Module12-indexes.htm; updated June 11, 2012; Some figures shown in these notes are from Oracle document D11321GC11.

Module 12 – Managing Indexes

Objectives

- Learn basic concepts about indexes including types of indexes.
- Create B-Tree, Bitmap, Hash Cluster, Global/Local, Reverse Key, and Function-based indexes.
- Reorganize indexes.
- Drop indexes.
- Obtain data dictionary information about indexes.

Introduction

Indexes are totally optional structures that are intended to speed up the execution of SQL statements against table data and cluster data.

From your earlier studies, you should realize that **Indexes** are used for **direct access** to a particular row or set of rows in a table. From your study of database management systems, you learned that indexes are most typically organized as some type of **tree structure**. You may not have studied **Bitmap Indexes** – you will also study this type of index in this course.

Oracle Database provides several indexing schemes that provide complementary performance functionality. These are:

- B-tree indexes: the default and the most common
- B-tree cluster indexes: defined specifically for cluster
- Hash cluster indexes: defined specifically for a hash cluster
- Global and local indexes: relate to partitioned tables and indexes
- Reverse key indexes: most useful for Oracle Real Application Clusters applications
- Bitmap indexes: compact; work best for columns with a small set of values
- Function-based indexes: contain the precomputed value of a function/expression
- Domain indexes: specific to an application or cartridge.

Index Concepts and Facts

An index can be composed of a **single column** for a table, or it may be comprised of **more than one column** for a table. An index based on more than one column is termed a **concatenated** (or **composite**) index.

Examples:

- The **SSN** column serves as an index key that tracks individual students at a university.

- The concatenated **primary key index** for an **ENROLL** table (**SSN** + **SectionID** + **Term** + **Year**) is used to track the enrollment of a student in a particular course section.

- 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.

- 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). An example would be an index on U.S. Mail zip codes.

- 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.

To create an index in your own schema:

- The table to be indexed must be in your schema, OR
- You have the **INDEX** privilege on the table to be indexed, OR
- You have the **CREATE ANY INDEX** system privilege.

To create an index in another schema:

- You have the **CREATE ANY INDEX** system privilege, AND
- The owner of the other schema has a quota for the tablespace that will store the index segment (or **UNLIMITED TABLESPACE** privilege).

Loading Data

Data initially loaded into a table will load more efficiently if the index is created after the table is created. This is because the index must be updated after each row insertion if the index is created before loading data.

Creating an index on an existing table requires sort space – typically memory values that are paged in and out of segments in the **TEMP** tablespace allocated to a user. Users are also allocated memory for index creation based on the **SORT_AREA_SIZE** parameter – if memory is insufficient, then swapping takes place.

Guidelines for Creating an Index

Use the following guidelines for determining when to create an index:

- Create an index if you frequently want to retrieve less than 15% of the rows in a large table.
    - The percentage varies greatly according to the relative speed of a table scan and how the distribution of the row data in relation to the index key.
    - The faster the table scan, the lower the percentage; the more clustered the row data, the higher the percentage.
- To improve performance on joins of multiple tables, index columns used for joins.

Index columns with these characteristics:

- Take into consideration the typical queries that will access the table. A **concatenated index** is only used by Oracle in retrieving data when the **leading column** of the index is used in a query's **WHERE** clause.
    - The order of the columns in the CREATE INDEX statement affect query performance – specify the most frequently used columns first.
    - An index that specifies **column1+column2+column3** can improve performance for queries with WHERE clauses on **column1** or on **column1+column2**, but will not be used for queries with a WHERE clause that just uses **column2** or **column3**.
- Values are relatively unique in the column (but not number columns that store currency values).
- There is a wide range of values (good for regular B-tree indexes).

- There is a small range of values (good for bitmap indexes).
- The column is a **virtual column** (can create both unique and non-unique indexes)
- The column contains many nulls, but queries often select all rows having a value. In this case, use the following phrase:

WHERE COL_X > <some numeric value goes her>

Using the preceding phrase is preferable to:

WHERE COL_X IS NOT NULL

This is because the first uses an index on **COL_X** (assuming that **COL_X** is a numeric column).

Small tables don't need indexes – but it is a good idea to still index the Primary Key.

