Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2019, 6(11):283-289

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Usage of Indexes in BW(Oracle) Business Information Warehouse

Naveen Muppa

10494 Red Stone Dr Collierville, Tennessee

Abstract In Oracle databases, including those used in Business Warehouse (BW) systems, indexes play a crucial role in optimizing query performance.

There are 2 ways to access data from a database. The first is of course without using any keys and indices. So for any query you give, the database SEQUENTIALLY retrieves all the records and checks for the match. This is probably good if you need most of the records if not all, say 85%. The second approach is to use keys. Keyed access provides direct addressing of records. A unique number or character(s) is used to locate and access records. In this case, when specified records are required (say, record 120, 130, 200 and 500), indexing is much more efficient than reading all the records in between.

Keywords Business Warehouse (BW) systems

1. Introduction

This document looks at the different kinds of indexing concepts and in particular the type of indexes used in SAP BW namely the Bitmap and the B-Tree (Binary Tree) indexes, their structures, the way they are created, their algorithms etc. SAP BW uses a combination these indexes to optimize the performance that is expected out of a ERP Datawarehouse. As the business volumes grow, so will the amount of data processed and stored in data warehouses. Companies will be more demanding in their need to access critical information. The success of a data warehouse depends a lot upon the speed with which data can be analyzed and thus indexes are key to the functioning of any data warehouse including SAP BW.

2. Indexes

An index can be composed of a single column for a table, such as the SSN (social security number) of a student at a university, or an index may also be comprised of more than one column for a table. This is termed a concatenated (or composite) index. An example of this could be a combination of Album Name and Track Number in a songs database. Each song is uniquely identified by an Album and Track.

The maximum number of columns for a concatenated index is 32; but the combined size of the columns cannot exceed about one-half of a data block size.

Different Types of Indexes are.

A unique index allows no two rows to have the same index entry. An example would be an index on student SSN.

A non-unique index allows more than one row to have the same index entry (this is also called a secondary key index or just secondary index). An example would be an index on U.S.

Mail zip codes. A database table can have only one unique index but multiple secondary indexes.

A function-based index is created when using functions or expressions that involve one or more columns in the table that is being indexed. A function-based index pre-computes the value of the function or expression and stores it in the index. Function-based indexes can be created as either a B-tree or a Bitmap index.

A partitioned index allows an index to be spread across several tablespaces - the index would have more than one segment and typically access a table that is also partitioned to improve scalability of a system. This type of index decreases contention for index lookup and increases manageability.
For the rest of the document, we will concentrate on function-based indexes (B-Tree and Bitmap) as they are of relevance to data warehousing and SAP BW.

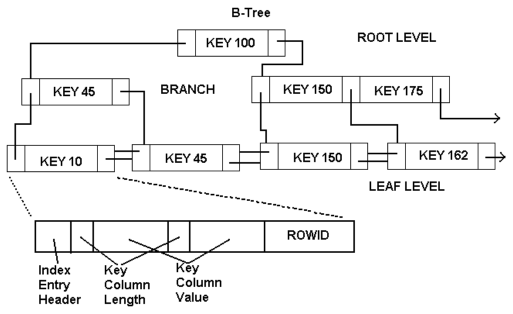
3. BI-TREE Indexes

The oldest and most popular type of Oracle indexing is the B-tree index, which excels at servicing simple queries. The b-tree index was introduced in the earliest releases of Oracle and remains widely used. B-tree indexes are used to avoid large sorting operations. For example, a SQL query requiring 10,000 rows to be presented in sorted order will often use a b-tree index to avoid the very large sort required to deliver the data to the end user.

Oracle offers several options when creating an index using the default b-tree structure. It allows you to index on multiple columns (concatenated indexes) to improve access speeds. Also, it allows for individual columns to be sorted in different orders. For example, we could create a b-tree index on a column called last name in ascending order and have a second column within the index that displays the salary column in descending order.

Structure of a B-Tree Index

A B-tree index usually stores a list of primary key values and a list of associated ROWID values that point to the row location of the record with a given primary key value. In this figure the ROWID values are represented by the pointers.



The top level of the index is called the Root. The Root points to entries at lower levels of the index - these lower levels are termed Branch Blocks.
A node in the index may store multiple (more than one) key values - in fact, almost all B-Trees have nodes with multiple values - the tree structure grows upward, and nodes split when they reach a specified size.
At the Leaf level the index entries point to rows in the table. At the Leaf level the index entries are linked in a bi-directional chain that allows scanning rows in both ascending and descending order of key values - this supports sequential processing as well as direct access processing.
In a non-partitioned table, Key values are repeated if multiple rows have the same key value – this would be a non-unique index (unless the index is compressed). Index entries are not made for rows that have all of the key column's NULL.

