Indexing Techniques For Time Series Data

The Oracle database offers powerful index structures such as B-Tree indexes which, when used appropriately, can significantly improve query performance. In this article I will explore the interval tree index, a well known structure for indexing time intervals, and demonstrate how a virtual interval tree can be built on top of a B-Tree index. The article will show the benefits in terms of performance of this technique and show how the drawbacks, such as added complexity, can be mitigated using Oracle features, for instance pipelined table functions and virtual columns.

James Checkley, Consulting Solutions Manager, TEOCO

Time Intervals

Time intervals present a challenge to the database administrator as they are described by two independent variables, a start or beginning time and an end time. Adding further complexity – predicates based on time intervals are often inequalities, and queries based around more than one inequality are difficult to optimise using a B-Tree.

This is difficult to visualise in abstract terms so for the purposes of this article I'm going to use a set of data describing planned roadworks, made available for public consumption by the DfT[1].

Our dataset contains one row for each planned roadwork job.

Each roadwork job is represented by a time interval, commencing at a certain point in time and finishing (hopefully!) at another pre-planned time. We can represent this data in a Gantt like format as shown in figure 1.

With this established, we can consider some typical queries that might be asked of this dataset:

1) How many road works start between the 12th October and the 28th October?
2) How many road works end between the 30th October and the 5th November?
3) What roadworks will be happening any time between 18:00 on the 23rd October and 22:00 on the 25th October?

The first two queries depend only on a single field of the roadworks data (the start and end respectively). They can be optimised trivially by, for example, creating a B-Tree index on the start time and the end time fields.

The third query is more difficult. Note that we are defining a window (a query

```
SQL> select * from temp_roadworks where rownum < 4;

REFERENCE ROAD DESCRIPTION                        START_TIME           END_TIME
---------- ---- ---------------------------------- -------------------- --------------------
   3012768 A2   East and Westbound btw Darenth 2014-10-18 20:00:00  2014-11-03 06:00:00
   3014310 A2   Eastbound at Lydden                2014-10-14 20:00:00  2014-10-15 06:00:00
   3018638 A2   Eastbound between Cobham RAB a 2014-10-13 20:00:00  2014-10-14 06:00:00
```

1. http://data.highways.gov.uk/ha-roadworks 2. http://web.cacs.louisiana.edu/~logan/521_f08/Doc/p832-allen.pdfw 3. http://en.wikipedia.org/wiki/Interval_tree
4. http://www.solidq.com/sqj/Pages/2011-September-Issue/A-Static-Relational-Interval-Tree.aspx 5. http //www.dbs.ifi.lmu.de/Publikationen/Papers/VLDB2000.pdf

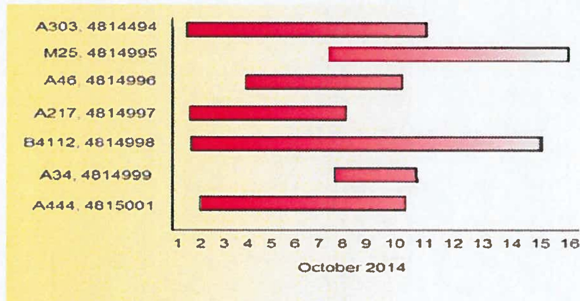

FIGURE 1: ROADWORK DATA

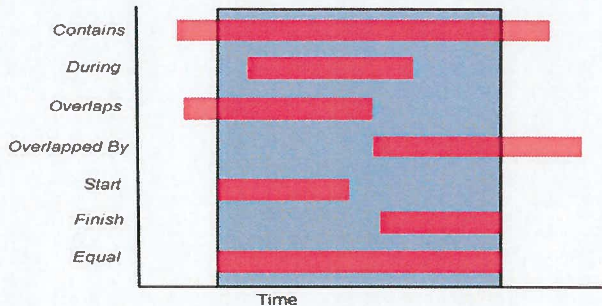

FIGURE 2: INTERVAL RELATIONS

window) and want to find rows in the table that have any interaction with the query window. We would be interested in any planned works that **contain, occur during, overlap, are overlapped by, start, finish** or are **equal to** our query window. These relations are some of the 13 general interval relations as defined by Allen[2] and can be shown graphically as per figure 2.

We can sum this up into two statements that must both be true in order for there to be an interaction between our query window (journey) and rows (planned works):

- **The works must start before the end of the journey**
- **The works must end after the beginning of the journey**

Using a B-Tree index, we can filter on one field or the other, but not both (even using a compound index – as a compound index does not work effectively on two inequalities).

Introduction to the Interval Tree

An interval tree is a data structure designed with intervals in mind[3]. I use the term data structure loosely here as we don't need to actually materialise an interval tree in memory or on disk in order to use it.

Like a B-Tree, an interval tree has a root node and branch nodes. It also has a number of parameters:

The **Root Node** is the top node in the tree. This node is at a fixed point in time that is usually in the centre of the total range of the intervals to be indexed.

The **Initial Step Size** is the distance in time between the root node and each of its immediate child nodes. This is determined by the total range of intervals to be indexed.

