[Zurück](javascript:history.back())

**Oracle Function Based Indexes**

**Introduction**

This is the first of many articles on new Oracle8i release 8.1 database features. I will be discussing in some depth many of the 150+ new features contained in the Oracle database over the next couple of months. In each of these articles we will explore some new feature found in the database and explain/provide:

**Function Based Indexes**

Oracle8i introduces a feature virtually every DBA and programmer will be using immediately -- the ability to index functions and use these indexes in query. In a nutshell, this capability allows you to have case insenstive searches or sorts, search on complex equations, and extend the SQL language efficiently by implementing your own functions and operators and then searching on them.

**Why to use this feature**

* It's easy and provides immediate value.
* It can be used to speed up existing applications without changing any of their logic or queries.
* It can be used to supply additional functionality to applications with very little cost.

So why is it easy and of immediate value? It's easy because it's just a CREATE INDEX statement. Consider the following example: I begin by creating a copy of the «scott/tiger» demo employee table. I then change the data in the employee name column to be in mixed case. I then create an index on the UPPER of the *ename* column -- effectively creating a case insensitive index:

|  |  |
| --- | --- |
|  | update emp set ename = initcap(ename); commit; create index emp\_upper\_idx on emp(**upper(ename)**); |

We now have an index on the «UPPER» of a column. Any application that already issues 'case insensitive» queries of the form:

|  |  |
| --- | --- |
|  | set autotrace on explain select ename, empno, sal from emp where **upper(ename)** = 'KING';  ENAME           EMPNO        SAL ---------- ---------- ---------- King             7839       5000  Execution Plan ----------------------------------------------------------    0      SELECT STATEMENT Optimizer=ALL\_ROWS (Cost=1 Card=1 Bytes=40)    1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=1 Card=1 Bytes=40)    2    1     INDEX (RANGE SCAN) OF 'EMP\_UPPER\_IDX' (NON-UNIQUE) (Cost=1 Card=1) |

will transparently make use of this index -- gaining the performance boost an index can deliver. Before this feature was available, every row in the EMP table would have been scanned, upper-cased and compared. In constrast, with the index on upper(ename, the query takes the constant KING to the index, range scans a little data and accesses the table by ROWID to get the data. This is very fast.

This performance boost is most visible when indexing user written functions on columns. Oracle7 Release 7.1 added the ability to use user written functions in SQL so that you could:

|  |  |
| --- | --- |
|  | select **my\_function(ename)**  from emp  where **some\_other\_function(empno)** > 10  / |

This was great because you could now effectively extend the SQL language to include application specific functions. Unfortunately however, the performance of the above query was a bit disappointing. Say the EMP table had 1,000 rows in it -- the function «some\_other\_function» **would be executed 1,000** times during the query, once per row. Additionally, assume the function took 1/100 of a second to execute. This relatively simple query now takes at least 10 seconds.

**How to enable Function Based Indexes**

The following is a list of what needs to be done to use function based indexes:

* You must have the system privelege **query rewrite** to create function based indexes on tables in your own schema.
* You must have the system privelege **global query rewrite** to create function based indexes on tables in other schemas
* For the optimizer to use function based indexes, the following session or system variables must be set:

QUERY\_REWRITE\_ENABLED=TRUE  
QUERY\_REWRITE\_INTEGRITY=TRUSTED

You may enable these at either the session level with ALTER SESSION or at the system level via ALTER SYSTEM or by setting them in the init.ora parameter file. The meaning of query\_rewrite\_enabled is to allow the optimizer to rewrite the query allowing it to use the function based index. The meaning of is to tell the optimizer to «trust» that the code marked deterministic by the programmer is in fact deterministic. If the code is in fact not deterministic (that is, it returns different output given the same inputs), the resulting rows from the index may be incorrect.

* Use the Cost Based Optimizer. Function based indexes are only visible to the Cost Based Optimizer and will not be used by the Rule Based Optimizer ever.
* Use substr() to constrain return values from user written functions that return VARCHAR2 or RAW types. Optionally hide the substr in a view (recommended).

Once the above list has been satisfied, it is as easy as «CREATE INDEX» from there on in. The optimizer will find and use your indexes at runtime for you.