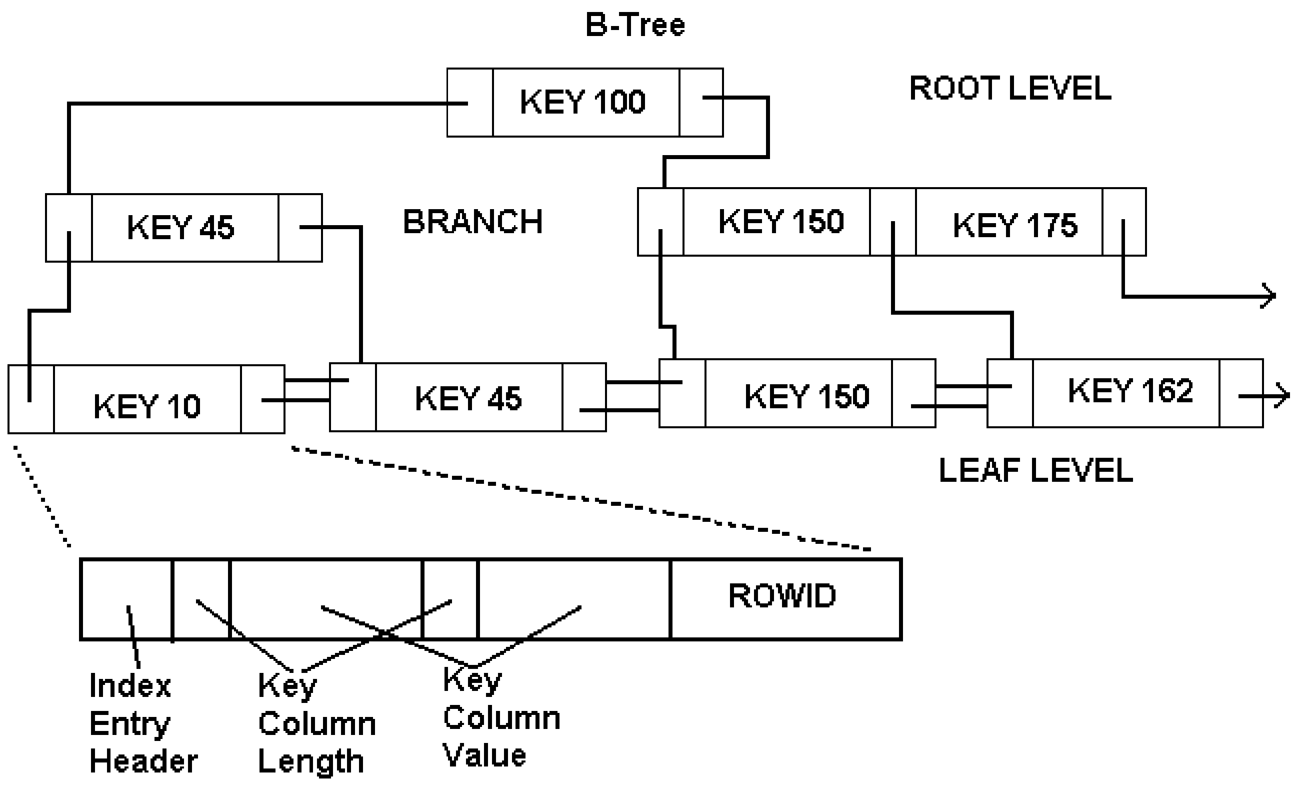
Do not index columns where:

- There are many nulls in the column and you do not search on the not null values.
- LONG and LONG RAW columns – these cannot be indexed.

B-Tree Index

B-Tree Structure

This figure illustrates in a very simple way the 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 at the **leaf level**.



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 columns NULL.

Leaf Index Format: 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.

Data Manipulation Language Effects: Any DML on a table also causes the Oracle Server to maintain the associated indexes.

- When a row is **inserted** into a table, the index must also be updated. This requires the physical **insertion** of an index entry into the index tree structure.

- When a row is **deleted** from a table, the index only has the entry "**logically**" deleted (turn a delete bit from **off** to **on**). The space for the deleted row is not available for new entries until all rows in the block are deleted.

- When a row key column is **updated** for a table, the index has **both** a **logical deletion** and a **physical insertion** into the index.