Format of the leaf Index

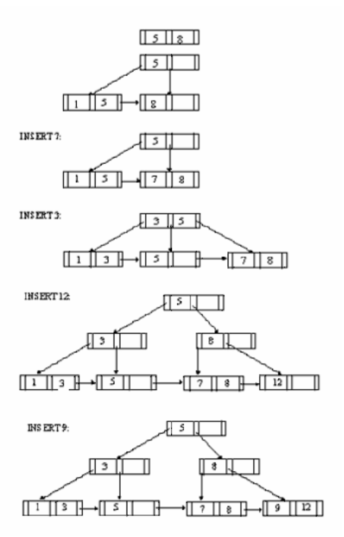
The index entry at the Leaf level is made up of three components.
- **Entry Header -** stores number of columns in the index and locking information about the row to which the index points.
- **Key Column Length -**Value Pairs - defines the size of the column in the key followed by the actual column value. These pairs are repeated for each column in a composite index.

- **ROWID -** this is the ROWID of a row in a table that contains the key value associated with this index entry.

Algorithm for creating a B-Tree Index:

1. First locate node (say N) at the lowest level of the index set.
   a. If the node contains a free space, insert value into the free space.
2. Otherwise, if node is full,
   a. Split the node N into 2 nodes N1 (left) and N2 (right).
      i. Place the smaller values in the left node N1 (Smallest value in the leftmost position).
      ii. Place the rightmost values in the right node N2.
   b. Create new parent node (if it doesn't already exist)
      i. Promote rightmost value from the left node else the middle value from the new value set (say W) to parent node.
3. An attempt is now made to insert W into parent node, and the process is repeated.

Taking an example,
Assume an insertion sequence of 8, 5, 1, 7, 3, 12, 9.



Features of B-Tree Indexes

- Index creation is bottom-upwards whereas index navigation is top-bottom.
- The index is always perfectly balanced.
- Every-node is atleast half-full.
- All leaf blocks of the tree are always at the same depth from the root.
- All the values will be present in the leaf nodes and in a sorted order if traversed from left to right or vice-versa.
- B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- B-tree performance is good for both small and large tables, and does not degrade as the size of a table grows.
- Indexes require overhead when updating, deleting or inserting rows.

Disadvantages

While B-tree indexes are great for simple queries, they are not very good for the following situations:

1. **Low-cardinality columns**—columns with less than 200 distinct values do not have the selectivity required in order to benefit from standard b-tree index structures.

2. **No support for SQL functions**—B-tree indexes are not able to support SQL queries using Oracle's built-in functions. Oracle9i provides a variety of built-in functions that allow SQL statements to query on a piece of an indexed column or on any one of a number of transformations against the indexed column.

4. BIT-Map Indexes

Oracle bitmap indexes are very different from standard b-tree indexes. In bitmap structures, a two-dimensional array is created with one column for every row in the table being indexed. Each column represents a distinct value within the bitmapped index. This two-dimensional array represents each value within the index multiplied by the number of rows in the table. At row retrieval time, Oracle decompresses the bitmap into the RAM data buffers so it can be rapidly scanned for matching values. These matching values are delivered to Oracle in the form of a Row-ID list, and these Row-ID values may directly access the required information.

The real benefit of bitmapped indexing occurs when one table includes multiple bitmapped indexes. Each individual column may have low cardinality. The creation of multiple bitmapped indexes provides a very powerful method for rapidly answering difficult SQL queries.

For example, assume there is a motor vehicle database with numerous low-cardinality columns such as car_color, car_make, car_model, and car_year. Each column contains less than 100 distinct values by themselves, and a b-tree index would be fairly useless in a database of 20 million vehicles. However, combining these indexes together in a query can provide blistering response times a lot faster than the traditional method of reading each one of the 20 million rows in the base table. For example, assume we wanted to find old blue Toyota Corollas manufactured in 1981.

```
Select license_plat_nbr
from vehicle
where color     = ‘blue’
and make        = ‘toyota’
and year        = 1981;
```

Oracle uses a specialized optimizer method called a bitmapped index merge to service this query. In a bitmapped index merge, each Row-ID, or RID, list is built independently by using the bitmaps, and a special merge routine is used in order to compare the RID lists and find the intersecting values. Using this methodology, Oracle can provide sub second response time when working against multiple low-cardinality columns.

Structure of a Bit-map Index

To understand the structure of a bit-map index better, we will take a simple example of customer data, which contain low cardinality values such as Marital Status (Single/Married/Divorced), Region (East/West/Central), Gender (Male/Female) and Income Level (1/2/3).

Customer No.	Marital Status	Region	Gender	Income Level
101	Single	East	Male	1
102	Married	Central	Female	2
103	Married	West	Female	3
104	Divorced	Central	Female	1
105	Married	West	Male	2
106	Single	Central	Male	2
107	Single	Central	Male	1

The bit-map index for this data will have the following structure.

