ORACLE8I INDEXING CHOICES: BEST OF BREED

John Jay King, King Training Resources

ABSTRACT

Oracle8i provides expanded options for indexing. Index choices include: traditional B-Tree indexes, Reverse-key indexes,

Bit-map indexes, Hash indexes, and Index-Organized Tables. In addition, indexing is now available when using a function or expression on an index column. Choosing the proper index style is an important part of database and application design. Significant performance differences can be achieved by the careful creation of indexes and their subsequent use in applications. This session compares and contrasts the various options available and how to choose from among them. Specific topics include: B-tree indexes, Reverse key indexes, Bitmap indexes, Hash indexes, Index-Organized Tables, the ability to base indexes on functions or expressions, negative impacts of indexes, and hints involving indexes. Attendees will be better equipped to build effective Oracle8i applications.

INTRODUCTION

Choosing the appropriate type of index may sometimes be as important as the choice to use an index or not. In the past indexing decisions were largely a matter of what columns to index. Today, we need to explore the options further and also decide how columns should be indexed. The type of index selected may impact performance dramatically. Oracle8i provides a comprehensive set of indexing options that address most application needs. Most of the information in this paper also applies to Oracle8 and to a lesser-extent Oracle 7.3. Notations in the text indicate features that require Oracle8i.

Indexes are traditionally used to guarantee unique values (if defined as unique) and to speed performance by using a key value's rowid to access data rather than a row-by-row search of the tablespace. However, changes to table data that includes index keys can multiply I/O times several times; slowing the speed of inserts, updates, and deletes to keep indexes synchronized.

TYPES OF INDEXES

Oracle8i offers different types of indexes: B-Tree (traditional) indexes, Hash-cluster indexes, Bitmap indexes, and Index-Organized Tables. The B-Tree index offers the Reverse-Key variation and both B-Tree and Bitmap indexes allow Function/Expression-based indexes.

Choosing the appropriate indexing method is frequently an important performance-related question. Except for Index-Organized Tables, Oracle's indexes provide a means of translating a key value (one or more columns in the database row) into a rowid. Index use reduces the I/O required to obtain a row to: the I/O or calculations necessary to find the rowid, followed by a direct access using the rowid. This is usually faster than reading all possible rows looking for a match (table scan).

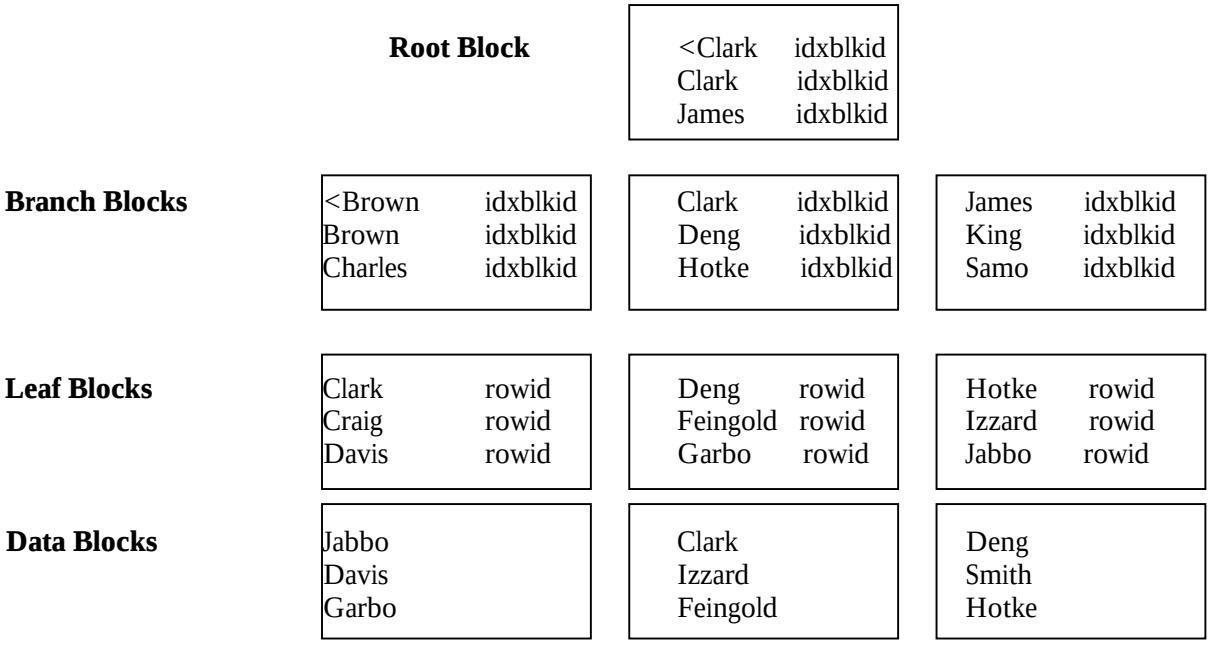
B-Tree indexes store key values sequentially and are traversed from: root block, to branch block (sometimes multiple levels of branch blocks), to leaf block, to data block containing the row. Hash-cluster indexes convert the key value using an algorithm to determine which data block to read. Bitmap indexes contain a bit (0 or 1) for each key value that corresponds to every rowid in the table. Index-Organized Tables actually contain the table row data, so, once an index value is found in the index, the data is immediately available. The different index types are discussed in more detail later in this paper.