- **PCTFREE** has no effect on an index except when the index is created. New entries to an index may be added to an index block even if the free space in the block is less than the PCTFREE setting.

  1. If an indexed table has lots of rows to be inserted, set **PCTFREE** high to accommodate new index values.

  2. If the table is static, set **PCTFREE** low.

  3. **PCTUSED** cannot be specified for indexes.

Creating B-Tree Indexes

An example **CREATE INDEX** command for a normal B-tree index is given here.

```
CREATE [UNIQUE] INDEX USER350.Orders_Index
    ON USER350.Orders(OrderId)
    PCTFREE 20
    INITTRANS 6 MAXTRANS 10
    LOGGING
    TABLESPACE Index01;
```

```
CREATE UNIQUE INDEX hr.jobs_Index
    ON hr.jobs(job_title)
    PCTFREE 20
```

- The **UNIQUE** clause specifies unique entries - the **default** is **NONUNIQUE**. Note that the owner's schema (**USER350**) is specified – this is optional.

- The **PCTFREE** parameter is only effective when the index is created - after that, new index block entries are made and **PCTFREE** is ignored. **PCTFREE** is ignored because entries are not updated - instead a logical delete and physical insert of a new index entry is made.

- **PCTUSED** **cannot** be specified for an index because updates are not made to index entries.

Use a **low** **PCTFREE** when the indexed column is **system generated** as would be the case with a **sequence** (sequence indexes tend to increase in an ascending fashion) because new entries tend to be made to new data blocks - there are no or few insertions into data blocks that already contain index entries.

Use a **high** **PCTFREE** when the indexed column or set of columns can take on **random** values that are **not predictable**. Such is the case when a new **Orderline** row is inserted - the **ProductID** column may be a **non-unique foreign key index** and the product to be sold on an **Orderline** is not predictable for any given order.

The **Default** and **Minimum** for **INITTRANS** is **2**. The limit on **MAXTRANS** is **255** - this number would be inordinately large.

By default, **LOGGING** is on so that the index creation is logged into the redo log file. Specifying **NOLOGGING** would increase the speed of index creation initially, but would not enable recovery at the time the index is created.

Interestingly, Oracle will use existing indexes to create new indexes whenever the key for the new index corresponds to the leading part of the key of an existing index.

Indexes and Constraints

The **UNIQUE** and **PRIMARY KEY** constraints on tables are enforced by creating indexes on the unique key or primary key – creation of the index is automatic when such a constraint is enabled.

You can specify a **USING INDEX** clause to control the creation process. The index created takes the name of the constraint unless otherwise specified.

Example creating a **PRIMARY KEY** constraint while creating a table.

```
CREATE TABLE District (
    District_Number     NUMBER(5)
        CONSTRAINT District_PK PRIMARY KEY,
    District_Name       VARCHAR2(50)
    )
  ENABLE PRIMARY KEY USING INDEX
  TABLESPACE Index01
  PCTFREE 0;
```

Key-Compressed Index

This is a B-tree with compression – compression eliminates duplicate occurrences of a key in an index leaf block.

```
CREATE INDEX Emp_Name ON Emp (Last_Name, First_Name)
    TABLESPACE Index01
    COMPRESS 1;
```

This approach breaks an index key into a prefix and suffix entry in the index block. Compression causes sharing of the prefix entries among all suffix entries and save lots of space allowing the storage of more keys in a block.

Use key compression when:

- The index is **non-unique** where the **ROWID** column is appended to the key to make the index key unique.
- The index is a non-unique multicolumn index – example: **Zip_Code + Last_Name**.

Creating a Large Index

When creating an extremely large index, consider allocating a larger temporary tablespace for the index creation using the following procedure:

1. Create a new temporary tablespace using the **CREATE TABLESPACE** or **CREATE TEMPORARY TABLESP**ACE statement.

2. Use the **TEMPORARY TABLESPACE** option of the **ALTER USER** statement to make this your new temporary tablespace.

3. Create the index using the **CREATE INDEX** statement.

4. Drop this tablespace using the **DROP TABLESPACE** statement. Then use the **ALTER USER** statement to reset your temporary tablespace to your original temporary tablespace.

