index structure in a relational database system with n objects and reporting the r_ρ results for a range ρ takes $O(r_\rho/b)$. This is confirmed by the following theorem and its proof:

Theorem (*Complexity of Query Processing*)

An intersection query on a Relational Interval Tree of height h that returns r results from the n intervals in the tree has an I/O complexity of $O(h \cdot \log_b n + r/b)$

Proof. For each of the $O(h)$ entries in the transient relational *leftNodes* and *rightNodes*, an index search of $O(\log_b n)$ I/O complexity is performed. Scanning and reporting the total of r results requires $O(r/b)$ operations." (From [KPS 00])

2.5 Handling Temporal Intervals

In temporal databases we have also the special values *now* and *infinity* which are valid as upper bounds for interval. If we would manage these new values in separate indexes we would have additional SQL queries which we would have to execute. We absolutely want to avoid this and that's why we manage these values in the *fork nodes*. So we are able to integrate them naturally into the *Relational Interval Tree*. Lets have a look what this two new values specify.

Infinity: We just set the upper bound of a *fork node* which has an infinite interval to MAXINT so we don't need to modify the existing algorithm. To avoid that the tree becomes very high but is nearly empty close to the root, we just assign an artificial value $fork_{\infty}$ to the node of the infinite interval. It will be replaced by MAXINT and inserted into *rightNodes* at query time. So it's still possible to compare the lower bounds of intervals which end at *infinity*.

Now: *infinity* stays constant over time so it was more or less an easy task but *now* changes from time to time. So we need to solve this otherwise. If we just would correct *now* every time it changes we would have to modify the whole tree again and again which would cause a lot of overhead which we want to avoid. So we use again an artificial node value. This time we name it $fork_{now}$ and set it equal to MAXINT - 1. This value $fork_{now}$ we will assign to every interval which upper bound is *now*. At query time $fork_{now}$ will be inserted into *rightNodes* if and only if $lower \leq now$. So the SQL query will then check the lower bounds of the *now*-intervals against the upper bound of the query interval as it would have done with any other interval.

3 Object-Relational Wrapping

The only requirement for our *Relational Interval Tree* is that the relational DBMS features a procedural query language. This means that the DBMS has to support SQL with its procedural extension. This is the case if the DBMS supports SQL PL from IBM or PL/SQL from Oracle for example. There are some more procedural extensions from other companies as well which will also work fine. And this is the only requirement. If this requirement is fulfilled we can implement the *Relational Interval Tree* on top of any DBMS.

The system specific index parameters *root* and

minstep can be stored in a persistent data dictionary whereas the *leftNode* and *rightNode* parameters can be efficiently managed in a transient user session state because they are more likely to change. As shown in the previous section insertion and deletion both only need one SQL statement. This is because the computation and storage of the *fork node* and the update of the index parameters can be performed automatically by database triggers. So it's possible to let the index maintenance be managed by trigger mechanism. We only need to start the query processing manually by calling the procedure we just showed in a previous section and the rest will be done automatically.

Due to the fact that the maintenance of the index is triggered automatically the end user can use the *Relational Interval Tree* just as any other built-in index. If we would register a cost model at the optimizer even the generation of efficient execution plans for queries would be possible.

4 Experimental Evaluation

The authors of the paper have done some experiments to show the efficiency of the *RI - Tree* which I'll explain in this section and show what they were able to find out.

4.1 Experimental Setup

For the experiments they had integrated their *Relational Interval Tree* into an Oracle Server Release 8.1.5 which was using PL/SQL. As mentioned in the previous section a procedural query language as PL/SQL provides is the only requirement to the DBMS. The *RI - Tree* was integrated with the SQL statements we have seen. The software was running on a Pentium Pro/180 server having 128 MB main memory and an U-SCSI hard drive. They didn't change the default values what means that the block cache could handle 200 database blocks with a block size of 2 KB.

They did several tests for several distributions. They had two test scenarios with a uniform distribution of starting points (D_1 and D_2) and inserted new points uniformly into D_1 and exponentially into D_2 . Then they did the same again but started with a database which had the starting points distributed as a poisson function (D_3 and D_4) and so they inserted again uniformly into D_3 and exponentially

into D_4 .

They compared this test scenarios with the *Window – List*, *Tile Index* ($T – Index$) and the *Interval – Spatial Transformation* (IST) because these are the only interval access methods which are able to use the existing $B + – tree$ without any internal modifications or augmentations of the indexes.

