**Oracle database concepts**

**ROWID**

Oracle Database uses a **rowid** to uniquely identify a row. Internally, the rowid is a structure that holds information that the database needs **to access a row**. A rowid is not physically stored in the database, but is inferred from the file and block on which the data is stored.





Row directory describes the location of rows in the data portion of the block. The database can place a row anywhere in the bottom of the block. The row address is recorded in one of the slots of the row directory vector. A rowid points to a specific file, block, and row number. The row number is an index into an entry in the row directory. The **row directory entry contains a pointer** **to** the location of **the row** on the data block. If the database moves a row within a block, then the database updates the row directory entry to modify the pointer. The rowid stays constant.

A single data segment in a database stores the data for one user object. There are different types of segments. Examples of user segments include:

**Table, table partition, table cluster**, LOB or LOB partition**, Index** or **index partition**.

**Hash Cluster**

To find or store a row in a hash cluster, Oracle Database applies the hash function to the cluster key value of the row. **The resulting hash value corresponds to a data block** in the cluster, which the database reads or writes on behalf of the issued statement.

Hashing multiple input values to the same output value is called a **hash collision**. The database links block 100 to a new overflow block, say block 200, and stores the inserted rows in the new block.



An **index** is an optional structure, associated with a table or table cluster, that can sometimes speed data access. By creating an index on one or more columns of a table, you gain the ability in some cases to retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of reducing disk I/O.

A key is a set of columns or expressions on which you can build an index. Although the terms are often used interchangeably, indexes and keys are different. Indexes are structures stored in the database that users manage using SQL statements. Keys are strictly a logical concept.

A **composite index**, also called a concatenated index, is an index on multiple columns in a table.

A B-tree index has two types of blocks: **branch blocks** for searching and **leaf blocks** that store values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

Indexes can be **unique** or **nonunique**. Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns. Nonunique indexes permit duplicates values in the indexed column or columns. Oracle Database does not index table **rows in which** **all key columns are** [**null**](https://docs.oracle.com/database/121/CNCPT/glossary.htm#GUID-8854502F-2B8F-4ABC-98FA-BBFC3695A964), except for bitmap indexes or when the cluster key column value is null.

In an **ascending** index, Oracle Database stores data in ascending order. By specifying the DESC keyword in the CREATE INDEX statement, you can create a **descending** index. In this case, the index stores data on a specified column or columns in descending order.

A **reverse key index** is a type of B-tree index that **physically reverses the bytes of each index key** while keeping the column order. For example, if the index key is 20, and if the two bytes stored for this key in hexadecimal are C1,15 in a standard B-tree index, then a reverse key index stores the bytes as 15,C1. Reversing the key solves the problem of contention for leaf blocks in the right side of a B-tree index.



**Index Scan**

In an index scan, the database retrieves a row by traversing the index, using the indexed column values specified by the statement. If the database scans the index for a value, then it will find this value in n I/Os where n is the height of the B-tree index. This is the basic principle behind Oracle Database indexes.

**Full Index Scan**

In a full index scan, the database reads the entire index in order.

**Fast Full Index Scan**

A fast full index scan is a full index scan in which the **database accesses the data in the index itself** without accessing the table, and the database reads the index blocks in no particular order.

**Index Range Scan**

An index range scan is an **ordered scan of an index** that has the following characteristics:

One or more leading columns of an index are specified in conditions.

**Index Unique Scan**

In contrast to an index range scan, an index unique scan must have either 0 or 1 rowid associated with an index key. The database performs a unique scan when a predicate references all of the columns in a UNIQUE index key using an equality operator. An index unique scan stops processing as soon as it finds the first record because no second record is possible.

**Index Skip Scan**

An index skip scan uses logical subindexes of a composite index. The database "skips" through a single index as if it were searching separate indexes. Skip scanning is beneficial if there are few distinct values in the leading column of a composite index and many distinct values in the nonleading key of the index.

**Index Compression**

To reduce space in indexes, Oracle Database can employ different compression algorithms. Only keys in the leaf blocks of a B-tree index are compressed. Assume that in a table repeated values occur in the first two columns. An index block could have entries as shown in the follow example:

online,0,AAAPvCAAFAAAAFaAAa

online,0,AAAPvCAAFAAAAFaAAg

online,0,AAAPvCAAFAAAAFaAAl

online,3,AAAPvCAAFAAAAFaAAq

online,3,AAAPvCAAFAAAAFaAAt

If the index in the preceding example were created with prefix compression COMPRESS 2, the index would factor out duplicate occurrences of the first two key values:

online,0

AAAPvCAAFAAAAFaAAa

AAAPvCAAFAAAAFaAAg

AAAPvCAAFAAAAFaAAl

online,3

AAAPvCAAFAAAAFaAAq

AAAPvCAAFAAAAFaAAt

In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows. Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then the row with the corresponding rowid contains the key value. A **mapping function converts the bit position to an actual rowid**, so the bitmap index provides the same functionality as a B-tree index although it uses a different internal representation.

The following table illustrates a bitmap index for the gender column of a table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Value | row1 | row2 | row3 | row4 | row5 | row6 | row7 |
| M | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| F | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

**Function-based Index**

You can create indexes on functions and expressions that involve one or more columns in the table being indexed. A function-based index computes the value of a function or expression involving one or more columns and stores it in the index. A function-based index can be either a B-tree or a bitmap index. Function-based indexes are efficient for evaluating statements that contain functions in their WHERE clauses.