This procedure avoids the problem of expanding the usually shared, temporary tablespace to an unreasonably large size that might affect future performance.

Bitmap Index

Bitmap Index Structure

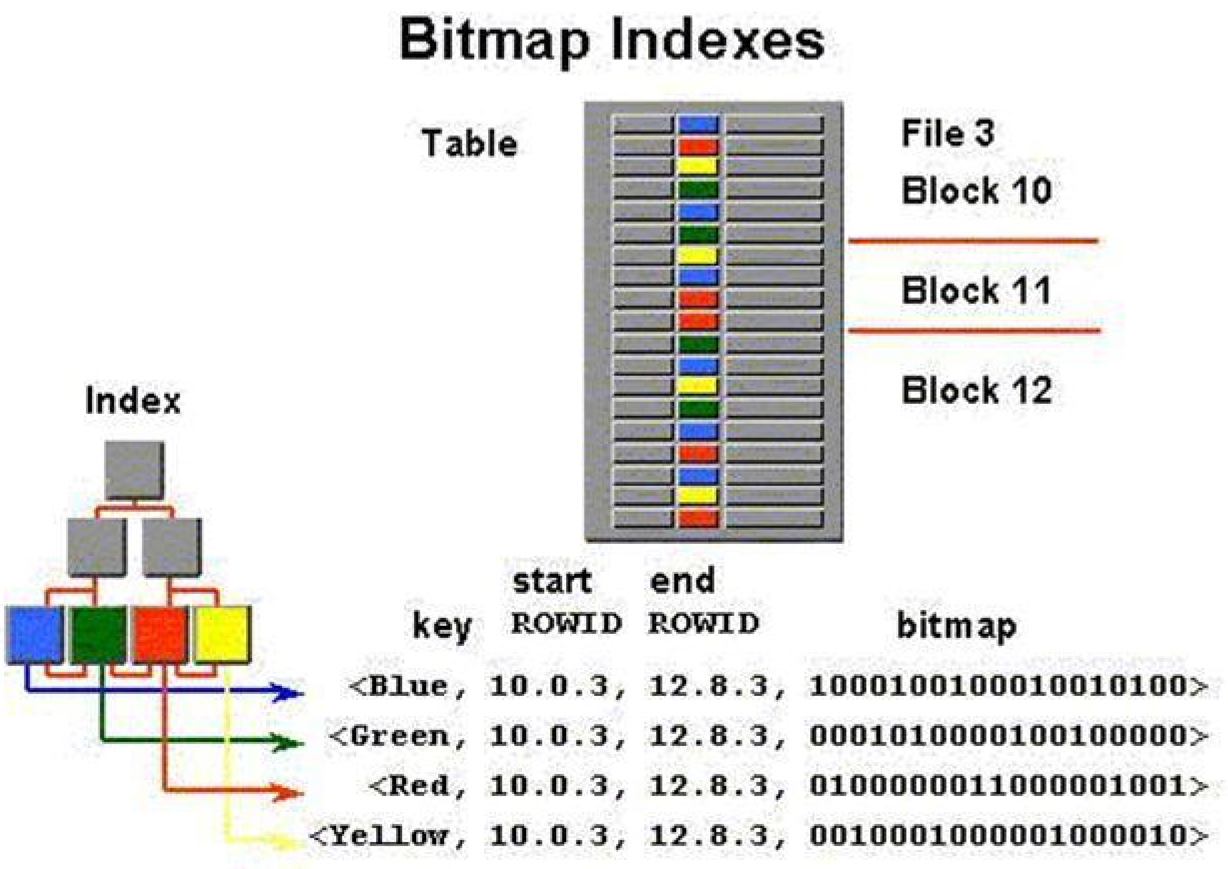
Bitmap indexes are alternatives to B-tree indexes and are only used to create secondary key indexes. They are used in certain situations:

- If a table has **millions of rows** and there are very **few** distinct values for index entries, for example, indexes on zip codes could have many, many rows for a single zip code value, then a bitmap index may perform well.

- A bitmap index may best support WHERE conditions involving the OR operator.

- A bitmap index may work best for low update activity tables or read-only key columns.