(The *Window – List* technique is a static solution; the *Tile index* ties several datasets into one single layer; the *Interval – Spatial Transformation* maps a given time interval to a point in a two-dimensional space. You don’t need to understand this techniques in detail to be able to follow the coming comparison. I show you them here because this are the only techniques which could also be built-in directly and so come into question for a comparison.)

The *Window – List* used twice as many I/O operations as the dynamic $RI – tree$ and because the *Window – List* technique is a static technique they didn’t evaluate this technique in the following experiments because the following experiments needed a dynamic one.

They have reimplemented some code of the $T – index$ because the original $T – index$ was designed for two dimensional problems and they tested it only with one dimension and they were able to gain some performance compared to the original one.

With the IST they used the $D – order$ which is equivalent to a composite index on the attributes (*upper, lower*) and so queries on the $D – ordered$ intervals can also be expressed in only one SQL statement as shown below:

```
SELECT id FROM Intervals i
WHERE (i.upper >= :lower
      AND i.lower <= :upper);
```

The $RI – tree$ they have implemented as shown above and every test scenario contains intervals with length 0 (points). So the granularity of the respective data space is maximal and the *minstep* parameter will always reach its minimum value of 1 during index creation. The tree was always expanded to a height of 20 because the starting points were always distributed in the interval $[0, 2^{20} – 1]$.

4.2 Storage Occupation

The IST technique produces no redundancy and so the number of index entries is equal to the

number of indexed intervals. The $RI – tree$ needs two index entries for each interval because we had defined the index to contain the *lowerIndex* and the *upperIndex*. The $T – index$ has clearly lost this comparison because in their testcase it had a redundancy factor of 10.1 and so was no challenge for the other two. Such a huge factor could lead to serious problems in a large interval database.

4.3 Query Processing

Their first experiment compared the number of physical disk block accesses and the response time depending on the selectivity of the range queries for the three access methods mentioned before.

The $RI – tree$ outperforms the other two clearly. Already at a selectivity of only 0.5% the $T – index$ needs 10.8 times more block accesses, the IST 46.3 times more. At a selectivity of 3.0% the factor of the $T – index$ is already 22.8 and for the IST it’s still 13.6. So it’s clear that the $RI – tree$ has better performance concerning the disk access as well as for low query selectivity as for high query selectivity. The more or less fast response times of the $T – index$ and the IST comes from the good clustering of the bulk loaded indexes but the response time will get worse in a dynamic environment.

The second experiment was to show the scale up of the three techniques when the dataset grows from 1’000 to 1’000’000 stored intervals. They compared the average response time of 20 range queries and the average of disk access needed. Also here it becomes clear that the $RI – tree$ is better then the other solutions. The $T – index$ and IST both scale linearly whereas the $RI – tree$ only scales sublinear and so clearly can outperform the other two. The factor to compare the $T – index$ to the $RI – tree$ is 2.0 for 1’000 intervals and 42.0 for 1’000’000 intervals. For the disk access the factor grows from 2.0 to 4.9. The IST ’s factors are similar to the ones of the $T – index$.

The third experiment had to test the influence of the granularity of the data space on the query performance. This means they wanted to know how response time changes when the minimum length of the intervals increases. When the minimum length of the intervals increases the parameter *minstep* grows larger and fewer levels have to be searched through. We would expect that it takes less time. Surprisingly the response time is almost independent of the minimum length of the intervals and a difference can hardly be recognized.

The next test should show the influence of the average length of the stored intervals. Also here the *RI-tree* outperforms the other two competitors. The performance of the *RI-tree* becomes better when the mean length of the intervals grows.

In the last one they tested the influence of the location of the query range with respect to the data domain. This experiment shows that the *RI-tree* performs independent of the location whereas the other two show some differences. The *T-index* has some strange oscillation and the *IST*'s response time grows awfully further away the upper bound is from the query range.

5 Conclusions

The main thing to say is that the *Relational Interval Tree* can be easily integrated in any relational DBMS because it exploits the built-in indexes and needs no changes or additions to the internal layer.

The performance is as shown in the experiments superior to the other methods and also no space is wasted because the intervals are stored object-driven.

Thanks to the parameters *now* and *infinity* the *RI-tree* can also be extended without any problems.

6 References

[KPS 00]

Hans-Peter Kriegel, Marco Pötke, Thomas Seidl:
"Managing Intervals Efficiently in Object-Relational
Databases, Proceedings of the 26th International
Conference on Very Large Databases, Cairo, Egypt,
2000