Oracle bitmap Indexes and their use in pattern matching.

Author : Dominic Giles

Date : 8 July 2005

Introduction

It is a common requirement to search for text on the web or in files or databases and many solutions exist to solve this problem. Another similar requirement is to be able to search relational database tables for pattern matches on relatively small strings i.e. emails, product descriptions.

Searches of this class are typically characterised by relatively low numbers of rows, usually in the thousands and in exceptional circumstances millions. Whilst the number of rows being searched against remains low and the search criteria are exact matches or right truncated wildcards (i.e. “dominc%”) the solution used is typically memory resident scans or the use of various types of standard indexes i.e. b-tree . The problem becomes harder to solve efficiently when the quantity of rows that need to be searched increase to many millions or queries use use left or doubly truncated wildcards (i.e. “%omini%”).

Simple Pattern Matches (right truncated wildcards)

Examples of simple searches (using Oracle syntax)

where product = 'super widescreen TV'	Find all rows that exactly match their column “product” with the text “super widescreen TV”
where description like '17inch%'	Find all rows whose columns “description” starts with “17inch”

The Oracle database provides a number of solutions to provide fast responses to queries of this class.

Full Tables Scans	In some instances where the amount of data being queried is small (i.e. can be cached in memory) , this approach may provide a simple efficient means of performing both simple and complex searches.	Pros: Simple, No additional resource required. Cons : Performance will quickly degrade as volumes increase
B-tree Indexes	Provides fast access for exact and simple pattern matches. This approach scales well from small to large data sets.	Pros: Simple and well understood, scales well. Copes well with dynamic data sets. Cons: Requires additional storage and system resource to create index.
Bitmap Indexes	Provides fast access for exact and simple pattern matches. This approach scales well from small to large data sets. Also provides very efficient AND/OR and count operations.	Pros: typically small in comparison to B-tree indexes. Cons: Requires additional storage and system resource to create index. Not designed to handle many small updates

Full Tables Scans	In some instances where the amount of data being queried is small (i.e. can be cached in memory) , this approach may provide a simple efficient means of performing both simple and complex searches.	Pros: Simple, No additional resource required. Cons : Performance will quickly degrade as volumes increase
		efficiently.

Complex Pattern Matches (left or doubly truncated wildcards)

Examples of complex searches (using Oracle syntax)

where product like '%widescreen%'	Find all rows that have "widescreen" some where within the text of the product column
where regexp_like(description, '[[:digit:]]+ inch')	Find all rows that have digits (i.e 1234567890) followed by the word "inch" within the text of the description column
where email like '%dom99%@dominicgiles.com'	Find all rows that have "dom99" followed by "dominicgiles.com" somewhere within the text of the email column

Standard indexing approaches don't support complex pattern matching . Attempts to perform complex pattern searches on tables with only b-tree or bitmap indexes on the target column will result in full table scans. Oracle does however provide a number of solutions that can assist in performing complex pattern matches.

Oracle Text	Powerful text indexing engine supporting both complex token and pattern matching operations. (pattern matching optimisations result in bigger text indexes and longer times to create them)	Pros: Powerful search capabilities, works well on text with strong delimiters and stop words Cons : Large index sizes when compared to the small columns being searched. Long indexing times.
Functional Indexes	Functional bit map indexes enable the sub strings within a column to be indexed. It is then possible to build a query that makes use of the indexes (This approach is discussed later)	Pros: Comparatively small index sizes. Fast pattern match count (if bitmap index used). Works well on text with no delimiters or stop words Cons: Comparatively complex configuration. Only valid for small string lengths.

Whilst neither of these approaches supports the level of sophisticated pattern matching that can be achieved using regular expressions both approaches can be used as primary filters for queries that then perform exact regular expression matching using Oracle functions such as "regexp_like".

Oracle Text

Oracle Text is a technology which supports text query and document classification applications. Oracle Text provides indexing, word and theme searching, and viewing capabilities for text.

A text query application enables users to search document collections such as Web sites, digital libraries, or document warehouses. Searching is enabled by first indexing the document collection. The collection is typically static with no significant change in content after the initial indexing run. Documents can be of any size and of different formats such as HTML, PDF, or Microsoft Word. These documents are stored in a document table.