B-TREE INDEXES

The original type of index supported by Oracle is the B-tree index. These indexes provide a linked-list type access using a file that contains the key values in sequence, with the rowid. In the case of non-unique indexes, multiple rowids might be stored for a given key value. B-tree indexes are a good choice when a given key value is unique or has few duplicates in a table.

A B-tree index is stored as a set of pages in a hierarchy. The first index page is called the "root" and it points to lower-level "branch" index pages. Multiple levels of "branch" pages might exist depending upon the size of table and index keys. The lowest level index page is called a "leaf" page and it contains the address of data blocks containing table rows. The actual key value is stored (unless using key compression) in the various levels on index. If indexes are based upon multiple columns (concatenated keys) the larger index entries can increase the number of levels necessary to address a table's data. Some organizations avoid this issue by assigning key values (sometimes using a sequence generator) that correspond to a concatenated key value. This way, the index entries are smaller, resulting in fewer levels in the index, resulting in faster access (not to mention simpler SQL statements!). Rows with NULL key values are not represented by the B-tree index. Some organizations use a null-capable column as a switch to gain faster access to values that are not null.

Each index requires overhead when updating, deleting, or inserting rows in the base table. If a table is read-only, this additional overhead is avoided. If a table is volatile, it is a good idea to limit the number of indexes that must be maintained.



In the example above, a search for "Garbo" would first read the root block, then the branch block beginning with "Clark" and finally the leaf block beginning with "Deng" to obtain the rowid of the "Garbo" row. This means that the record is retrieved with four block reads (at most in this example) rather than reading each data block in the table.

Index rows are stored in sequence according to key values (as above), however, data is not necessarily sequenced. The Clustering Factor statistic is used by the optimizer to determine whether rows of subsequent reads are likely to be on pages already in the cache.

A "Fast Full Scan" is sometimes selected by the optimizer so that rather than traversing the index hierarchy the Leaf Blocks are read using the same multi-block I/O used by table scans.

Key design can cause some key ranges to become more dense than others, especially when key values are geographic in nature. When key designs lead to index paths being more densely or sparsely populated than other index paths, insert or update of key values can lead to deeper levels of index than would otherwise be necessary. An index "tree" that is unbalanced may inhibit performance of the "Fast Full Scan," rebuilding the index will redistribute the levels. Another solution to the problem of unbalanced key distribution may be the Reverse-Key index.

REVERSE KEY INDEXES

If keys and usage are heavily clustered in a table (e.g. key is generated as next sequential number, and most activity occurs on recent entries; or geographically-oriented and many records are in a specific area), then, Reverse Key Indexes might speed things up. A good example of a geographically oriented key is the Social-Security Number used for taxpayer identification in the United States. This number's first three digits was originally designed to show the region of the country where the person resided when the number was issued. In systems that service the eastern coast of the US, the social-security numbers of New York-based values might skew the indexes of a heavily inserted table using Social-Security Number as its key.

A Reverse Key Index is simply a standard index with the key values stored in reversed form (e.g. '1234' becomes '4321', '1235' becomes '5321'). Actual table data is not changed. By reversing the key values the index blocks might be more evenly distributed reducing the likelihood of some index paths being densely or sparsely populated. This is especially useful in Parallel Processing environments but might be useful any time key ranges become too dense and cause extra processing due to the key density. In most non-Parallel environments, Reverse Key indexes may actually cause significant performance degradation. Be sure to carefully test Reverse-Key indexes to verify that real benefits are occurring.