Customer	Status			Region			Gender		Income Level		
	Single	Married	Divorced	East	West	Central	Male	Female	1	2	3
101	1	0	0	1	0	0	1	0	1	0	0
102	0	1	0	0	0	1	0	1	0	1	0
103	0	1	0	0	1	0	0	1	0	0	1
104	0	0	1	0	0	1	0	1	1	0	0
105	0	1	0	0	1	0	1	0	0	1	0
106	1	0	0	0	0	1	1	0	0	1	0
107	1	0	0	0	0	1	1	0	1	0	0

Thus, 'Married' will have a bit-map of 0110100.
If a query comes in like
SELECT from customer data
where status = 'Married'
and region in ('East', 'Central')
and income <> '3'.
This will be evaluated by the following logic,

Iteration 1:
Status = 'Married' Region = 'East' Region = 'West' Income = '3'
Thus, we can conclude that it is only the second row of the table customer data which satisfies the search criteria.
It can be inferred from above that bit-map indexes provide the best results only if the cardinality of the columns is low. If the number of unique values in any column were high then this would significantly increase the size of the index as well as the query execution time.
Now, some calculations: for a table with N number of rows with D distinct values, the hard disk memory taken to store an index will be N * D * 1 (bit for 1 or 0) / 8 (bits per byte number of bytes).

Advantages of Bitmap Indexes
1. Reduced response time for large classes of queries.
2. A substantial reduction of space usage compared to other indexing techniques.
3. Dramatic performance gains even on very low-end hardware
4. Very efficient parallel DML (Data Manipulation Language) and loads.
5. Efficient for queries using OR, IN or '=' predicates.
6. Bitmap includes all rows, even those with null values (unlike B-Tree indexes).
7. Bitmap indexes are best in read-only situations like data warehouses or where concurrent transactions are unlikely.

Disadvantages of Bitmap Indexes
1. Bitmap index maintenance can be expensive, an individual bit may not be locked, thus a single update may lock large portions of the index.
2. If the cardinality of the values in the table were to increase, this will dramatically increase the size of the index.
3. Not suitable to find unique records in a table.

Tips on creating indexes:
The tablespace of an index's segment is either the owner's default tablespace or a tablespace specifically named in the CREATE INDEX statement. You do not have to place an index in the same tablespace as its associated table. Furthermore, you can improve performance of queries that use an index by storing an index and its table in different tablespaces located on different disk drives because Oracle can retrieve both index and table data in parallel.

5. Usage Of Indexes In BW

For the usage of these indexes in BW Objects please refer to the attached SAP Document on Oracle Indexing in BW. The document is self-explanatory with notes against each slide. Please do not miss the notes against each slide as it will help comprehend the diagrams in each slide better.

Oracle Indexing in BW.ppt

6. How to Decide What to Index

The most important thing to remember is that for the database to use an index, the column(s) must be in your query! So if a column never appears in your queries, it's not worth indexing. With a couple of caveats:

- Unique indexes are really constraints. So you may need these to ensure data quality. Even if you don't query them columns themselves.
- You should index any foreign key columns where you delete from the parent table or update its primary key.

A key point is it's not so much about how many rows you get, but how many database blocks you access. Generally, an index is useful when it enables the database to touch fewer blocks than a full table scan reads. A query may return "lots" of rows, yet access "few" database blocks.

This typically happens when logically sorting the rows on the query column(s) closely matches the physical order database stores them in. Indexes store the values in this logical order. So you normally want to use indexes where this happens. The clustering factor is a measure of how closely these logical and physical orders match. The lower this value is, the more effective the index is likely to be. And thus the optimizer to use it.

But contrary to popular belief, you don't have to have the indexed column(s) in your where clause. For example, if you have a query like this:

```
select indexed_col from table
```

Oracle Database can full scan the index instead of the table. Which is good because indexes are usually smaller than the table they're on. But remember: nulls are excluded from B-trees. So it can only use these if you have a not null constraint on the column!

If your SQL only ever uses one column of a table in the join and where clauses then you can have single column indexes and be done.

But the real world is more complicated than that. Chances are you have relatively few basic queries like:

```
select * from tab
where  col = 'value';
```

And a whole truckload with joins and filters on many columns, like:

```
select * from tab1
join   tab2
on     t1.col = t2.col
join   tab3
on     t2.col2 = t3.col1
where  t1.other_col = 3
and    t3.yet_another_col
order  by t1.something_else;
```

As discussed earlier, if you're using bitmaps you can create single column indexes. And leave the optimizer to combine them as needed. But with B-trees this may not happen. And it's more work when it does.

The Access Restrictions defined for universes in the universe design tool and their assignments are converted into the equivalent Data and Business Security Profiles when a secured universe is converted.

From the Security Editor, the developer can run a query on a universe in a repository. The query is then secured by the Data Security Profiles and Business Security Profiles that apply to the user used to log into the Security Editor.

7. Conclusion

The tablespace of an index's segment is either the owner's default tablespace or a tablespace specifically named in the CREATE INDEX statement. You do not have to place an index in the same tablespace as its associated table. Furthermore, you can improve performance of queries that use an index by storing an index and its table in different tablespaces located on different disk drives because Oracle can retrieve both index and table data in parallel.

References

[1]. http://www.lsbu.ac.uk/bcim/ (London South Bank University: Business Computing & Information Management)

[2]. http://www.siue.edu/ (Southern Illinois University)

[3]. http://www.ohio.edu/engineering/index.cfm (Russ College of Engineering & Technology, Ohio University)