Queries usually consist of words or phrases. Application users can specify logical combinations of words and phrases using operators such as OR and AND. Other query operations such as stemming, proximity searching, and wildcarding can be used to improve the search results.

Oracle text typically consists of a ”Context” index on a document table. Users then perform queries using this index via the “Contains” operator in the where clause of a select statement.

To improve the performance of wild card queries it is recommended that users specify the preference for prefix and substring indexing.

Whilst Oracle Text excels on most forms of text indexing it typically depends on strong delimiters. If the text has none and pure pattern matching is being performed then the size of the indexes and time taken to find matches can be punitive .

Functional Indexes

It is possible if the column being searched is relatively small, typically less than 255 characters, to create functional indexes that index all of the possible sub strings. By creating a query that breaks the pattern being searched for into substrings the optimiser will use the functional indexes to provide a efficient access path to the rows that meet the patterns criteria. The use of bitmap indexes accelerates both the AND and OR operation used in these queries and provides an optimisation for counts.

The advantage of this approach over text is that it doesn't depend on strong delimiters and as a result the size of the indexes and query performance is fairly consistent regardless of the text being indexed.

Example :

The output is truncated for brevity

```
SQL > desc directory
Name              Null?       Type
---------------------------------------------------------------
ID                            NUMBER(8)
FIRSTNAME                     VARCHAR2(255)
SURNAME                       VARCHAR2(255)
EMAIL                         VARCHAR2(255)

SQL> create bitmap index dir_substr_1 on dir(substr(email,1,2));
create bitmap index dir_substr_2 on dir(substr(email,2,2))
create bitmap index dir_substr_3 on dir(substr(email,3,2))
create bitmap index dir_substr_4 on dir(substr(email,4,2))
...
create bitmap index dir_substr_49 on dir(substr(email,49,2))
SQL>
Index created
Index created
Index created
Index created
...
Index created
SQL> select email /* search for dominic occurring anywhere in the email column
*/
from dir
where (substr(email, 1,2) = 'do' and
            substr(email, 3,2) = 'mi' and
            substr(email, 5,2) = 'ni' and
            substr(email, 6,2) = 'ic')
      or (substr(email, 2,2) = 'do' and
            substr(email, 4,2) = 'mi' and
            substr(email, 6,2) = 'ni' and
            substr(email, 7,2) = 'ic')
      or (substr(email, 3,2) = 'do' and
            substr(email, 5,2) = 'mi' and
            substr(email, 7,2) = 'ni' and
            substr(email, 8,2) = 'ic')
...
      or (substr(email, 44,2) = 'do' and
            substr(email, 46,2) = 'mi' and
            substr(email, 48,2) = 'ni' and
            substr(email, 49,2) = 'ic');
EMAIL
---------------------------------------------------------------
```

The execution of the query results in a explain plan

```
SELECT STATEMENT
ALL_ROWS
      SORT(AGGREGATE)
            BITMAP CONVERSION(TO ROWIDS)
                  BITMAP OR
                        BITMAP AND
                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_1

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_3

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_5

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_6

                        BITMAP AND
                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_2

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_4

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_6

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_7

                        BITMAP AND
                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_3

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_5

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_7

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_8
                        . . . . .
                        BITMAP AND
                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_44

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_46

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_48

                        BITMAP INDEX(SINGLE VALUE) DIR.DIR_SUBSTR_49
```