Syntax to create a Reverse-Key index is to simply add the word REVERSE after the column specification:

```
CREATE INDEX employee_ssn_rev
      ON employee_table (ssn) REVERSE
      /* … rest of index definition … */;
```

FUNCTION / EXPRESSION-BASED INDEXES (ORACLE8I)

Normally, developers attempt to use index columns wherever possible to speed access to large database tables. Until now, this has often been hampered by limitations of index design. Until Oracle8i, index columns always represented the actual value of the column in the database and an SQL WHERE clause needed to specify the original column value or the index would be ignored. Now, an index can represent a column value after some function or expression has been applied. As long as an SQL WHERE clause references a column value after a function or expression exactly as specified (case-insensitive and blanks are ignored) in the index creation, an index may be used by the system. Given the following create index lines:

```
CREATE INDEX … ON EMP (UPPER(ENAME)) …
CREATE INDEX … ON INVENTORY (IN_STOCK + ON_ORDER) …
```

The following two WHERE clause lines may be indexed by the optimizer (requires cost-based optimization).

```
SELECT … FROM EMP WHERE UPPER(ENAME) = UPPER(:hostvar) …
SELECT … FROM INVENTORY WHERE IN_STOCK + ON_ORDER > :LARGE_QTY_ITEMS …
```

These indexes will only be used if cost-based optimization is invoked, statistics are available on both table and function/expression-based index and Query Rewrite is properly set.

```
Alter session set query_rewrite_enabled = true;
```

For enabling user-defined functions:

```
Alter session set query_rewrite_integrity = trusted;
```

BITMAP INDEXES

When creating B-Tree indexes, columns with few values over many rows (low-cardinality) should be avoided. This means that columns like the Country Code of a customer or Gender of a customer would make poor index columns

due to the small number of different values in the table. Bitmap indexes offer performance improvement when created for columns with a relatively small number of values.

When a bitmap index is created, the index contains the key value and a bitmap listing the value of 0 or 1 (yes/no) for each row indicating whether the row contains that value of not.

For an index of customers in a particular country (presume that our organization only does business in the United States (US), United Kingdom (UK), Japan (JA), and Australia (AU)):

COUNTRY_CD=AU	COUNTRY_CD=JA	COUNTRY_CD=UK	COUNTRY_CD=US
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	0	1

Each entry in the bitmap corresponds to a row in the table, a value of 1 indicates which value that row contains. Bitmaps include all rows, even those with NULL values (unlike B-Tree indexes). Bitmap indexes are significantly smaller than B-Tree indexes for the same table.

Bitmap indexes are most effective when used with equality-type tests (= or IN). Bitmap indexes are ideally used in conjunction with other indexes to reduce the number or rows returned. The following query will be much quicker given bitmap indexes on the low-cardinality columns COUNTRY_CODE, GENDER, and CREDIT_CARD.

```
SELECT CUSTOMER_ID, LAST_NAME, BALANCE
     FROM NON_GOV_CUSTOMERS
     WHERE COUNTRY_CODE IN ('AU','UK')
          AND GENDER = 'M'
          AND CREDIT_CARD = 'AX';
```

Bitmap index maintenance can be quite expensive. Since an individual bit may not be locked, a single update locks potentially large portions of the index. It is best if Bitmap indexes are used primarily in read-only situations like data warehouses or where concurrent transactions are unlikely.

HASH-CLUSTER INDEXING

In B-Tree and Bitmap indexes, the key value is used to find rows that match. This process requires I/O to process the index before obtaining the data row(s) desired. A hash cluster determines how to find a row by applying an algorithm based upon the key value, the only read is to get the data row. Rows are stored together based upon their hash value.

The hashing algorithm may be specified to use Oracle's internal algorithm, to use the cluster key as the hash function, or to use a user-defined hash function. Hashing is not new to the computing industry, entire books have been devoted to developing optimal hash algorithms.

Hash clusters can be the fastest type of index given that: very-high-cardinality columns are used for the index, only equal (=) tests are used, index values do not change, the number of rows and rows/index value are known at index creation time, and only minimal insert/delete activity will occur. Only one Hash-cluster is allowed per table. The size of the hash index should be known at index creation and it should allow for distribution of rows with few collisions when hashing a specific key (ideally no collisions). If each key hashes to a unique value optimal results will occur. If many keys hash to the same value (a collision) then chaining is likely to occur reducing the benefit of the Hash cluster. There is no graceful way to reorganize a Hash-cluster, the index must be dropped and recreated.