**Real Example**

Here is a real example - a modified «soundex» routine in PL/SQL:

|  |  |
| --- | --- |
|  | alter session set query\_rewrite\_enabled = true; alter session set query\_rewrite\_integrity = trusted;  create or replace package stats as      cnt number default 0; end; /  create or replace function my\_soundex(p\_string in varchar2) return varchar2 **deterministic** as      l\_return\_string   varchar2(6) default substr(p\_string, 1, 1);      l\_char            varchar2(1);      l\_last\_digit      number default 0;       type vcArray is table of varchar2(10) index by binary\_integer;      l\_code\_table      vcArray;  begin      stats.cnt := stats.cnt+1;       l\_code\_table(1) := 'BPFV';      l\_code\_table(2) := 'CSKGJQXZ';      l\_code\_table(3) := 'DT';      l\_code\_table(4) := 'L';      l\_code\_table(5) := 'MN';      l\_code\_table(6) := 'R';        for i in 1 .. length(p\_string)      loop          exit when (length(l\_return\_string) = 6);          l\_char := substr(p\_string, i, 1);           for j in 1 .. l\_code\_table.count          loop          if (instr(l\_code\_table(j), l\_char) > 0 AND j <> l\_last\_digit)          then              l\_return\_string := l\_return\_string || to\_char(j,'fm9');              l\_last\_digit := j;          end if;          end loop;      end loop;       return rpad(l\_return\_string, 6, '0'); end; / |

Notice in this function, the usage of the keyword **«deterministic»**. Deterministic declares that the above function -- when given the same inputs -- will always return the exact same output. **This keyword is needed in order to create an index on a user written function**. You must tell Oracle that the function is «deterministic» and will return a consistent result given the same inputs. This implies for example that you cannot index using the package «dbms\_rand», the random number generator. Its results are not deterministic, given the same inputs you'll get random output. The builtin sql function UPPER on the other hand is deterministic so you can create an index on the UPPER of a column.

Now that we have the function «My\_Soundex()», lets see how it performs without an index...

|  |  |
| --- | --- |
|  | create table test\_soundex(name varchar2(30));  insert into test\_soundex        select object\_name          from all\_objects         where rownum <= 5000;  exec stats.cnt := 0;  set timing on set autotrace on explain select name   from test\_soundex A  **where my\_soundex(name) = my\_soundex('FILE$')** /  NAME ------------------------------ FILE$  **Elapsed: 00:00:03.00**  Execution Plan ----------------------------------------------------------    0      SELECT STATEMENT Optimizer=ALL\_ROWS (Cost=1 Card=1 Bytes=34)    1    0   TABLE ACCESS (FULL) OF 'TEST\_SOUNDEX' (Cost=1 Card=1 Bytes=34)  set autotrace off set timing off set serveroutput on exec dbms\_output.put\_line(stats.cnt) 10000 |

So, we can see this query took over 3 seconds to execute and had to do a full scan on the table. The function *my\_soundex* was invoked 10,000 times (according to our counter), twice for each row. Lets see how indexing the function can be used to speed things up.

The first thing we will do is create the index as follows:

|  |  |
| --- | --- |
|  | create index test\_soundex\_idx on test\_soundex(**substr(my\_soundex(name),1,6)**); |

Now, the interesting thing to note in this create index command is the use of the substr function. This is because we are indexing a function that returns a string. If we were indexing a function that returned a number or date this substr would not be necessary. The reason we must substring the user written function that returns a string is that they return VARCHAR2(4000) types. That is too big to be indexed -- index entries must fit within 1/3 the size of the block. If we tried we would recieve (in a database with a 2k blocksize) the following:

|  |  |
| --- | --- |
|  | create index test\_soundex\_idx on test\_soundex(my\_soundex(name),1,6); create index test\_soundex\_idx on test\_soundex( my\_soundex(name),1,6 )                                                                   \* ERROR at line 1: ORA-01450: maximum key length (3118) exceeded |

In databases with larger block sizes, the number 3118 would vary but, unless you are using a 16k or larger blocksize, you will not be able to index a VARCHAR2(4000).

So, in order to index a user written function that returns a string, we must constrain the return type in the create index statement. In the above, knowing that *my\_soundex* returns at most 6 characters, we are substring the first six characters.