Clearly there is a issue with a issue with the complexity and size of the select statement required to query the table. The complexity of creating the required SQL can be overcome by using a in-line table function such as

```
create or replace function getMatches (myTable varchar2, myColumn varchar2,
token varchar2, indexwidth number default 2) return matchtypeset
parallel_enable
pipelined
is
  pragma          autonomous_transaction;
  type cursortype is ref cursor;
  matchvalue      varchar2(2000);
  refcursor       cursortype;
  outrec          matchtype := matchtype(null);
  sqltext         varchar2(32000) := 'select /*+ index_combine(t) */ '||
myColumn||' from '||myTable||' t where (';
  k               pls_integer;

begin
  for j in 0..50-length(token)
  loop
    if j > 0 then
      sqltext := sqltext || ') or (';
    end if;
    for i in 1..ceil(length(token)/indexwidth)
    loop
      if i > 1 then
        sqltext := sqltext || ' and ';
      end if;
      if indexwidth * i > length(token) then
        k := length(token) - indexwidth + 1;
      else
        k := indexwidth*(i -1) + 1;
      end if;
      sqltext := sqltext || ' substr(t.'||myColumn||', '|| (k+j) || ','||
indexwidth||') = ''' || substr(token,k,indexwidth)||'''';
    end loop;
  end loop;
  sqltext := sqltext || ')';
  open refcursor for sqltext;
  loop
    fetch refcursor into matchvalue;
    exit when refcursor%notfound;
    outrec.matchvalue := matchvalue;
    pipe row(outrec);
  end loop;
  return;
end;
```

This simplifies the SQL required to perform simple pattern matches to

```
DIR@DOM10 > select matchvalue from table(getmatches('dir', 'email', 'steve'))

MATCHVALUE
--------------------------------------------------------------------------------
steve.lepez@qqz.tv
cathryn.stevens@lzx.name
eli.steven@exj.tv
steve.garratt@mhq.co
steven.hulse@wks.name
steven.mondor@wta.tv
steve.kornreich@tsb.co
steven.claypole@ife.com
steve.saint@pgg.co

9 rows selected.
```

The query can then make use of functions such as "regexp_like" to implement more sophisticated pattern matching operations. For example

```
DIR@DOM10 > ;
  1  select matchvalue
  2  from table(getmatches('dir', 'email', 'steve'))
  3  where regexp_like(matchvalue, 'steve[[:digit:]]+')
  4*
DIR@DOM10 > /

MATCHVALUE
--------------------------------------------------------------------------------
steve3xc@irs.com
steve5I3@ufm.co
steve3ph@mmkp.org
steve2bf@gza.co
steve4nC@sff.tv
steve6E2@uwlq.gov
steve1K0@lldze.name
steve1EX@wmlo.gov
steve7nw@ffx.name
```

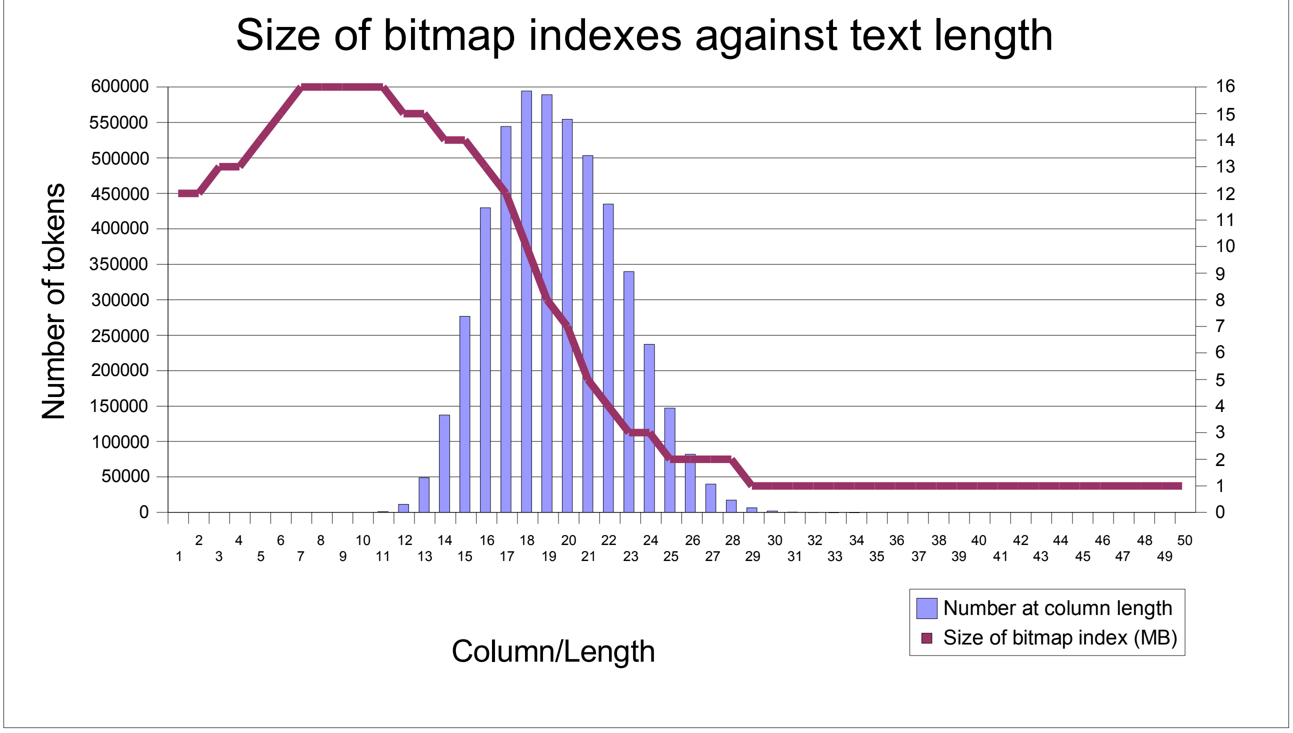
A similar function can be used to take advantage of Oracle's bitmap index optimisations for count operations.

```
1* select getmatchcount('dir', 'email', 'steve') as matchcount from dual
DIR@DOM10 > /


MATCHCOUNT
----------
      3056
```