A **bitmap index** is also organized as a **B-tree** structure; however, the **leaf nodes** in a bitmap index stores a **bitmap** for each key value - it does **NOT** store the ROWIDs. Instead, each **bit** corresponds to a possible ROWID. If the bit is set on, the row with the corresponding ROWID contains the key value. This figure illustrates this concept. In the figure, the key consists of only one column, and the first entry has a logical key value of **Blue**, the second is **Green**, the third **Red**, and the fourth **Yellow**.



```
            start     end
  key      ROWID    ROWID            bitmap
  <Blue,   10.0.3,  12.8.3,  1000100100010010100>
 <Green,   10.0.3,  12.8.3,  0001010000100100000>
   <Red,   10.0.3,  12.8.3,  0100000011000001001>
<Yellow,   10.0.3,  12.8.3,  0010001000001000010>
```

Comparisons and Limitations: Updates to key columns for **bitmap indexes** require locking a significant portion of the index in order to perform the update. This table compares B-tree and bitmap indexes.

B-Tree	Bitmap
Suitable for high-cardinality columns	Suitable for low-cardinality columns
Updates on keys relatively inexpensive	Updates to key columns very expensive

Inefficient for queries using OR logical operator	Efficient for queries using OR logical operator
Used mostly for On-Line Transaction Processing	Very useful for Data Warehousing

Creating Bitmap Indexes

Example **CREATE BITMAP INDEX** command:

```
CREATE BITMAP INDEX USER350.Products_Region_Idx
    ON USER350.Products_Region (RegionId ASC)
    PCTFREE 40
    INITTRANS 6 MAXTRANS 10
    LOGGING
    TABLESPACE Index01;
```

A **BITMAP** index **cannot** be specified as **unique** since they are only used for **secondary key** indexes.

You can specify an **init.ora** parameter **CREATE_BITMAP_AREA_SIZE** to specify the amount of memory allocated for storing bitmap segments.

- The **default** memory allocated is **8MB**.
- If the memory allocated for bitmap index segment creation is larger, then indexes are created faster.

- If there are only a few unique values for the index key field, then you can allocate much less memory to the creation of bitmap index segments, perhaps only a few kilobytes.

Other General Topics

This section covers altering, rebuilding, coalescing, and dropping indexes. It also covers validating indexes and identifying unused indexes. Guidelines are provided for when to index columns.

Altering Index Storage Parameters

The **ALTER INDEX** command is used primarily to alter storage parameters. Other uses include:

- Rebuild or coalesce an existing index
- Deallocate unused space or allocate a new extent
- Specify parallel execution (or not) and alter the degree of parallelism
- Alter storage parameters or physical attributes
- Specify LOGGING or NOLOGGING
- Enable or disable key compression
- Mark the index unusable

- Make the index invisible
- Rename the index
- Start or stop the monitoring of index usage

You cannot alter index column structure.

Use of the **ALTER INDEX** command is straight-forward as illustrated in this example.

```
ALTER INDEX USER350.Products_Region_Idx
    STORAGE (NEXT 200K MAXEXTENTS 200);
```

One of the most common changes is to increase **MAXEXTENTS** for an index as a system grows in size.

This example enforces the primary key constraint that might have previous been relaxed.

```
ALTER TABLE Student
     ENABLE PRIMARY KEY USING INDEX;
```

Allocating and De-allocating Index Space: If a DBA is going to insert a lot of rows into a table, performance can be improved by adding extents to the indexes associated with the table prior to doing the inserts. This improves performance by avoiding the dynamic addition of index extents. This can be accomplished by using the **ALTER INDEX** command with the **ALLOCATE EXTENT** option.

```
ALTER INDEX USER350.Products_Region_Idx
    ALLOCATE EXTENT (SIZE 400K
    DATAFILE

'/u01/student/dbockstd/oradata/USER350index01.dbf');
```

You can **DEALLOCATE** unused index space after the insertions are complete. This will free up space that is not in use within the tablespace.

```
ALTER INDEX USER350.Products_Region_Idx
    DEALLOCATE UNUSED;
```

Creating an Index Online

Creating an **Online Index** allows Data Manipulation Language (DML) operations against the table while the index build is being completed. DDL operations on the table are not allowed during the index creation process.

```
CREATE INDEX Manager_Employee_Idx ON Employee

    (Manager_ID, Emp_SSN) ONLINE;
```

There are some restrictions:

- Temporary tables cannot be created/rebuilt online.