INDEX-ORGANIZED TABLES

Beginning with Oracle8, CREATE TABLE has an ORGANIZATION INDEX clause that causes the table's data to be incorporated into the B-tree index representing the table's Primary Key (and only the primary key). This means that the table's data is always available in sequence by Primary Key and many sorts can be avoided by the optimizer. Oracle8i adds the ability to create secondary indexes for Index-Organized tables (not allowed in Oracle8.0). Index-Organized Tables are especially useful for "lookup" type tables that are often used to create drop-down lists in GUI interfaces; since table data is already in the desired sequence, access is faster.

A sequential scan of an index-organized table yields all values in sequence by primary key.

The advent of Index-Organized Tables has caused the new term “heap organized” to describe traditional tables.

The entire Index-Organized Table is stored in the index and has no specific rowid. Instead, Oracle uses a “virtual rowid” to provide secondary indexing capability. This is quicker than a scan of the Index-Organized Table, but, not quite as fast as a traditional B-Tree secondary index.

Index-Organized Tables work best when there are few columns in the table/index (a “narrow” table) and the size of a row is small compared to the size of a block. Index-Organized columns may not contain LONG columns, but may contain BLOB, CLOB, or BFILE data. Index-Organized Tables may not be used in a CLUSTER.

Comparing Index Strengths and Weaknesses

For high-cardinality key values, B-tree indexes and Hash-cluster indexes may offer the best alternative. B-Tree indexes work with all types of comparisons and gracefully shrink and grow as table data changes. Hash-clusters work only with equal tests and table growth is a significant problem.

For low-cardinality key values that are not changed by concurrent transactions, Bitmap indexes are often superior to B-tree indexes. Hash-clusters are not a good choice for low-cardinality data (many collisions).

If a key design causes dense or sparse population of index values, it may be good to test using Reverse-key indexes to see if overall performance is improved.

Index-Organized tables are a good choice if tables have few columns, have relatively small rows, and are frequently sorted by Primary Key.

Using Hints to Suggest Indexes

Sometimes, the optimizer may not choose to use an index, or might not choose to use it in the way you intended. In this case you might want to consider using Hints to control the processing of the SQL. Through careful testing of hints, it is sometimes possible to improve on the optimizer’s decisions. Be sure to use trace information in addition to Explain output when determining whether or not a hint is useful.

Be careful! Test statements thoroughly before and after adding Hints, your “help” might make things worse (oops!).

Revisit decisions to use Hints regularly (at least as often as installations of Oracle releases, probably more frequently). A performance improvement due to hints today, using today’s data, in today’s environment might not be a good thing tomorrow.

User-Defined Index Types

Oracle8i provides the ability to create a user-defined index type to provide indexing of complex data such as documents, images, video clips, spatial data, or audio clips. Creation uses the object features first introduced in Oracle8 and allows creation of indexes specifically designed for complex applications such as On-Line Analytical Processing (OLAP).

User-defined indexes may be used in conjunction with user defined operators (CREATE OPERATOR).

Specifics concerning User-Defined Index Types may be found in the Oracle8i Concepts manual, no further discussion is provided in this paper.

Conclusion

This presentation attempted to illustrate the indexing capabilities of Oracle8i and to help system developers when deciding not just what columns to index but how to index them. This paper presented various indexing options available and suggested when choosing a particular type of index might be the best choice. As with all performance related issues, test, test, and test again. Just because something appears on paper to be a good choice does not mean that it is best for you. Also, any performance oriented decision must be revisited periodically to make sure that the best choice is still being made.

ABOUT THE AUTHOR

John King is a Partner in King Training Resources, a firm providing instructor-led training since 1988 across the United States and Internationally. John has worked with Oracle products and the database since Version 4 and has been providing training to application developers since Oracle Version 5. He has presented papers at various industry events including: IOUG-A Live!, UKOUG Conference, EOUG Conference, ECO, RMOUG Training Days, and the ODTUG conference.


```
John Jay King
King Training Resources
6341 South Williams Street
Littleton, CO 80121-2627
U.S.A.
Phone:     1.303.798.5727  1.800.252.0652 (within the U.S.)
Fax:       1.303.730.8542
Email:     john@kingtraining.com
Website:   www.kingtraining.com
```


If you have any questions or comments, please contact me in the fashion most convenient to you. Copies of this paper are available for download from King Training Resources upon request (www.kingtraining.com).

BIBLIOGRAPHY

Oracle8i SQL Reference, Oracle Corporation

Oracle8i Concepts, Oracle Corporation

Oracle8i Administrator's Guide, Oracle Corporation