Functional Index sizes

The total size required to index text using bitmap indexes is a function of the maximum length of data being indexed, its average length and the number of unique sub-strings . For example to index a column with a maximum length of 50 characters it is necessary to create 49 bitmap indexes. However the size of each of these bitmap indexes will vary in size depended on the average length of data and how many unique combinations of sub-strings exist along the length of the string. The following graph illustrates this.



The graph shows that the typical length of the text being indexed is 20 characters, its minimum is 12 and it maximum is 30. The bitmap indexes increase in size as the number of unique combinations of 2 letter sub-strings increase (English words typically start with a limited number of letters) and decrease in size when the combination of sub-strings diminishes. The advantage of bitmap indexes and their use in functional indexes are that when no sub-string combinations occur the index compresses very efficiently.

As with all all bitmap indexes the lower the cardinality the smaller the bit map indexes tend to be. This means that for data with repetitive sequences this approach results in smaller indexes which means that more of the index tends to be cached.

Test results for text lookup using Oracle Functional Bitmap Indexes

The following sections describe the performance profile of Oracle Text compared to Oracle functional bitmap Indexes in three different scenarios.

- Directory : Indexing 5 million email strings
- Bioinformatics : Indexing 5 million repetitive strings (using AGCT).
- Catalogue : Indexing 5 million short product descriptions.

It is worth pointing out that Oracle Text is not necessarily suited to the first two scenario's but it does provide a useful comparison.

The tests were carried out on the following hardware

Component	*Description*
Hardware	HP DL380, Xeon 3.4GHz, 4GB Memory, MSA 500 Storage Array (10 x 72GB disks).
Operating System	Redhat Enterprise Linux 4.0
Database	Oracle Enterprise 10.1.0.4

Index Sizes and Creation Times

The following tables describes the sizes of both the functional indexes and the text indexes including their time to create

Scenario	*Functional Bitmap*	
	Size (Total)	*Creation Time*
Directory	280MB	00:04:03.00
Bio-Informatics	337MB	00:05:22.00
Catalogue	317MB	00:03:51.00

Effect of changing substring length

The size of the substring used in the functional bitmap index is determined by the smallest string that needs to be searched. For example if you need to be able to efficiently count the number of times “ae” occurs in your data then the substring should be of length 2. Ideally the substring size should be one less that the minimum search screen that needs to be found, this results in a more efficient query plan. The down side of large substrings is that it increases the size of the resulting bitmap index. A substring index of 3 or 4 seems to be a reasonable comprise for flexibility and performance.