The **Minimum Step Size** is the smallest step size in the tree. It determines the maximum depth of the tree.

The tree begins at the root node, and each node in the tree except for those on the lowest level has two children. From the root, the two children are found by moving an amount specified by the Initial Step Size forwards and backwards from the root. The step size is then halved, and the next set of descendent nodes is found by moving half the initial step size forwards and backwards from each of the root node's descendents. This looks complicated when described in text; on a diagram it is much clearer. Figure 3 shows an example.

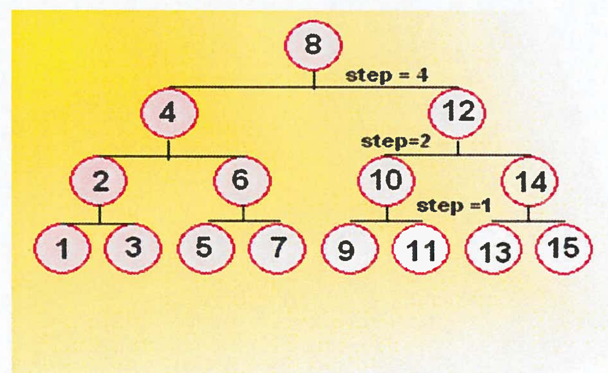

FIGURE 3: TREE STRUCTURE – NOTE THAT FOR SIMPLIFICATION I HAVE SUBSTITUTED REAL DATES WITH NUMBERS – IMAGINE THAT EACH NUMBER REPRESENTS A NUMBER OF DAYS FROM AN ARBITRARY EPOCH.

In order to actually make use of our tree, we need to associate each of the intervals that we want to index with a node in the tree. At this point I'm going to introduce a new term: The Fork Node. We define the fork node of an interval to be the highest node in the tree that is within that interval. We can define a general function for determining the fork node of an interval.

Start at the root node:
Is this node within the interval?
If yes, this node is the fork node
Is this node before (smaller than) the start of the interval?
If yes, follow the index down the right path and repeat with the next node
Is this node after (bigger than) the end of the interval?
If yes, follow the tree down the right path and repeat with next node

Once the fork node for an interval is known, it is possible to exploit some further properties of the interval tree in order to find intersections between the rows that have been indexed, and the query window.

>>

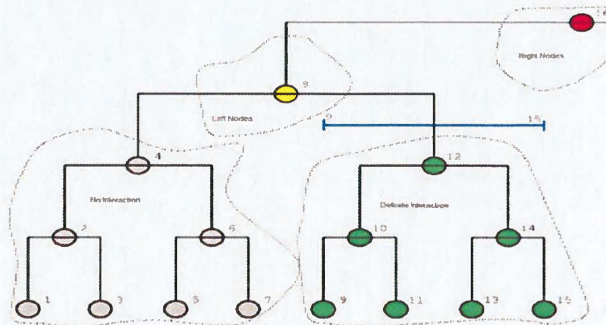

FIGURE 4: QUERY WINDOW PROPERTIES

Figure 4 shows a small subsection of an interval tree. An example query window is shown as a blue line running from position 9 to position 15 in the time series. Let's imagine that we want to find all the intervals in the table that have any type of interaction with our query window. To use the interval tree to do this we must define the following sets of index nodes based on the query window:

Inner nodes are the set of index nodes which are within the query window, shown in green on the diagram. **Left nodes** are found by walking up the tree from the fork node of the interval window to the root of the tree, only nodes which are on this walk and to the left of the interval window are in the left nodes set. Correspondingly, **right nodes** are also found by walking the tree from the fork node of the interval window, and identifying which nodes are both on this walk and to the right of the query window.

With these sets established, by inspection (and review of previous work on the subject), we can determine that the following rules apply.

Rule 1: Any interval which has a fork node that is a member of Inner Nodes definitely interacts with our query window.

Rule 2: Any interval that is a member of Left Nodes and ends after the start of the query window will definitely interact with the query window.

Rule 3: Any interval that is a member of Right nodes and starts before the query window will definitely interact with the query window.

Rule 4: Any intervals which do not meet the criteria of rules 1,2 or 3 will not interact with the query window.

Applying Interval Trees in PL/SQL

In PL/SQL, the pseudo code above describing the tree walk to find the fork node of an interval can be written as:

```
--Listing 1: Find fork node function
CREATE OR REPLACE FUNCTION
 temp_find_fork(root  number, init_step  number, interval_start
number
              ,interval_end number, min_step number)
 return number
 DETERMINISTIC
 is
 l_current_node number := root;
l_current_step number := init_step;
function is_node_in_interval(node number) return boolean
is
begin
  if interval_start <= node and interval_end >= node then
    return true;
  else return false;
  end if;
end is_node_in_interval;
begin
while true
loop
if is_node_in_interval(l_current_node) = true then
  return l_current_node;
elsif l_current_node < interval_start then
  l_current_node := l_current_node + l_current_step;
elsif l_current_node > interval_end then
  l_current_node := l_current_node - l_current_step;
end if;
if l_current_step < min_step then exit;
end if;
l_current_step := l_current_step / 2;
end loop;
return null;
end;
```

I will use the conventions of representing all datetimes as Unix 4 byte timestamps. The Unix Epoch (0) will be the root of the tree, and the initial step size will be set to the half the maximum value that can be stored in a 4 byte signed integer: 0173741824. This ensures that if an interval can be represented using Unix timestamps, it will be possible to index it in the tree.