- Partitioned indexes must be created/rebuilt one partition at a time.

- You cannot de-allocate unused space during an online rebuild.

- You cannot change the PCTFREE parameter for the whole index.

Rebuilding Indexes

You may improve system performance by rebuilding an index that is highly fragmented due to lots of physical insertions and logical deletions. Such may be the case for a **SALES_ORDER** table where filled orders are deleted over time.

```
ALTER INDEX USER350.Products_Region_Idx
    REBUILD
    TABLESPACE Index01;
```

- If you **rebuild** an index from an existing index, the rebuild operation is more efficient because the index information does not need to be sorted - it is already sorted.

- You can specify a **new tablespace** if necessary for the rebuilt index.

- The **older index** is **deleted** after the **new index** is rebuilt. You must have sufficient space in the tablespace for the old/new index during the rebuild operation.

- **Queries** use the old index while the rebuild operation is under way.

- You can also specify the **ONLINE** option when rebuilding an index – this allows DML updates to process even while the index is being rebuilt.

```
ALTER INDEX USER350.Products_Region_Idx
    REBUILD ONLINE;
```

Function-Based Indexes

These indexes support queries that use a value returned by a function or expression in a **WHERE** clause.

- The index computes and stores the value from the function or expression in the index.

- If user-based functions are used, the function must be marked **DETERMINISTIC** (stochastic functions won't work).

- If the function is owned by another user, you must have the **EXECUTE** privilege on the function object.

- The table must be analyzed after the index is created.

- The query must be guaranteed to not need **NULL** values from the indexed expression because NULL values are not stored in indexes.

Example --- this defines a function-based index (**Last_Name_Caps_Idx**) based on the **UPPER** function – this facilitates use of the **UPPER** function when specifying a **WHERE** clause condition and ensures use of the index to return values efficiently.

```
CREATE TABLE Faculty (

    Faculty_ID NUMBER(5)

       CONSTRAINT Employee_PK PRIMARY KEY,

    Last_Name  VARCHAR2(20),

    First_Name VARCHAR2(20)

    )

  TABLESPACE Data01;

INSERT INTO Faculty VALUES (1, 'Bock', 'Douglas');

INSERT INTO Faculty VALUES (2, 'Bordoloi', 'Bijoy');

INSERT INTO Faculty VALUES (3, 'Sumner', 'Mary');

INSERT INTO Faculty Values (4, 'BOCK', 'Ronald');
```

Now create the (non-unique) index, then select from the table.

```
CREATE INDEX Last_Name_Caps_Idx ON Faculty
(UPPER(Last_Name))

TABLESPACE Index01;
```

Note that the Oracle optimizer considers using the index because of the use of the **UPPER** function in the **WHERE** clause.

```
SELECT Faculty_ID, Last_Name, First_Name

FROM Faculty

WHERE UPPER(Last_Name) LIKE 'BOCK';
```

```
    EMP_ID LAST_NAME            FIRST_NAME
---------- -------------------- --------------------
         1 Bock                 Douglas
         4 BOCK                 Ronald
```