We are now ready to test the performance of the table with the index on it. We would like to monitoring the effect of the index on INSERTS as well as the speedup for SELECTS to see the effect on each. In the un-indexed test case, our queries took over 3 seconds.

|  |  |
| --- | --- |
|  | delete from test\_soundex; exec stats.cnt := 0;  insert into test\_soundex select object\_name  from all\_objects where rownum <= 5000;  exec dbms\_output.put\_line(stats.cnt) 5000  exec stats.cnt := 0;  set autotrace on explain set timing on  select name  from test\_soundex  B where **substr(my\_soundex(name),1,6) = my\_soundex('FILE$')** /  NAME ------------------------------ FILE$  Elapsed: 00:00:00.05  Execution Plan ----------------------------------------------------------    0      SELECT STATEMENT Optimizer=ALL\_ROWS (Cost=1 Card=1 Bytes=34)    1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST\_SOUNDEX' (Cost=1 Card=1 Bytes=34)    2    1     INDEX (RANGE SCAN) OF 'TEST\_SOUNDEX\_IDX' (NON-UNIQUE) (Cost=1 Card=1) |

The important things to note here are that:

* The insert of 5,000 records took longer. Indexing a user written function will necessarily affect the performance of inserts and some updates. Since most applications insert and update singleton entries.
* While the insert ran slower, the query ran much **faster**. Also, as the size of our table grows, the full scan query will take longer and longer to execute. The index based query will always execute with the near same performance characteristics as the table gets larger.
* We had to use «substr()» in our query. This is not as nice as just coding «where my\_soundex(name) = my\_soundex( 'FILE$' )» but we can easily get around that we will show below:

We would now like to see how to make it so the query does not have use the *substr* function call. The use of the *substr* call could be error prone -- your end users have to know to *substr* from 1 for 6 characters. If they used a different size, the index would not be used. Also, you want to control in the server the number of bytes to index. This will allow you do reimplement the my\_soundex function later with 7 bytes instead of 6 if you want to. We can do this, hide the substr, **with a view** quite easily as follows:

|  |  |
| --- | --- |
|  | create or replace view test\_soundex\_v   as   select name, substr(my\_soundex(name),1,6) name\_soundex  from test\_soundex /  exec stats.cnt := 0;  select name  from test\_soundex\_v B where name\_soundex = my\_soundex('FILE$') /  NAME ------------------------------ FILE$  Elapsed: 00:00:00.04  Execution Plan ----------------------------------------------------------    0      SELECT STATEMENT Optimizer=ALL\_ROWS (Cost=1 Card=1 Bytes=34)    1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST\_SOUNDEX' (Cost=1 Card=1 Bytes=34)    2    1     INDEX (RANGE SCAN) OF 'TEST\_SOUNDEX\_IDX' (NON-UNIQUE) (Cost=1 Card=1)  exec dbms\_output.put\_line( stats.cnt ) 2 |

So, what we have done here is to hide the substr( f(x), 1, 6 ) in the view itself. The optimizer still recognizes that this virtual column is in fact the indexed column and does the «right thing». We see the same performance improvement and the same query plan. Using this view is as good as using the base table, better even since it hides the complexity and allows you to change the size of the substr later.

**Conclusion**

Here are some of the pros of this feature

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * Its easy to use/implement and provides immediate value * It can be used to speed up existing applications without changing any of their logic or queries. Many orders of magnitude query improvement may be observed. * It can be used to precompute complex values without using a trigger * Can be created either as B\*Tree or bitmap index * Index can be built on an arithmetic expression or expression containing PL/SQL, package functions, C callout or SQL built-in functions * Optimizer can estimate selectivity more accurately if the expressions are materialized in a function-based index. Range scan can be used for queries with expression in where clause and has index build on the expression used. * Provides efficient linguistic collation to use NLS sort index * Indexes can be created on object columns and REF columns by using methods defined for the object. | * You cannot direct path load the table with a function based index if that function was user written and requires the SQL ENGIRE. That means you cannot direct path load into a table indexed using my\_soundex(x), but you could if it had indexed upper(x). * It will affect the performance of inserts and updates. (Remember, you insert a row once, you query it thousands of times.) * If the function you are indexing is a user written function and it returns a string, you may have to expose a view for the end users to use. |

In general, the pros heavily out weigh any of the cons in this case. The inability to direct path load with a pl/sql based index can easily be overcome by indexing after the load with the parallel query option. The performance of inserts is only marginally affected, most applications won't even notice the effect.