The following table describes the size of indexes as the size of the substring is increased in the functional bitmap index.

Scenario	*Total Bitmap Index Size By Substring Length*			
	2	3	4	5
Directory	280MB	365MB	519MB	837MB
Bio-Informatics	337MB	465MB	602MB	770MB

Scenario	*Total Bitmap Index Size By Substring Length*			
	2	3	4	5
Catalogue	317MB	385MB	413MB	468MB

Query Performance

The following shows how the response time varies for different functional bitmap indexes as the search string decreases in size for the Bioinformatics schema.

NOTE : these tests are performed on a fully memory resident database.

Time taken to perform full table doubly truncated wild card search = 9.6 secs.

Search String	*Response Time*			
	2	3	4	5
CATGCATCGATCGATCGTAGTTGGTAGGA CCGTCCCCTTATCTTAAA	00:00:00.69	00:00:00.13	00:00:00.03	00:00:00.03
CATGCATCGATCGATCGTAGTTGGTAGGA CCGTCCCCTT	00:00:01.27	00:00:00.24	00:00:00.06	00:00:00.06
CATGCATCGATCGATCGTAGTTGGTAGGA CC	00:00:03.72	00:00:00.39	00:00:00.10	00:00:00.08
CATGCATCGATCGATCGTAGTTGG	00:00:03.63	00:00:00.53	00:00:00.12	00:00:00.09
CATGCATCGATCGATCG	00:00:02.97	00:00:00.61	00:00:00.14	00:00:00.11
CATGCATCGAT	00:00:02.83	00:00:00.56	00:00:00.14	00:00:00.11
CATGCA	00:00:02.86	00:00:01.42	00:00:00.17	00:00:00.17
CAT	00:00:01.22	00:00:02.30	N/A	N/A

The following shows how the response time varies for different functional bitmap indexes as the search string decreases in size for the Directory schema

Time taken to perform full table doubly truncated wild card search = 4.3 secs.

Search String	*Response Time*			
	2	3	4	5
dominic106@drfyx.tv	00:00:00.07	00:00:00.05	00:00:00.01	00:00:00.01
dominic106@drfyx.	00:00:00.04	00:00:00.04	00:00:00.01	00:00:00.01
dominic106@drfy	00:00:00.04	00:00:00.03	00:00:00.01	00:00:00.01
dominic106@d	00:00:00.03	00:00:00.03	00:00:00.01	00:00:00.01
dominic10	00:00:00.02	00:00:00.04	00:00:00.01	00:00:00.01
domini	00:00:00.04	00:00:00.07	00:00:00.21	00:00:00.04
domi	00:00:00.04	00:00:00.17	00:00:00.31	N/A
dom	00:00:00.31	00:00:00.22	N/A	N/A

The following shows how the response time varies for different functional bitmap indexes as the search string decreases in size for the Catalogue schema

Time taken to perform full table doubly truncated wild card search = 6.5 secs.

Search String	***Response Time***			
	2	3	4	5
self-cleaning performance medium	00:00:00.15	00:00:00.02	00:00:00.01	00:00:00.02
self-cleaning performance med	00:00:00.19	00:00:00.02	00:00:00.01	00:00:00.02
self-cleaning performance	00:00:00.35	00:00:00.09	00:00:00.02	00:00:00.01
self-cleaning perform	00:00:00.24	00:00:00.02	00:00:00.01	00:00:00.01
self-cleaning pe	00:00:00.25	00:00:00.02	00:00:00.01	00:00:00.02
self-cleani	00:00:00.10	00:00:00.06	00:00:00.02	00:00:00.02
self-cl	00:00:00.05	00:00:00.01	00:00:00.01	00:00:00.02
sel	00:00:00.08	00:00:00.04	N/A	N/A

Further Work

Using the routines described here it is only possible to index data of a limited size. However it would be relatively simple to develop a loader to chunk longer text into rows and index this. The author will update this paper with the required code at a later date.

Conclusions

The use of functional bitmap indexes offers a flexible and efficient means of indexing data for simple pattern matching on data with little or no delimiters.