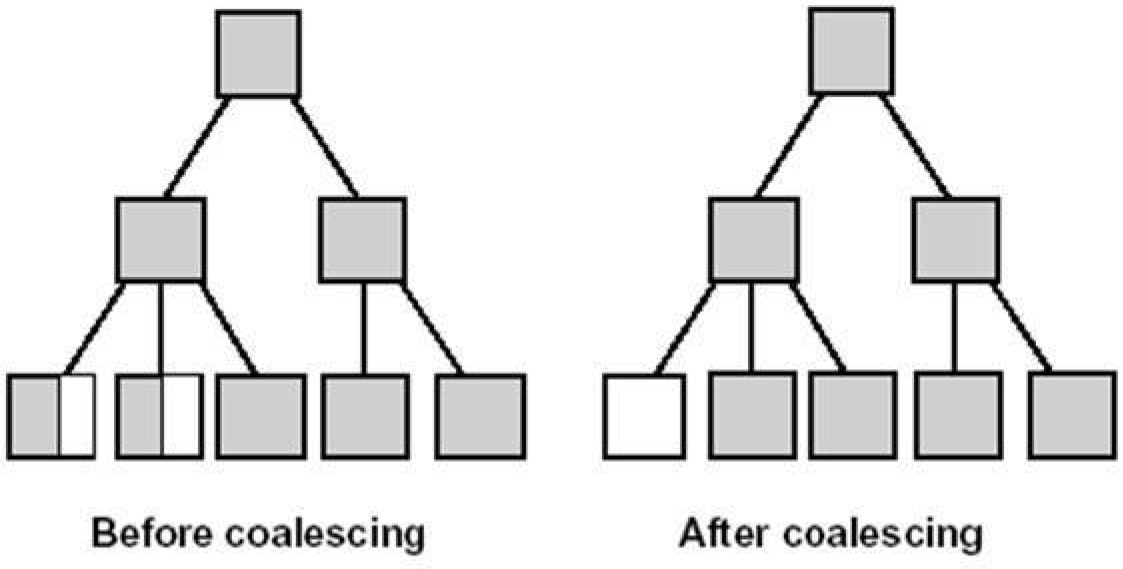
Coalescing Indexes

This is an alternative to the **ALTER INDEX…REBUILD** command – it eliminates index fragmentation. Coalesce on an index is a block rebuild that is done online.

In situations where a B-tree index has leaf blocks that can be freed up for reuse, you can merge leaf blocks using the **ALTER INDEX…COALESCE** statement.

```
ALTER INDEX USER350.Products_Region_Idx COALESCE;
```

In this figure, the first two leaf node blocks are about 50% full – fragmentation is evident. After coalescing, blocks are merged freeing up a block for reuse.



When to Rebuild or Coalesce

Rebuild Index	**Coalesce Index**
To quickly move an index to another tablespace.	Cannot move the index to another tablespace.
Requires more disk space.	Operation does not require more disk space.
Creates new index tree – height of tree may shrink.	Only coalesces leaf blocks.

Enables changing storage and tablespace parameters.	Frees up index leaf blocks for use.

Checking Index Validity

The **ANALYZE INDEX** command with the **VALIDATE STRUCTURE** option can be used to check for block corruption in an index.

```
ANALYZE INDEX dbock.pk_course VALIDATE STRUCTURE;
```

You can query the **INDEX_STATS** view and obtain information about the index.

```
SQL> ANALYZE INDEX Course_PK VALIDATE STRUCTURE;

Index analyzed.

SELECT blocks, pct_used, distinct_keys, lf_rows,
del_lf_rows
FROM index_stats;
```

```
    BLOCKS   PCT_USED DISTINCT_KEYS    LF_ROWS
DEL_LF_ROWS

---------- ---------- ------------- ----------
-----------

        15          1             2          2
0
```

An index needs to be reorganized when the proportion of deleted rows (**DEL_LF_ROWS**) to existing rows (**LF_ROWS – short for Leaf**) is greater than 30%.

Dropping an Index

DBAs drop indexes if they are no longer needed by an application. Here are reasons for dropping an index:

- The index is no longer required.
- The index is not providing anticipated performance improvements for queries issued against the associated table. For example, the table might be very small, or there might be many rows in the table but very few index entries.
- Applications do not use the index to query the data.
- The index has become invalid and must be dropped before being rebuilt.
- The index has become too fragmented and must be dropped before being rebuilt.

- A large data load is to be processed – drop indexes **prior to** large data loads - this improves performance and tends to use index space more efficiently.
- You also drop indexes that are marked by the system as **INVALID** because of some type of instance failure, or if the index is corrupt.

When you drop an index, all extents of the index segment are returned to the containing tablespace and become available for other objects in the tablespace.

Example:

```
DROP INDEX index_name;
```

You cannot drop an index used to implement an **integrity constraint**.

- You would first need to remove the integrity constraint by altering the table, then drop the index.
- You cannot drop only the index associated with an enabled UNIQUE key or PRIMARY KEY constraint.
- To drop a constraints associated index, you must disable or drop the constraint itself.

Identifying Unused Indexes

Statistics about the usage of an index can be gathered and displayed in **V$OBJECT_USAGE**.