The following PL/SQL function can be used to convert between an Oracle DATE type and the number of seconds since the Unix Epoch:

```
--Listing 2: Date to unix time function
CREATE OR REPLACE FUNCTION date_to_unix_time (p_date DATE)
return number
deterministic
As
Begin
Return (p_date - TO_DATE('1970-01-01', 'YYYY-MM-DD')) * 86400;
End ;
/
```

Armed with this function and these parameters, we can now try and calculate a fork node for every row in our sample data. In this case I'm going to add this as a virtual column:

```
--Listing 3, add virtual column
alter table temp_roadworks
add index_node
generated always as (
  temp_find_fork(0,1073741824,
    temp_date_to_unix_time(start_time),
    temp_date_to_unix_time(end_time)
   ,1))
```

I will use this virtual column to create two B-Tree indexes, which will be used to access the dataset based on the properties of the interval tree described above:

```
CREATE INDEX ROADWORKS_INTERVALS_START ON TEMP_ROADWORKS
("INDEX_NODE", "START_TIME")
CREATE INDEX ROADWORKS_INTERVALS_END ON TEMP_ROADWORKS("INDEX_
NODE", "END_TIME")
```

In order to identify the left node and right nodes set of a query window, we can use a pipeline function that will descend from the root of the interval tree down to a given node, visiting each node on the path:

```
--Listing 4: Descend to node pipelined function
CREATE OR REPLACE FUNCTION TEMP_DESCEND_TO_NODE  (root  number
               ,init_step   number, destination number
                 )
 return temp_node_list
 PIPELINED
 is
 l_current_node number := root;
 l_current_step number := init_step;
 begin
 while true
 loop
 pipe row (l_current_node);
 if l_current_node = destination then exit;
 elsif l_current_node < destination then
   l_current_node := l_current_node + l_current_step;
 elsif l_current_node > destination then
   l_current_node := l_current_node - l_current_step;
 end if;
 l_current_step := l_current_step / 2;
 end loop;
 end;
```

Finally, we can now write an interval tree query to answer the question:

```
 --Listing 5: Interval tree query to find any intersection
select count (distinct reference) from
( with left_nodes as
( select column_value node from
 table (temp_descend_to_node(0,1073741824, :window_start_unix))
where column_value <= :window_start_unix
),right_nodes as
( select column_value node from
 table (temp_descend_to_node(0,1073741824, :window_end_unix))
where column_value >= :window_end_unix
)
select /*+ ORDERED USE_NL(ln,tr) */ reference,road,start_
time,end_time,description from
 left_nodes ln ,temp_roadworks tr
where ln.node = tr.index_node
and tr.end_time > to_timestamp('2014-07-23 18','YYYY-MM-DD
HH24')
UNION ALL
select /*+ ORDERED USE_NL(ln,tr) */ reference,road,start_
time,end_time,description from
 right_nodes rn ,temp_roadworks tr
where rn.node = tr.index_node
and tr.start_time < to_timestamp('2014-07-23 22','YYYY-MM-DD
HH24')
UNION ALL
Select reference, road, start_time, end_time, description from
temp_roadworks where index_node between :window_start_unix and
:window_end_unix
);

COUNT(DISTINCTREFERENCE)
------------------------
                     703
```

Results & Conclusions

The result obtained using Listing 5 agrees with the result of a naive query which is answered using a full table scan. Using Tom Kyte's runstats utility we can obtain timing differences for the two queries.

Database state	Interval Tree Query	Naive Query	Interval tree ran in
Cold (first run)	70 hsecs	90 hsecs	77.78% of the time
Second run	1 hsecs	89 hsecs	1.12% of the time

We have demonstrated that it is possible to obtain significant performance improvements using interval trees; however this must be weighed against the extra complexity introduced. The interval index must be maintained, which will slow insert operations, and the dataset used here has been deliberately chosen with this article in mind and it is thus well suited to this type of index. It should also be noted that providing hints for the optimiser will limit its ability to adapt to different datasets and thus should generally be avoided.

Further Reading

Laurent Martin's work on the development of the binary arithmatic based Static interval tree was an inspiration for this article and I highly recommend reading his publications on the subject. Laurent has further optimised the structure using binary arithmatic to cut down on the tree walking steps[4]. The paper on relational interval trees by Hans-Peter Kriegel, Marco Pötke and Thomas Seidl is also recommmended reading[5]. ■

ABOUT THE AUTHOR

James Checkley
Consulting Solutions Manager, TEOCO

James is a manager in the services team at TEOCO, responsible for developing new techniques for optimising mobile phone networks. He has 10 years experience of using Oracle databases to process telecoms data on a Terabyte scale and his interests include APEX, PL/SQL and Spatial (Locator). James is a Chartered Engineer.

in uk.linkedin.com/pub/james-checkley/2/a9/674