- Start usage monitoring of an index:

```
ALTER INDEX USER350.dept_id_idx
MONITORING USAGE;
```

- Stop usage monitoring of an index:

```
ALTER INDEX USER350.dept_id_idx
NOMONITORING USAGE;
```

- If the information gathered indicates that an index is never used, the index can be dropped.
- In addition, eliminating unused indexes cuts down on overhead that the Oracle server has to do for DML, thus performance is improved.

Each time the **MONITORING USAGE** clause is specified, **V$OBJECT_USAGE** will be reset for the specified index. The previous information is cleared or reset, and a new start time is recorded.

V$OBJECT_USAGE Columns

- **INDEX_NAME**: The index name

- **TABLE_NAME**: The corresponding table
- **MONITORING**: Indicates whether monitoring is ON or OFF
- **USED**: Indicates YES or NO whether index has been used during the monitoring time
- **START_MONITORING**: Time monitoring began on index
- **END_MONITORING**: Time monitoring stopped on index

Guidelines for Creating Indexes

A table can have any number of indexes, but the more indexes, the more overhead that is associated with DML operations:

- A single row insertion or deletion requires updating all indexes on the table.
- An update operation may or may not affect indexes depending on the column updated.

The use of indexes involves a tradeoff - **query performance** tends to **speed up**, but **data manipulation language operations** tend to **slow down**.

- **Query performance** **improves** because the index speeds up row retrieval.
- **DML operations** **slow down** because along with row insertions, deletions, and updates, the indexes associated with a table must also have insertions and deletions completed.
- For very **volatile** tables, minimize the number of indexes used.

When possible, store indexes to a tablespace that **does not** have rollback segments, temporary segments, and user/data tables – DBAs usually create a separate tablespace just used for index segments.

You can minimize fragmentation by using extent sizes that are at least multiples of **5 times the DB_BLOCK_SIZE** for the database.

Create an index when row retrieval involves less than 15% of a large table's rows – retrieval of more rows is generally more efficient with a full table scan.

You may want to use **NOLOGGING** when creating large indexes - you can improve performance by avoiding redo generation.

- Indexes created with NOLOGGING requires a backup as their creation is not archived.
- Example:

```
CREATE BITMAP INDEX USER350.Products_Region_Idx
    ON USER350.Products_Region (RegionId ASC)
    NOLOGGING;
```

Usually index entries are smaller than the rows they index. Data blocks that store index entries tend to store more entries in each block, so the **INITRANS** parameter should be higher on indexes than on their related tables.

Checking Index Information in the Data Dictionary

You can check the **DBA_INDEXES** view (or **ALL_INDEXES** and **USER_INDEXES**) to get information on the INDEX_NAME, TABLESPACE_NAME, INDEX_TYPE, UNIQUENESS, and STATUS.

```
COLUMN tablespace_name FORMAT A15;

COLUMN index_type FORMAT A10;

SELECT Index_Name, Tablespace_Name, Index_Type,
    Uniqueness, Status
    FROM Dba_Indexes

    WHERE OWNER='DBOCK';


INDEX_NAME                     TABLESPACE_NAME
INDEX_TYPE UNIQUENES STATUS

------------------------------ ---------------
---------- --------- --------
```

```
PK_ROOM                        USERS          NORMAL
UNIQUE    VALID

PK_BEDCLASSIFICATION           USERS          NORMAL
UNIQUE    VALID

PK_BED                         USERS          NORMAL
UNIQUE    VALID

PK_PATIENT                     USERS          NORMAL
UNIQUE    VALID

<more rows display>
```

The **STATUS** column indicates if the index is valid. The **INDEX_TYPE** column indicates if the index is a bitmap or normal index.

You can query the **DBA_IND_COLUMNS** view (or **ALL_IND_COLUMNS** and **USER_IND_COLUMNS**) to determine which columns of a table are used for a particular index.

The **DBA_IND_EXPRESSIONS** (or **ALL_IND_EXPRESSIONS** and **USER_IND_EXPRESSIONS**) views describe the expressions of function-based indexes.

The **V$OBJECT_USAGE** view has information on index usage. produced by the **ALTER INDEX … MONITORING USAGE** command.

END OF NOTES