|  |
| --- |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Kiryl Bucha](mailto:Kiryl_Bucha@epam.com) | 12-JAN-2012 |  |  |
| 2.0 | Updated in accordance with renewed content | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |

Contents

[1. Overview of Tables 3](#_Toc383287772)

[1.1. Table Storage 3](#_Toc383287773)

[1.1.1. Storage of Null Values 3](#_Toc383287774)

[1.1.2. Rowids of Row Pieces 3](#_Toc383287775)

[1.1.3. Table Compression 4](#_Toc383287776)

[2. Overview of Table Clusters 4](#_Toc383287777)

[2.1. Index clustered tables. 5](#_Toc383287778)

[2.2. Hash clustered tables 5](#_Toc383287779)

[2.3. Sorted hash clustered tables 5](#_Toc383287780)

[3. Overview: Types of Indexes 6](#_Toc383287781)

[4. Source Books and Articles 7](#_Toc383287782)

# Overview of Tables

The table is the basic data structure used in a relational database. A table is a collection of rows. Each row in a table contains one or more columns. As of Oracle9i, you can define external tables. As the name implies, the data for an external table is stored outside the database, typically in a flat file. The external table is read-only; you cannot update the data it contains. The external table is good for loading and unloading data to files from a database, among other purposes.

Oracle Database 11g introduced the ability to create virtual columns for a table. These columns are defined by an expression and, although the results of the expression are not stored, the columns can be accessed by applications at runtime. Oracle Database 12c introduces the invisible column, which is stored and maintained like a regular column but is not accessible by user request or considered by the query optimizer. There are four major types of tables in Oracle, as follows:

1. **Heap organized tables:** These are normal, standard database tables. Data is managed in a heap-like fashion. As data is added, the first free space found in the segment that can fit the data will be used. As data is removed from the table, it allows space to become available for reuse by subsequent INSERTs and UPDATEs. This is the origin of the name “heap” as it refers to this type of table. A heap is a bunch of space, and it is used in a somewhat random fashion. By default, a table is organized as a heap, which means that the database places rows where they fit best rather than in a user-specified order. Thus, a heap-organized table is an unordered collection of rows. As users add rows, the database places the rows in the first available free space in the data segment. Rows are not guaranteed to be retrieved in the order in which they were inserted.
2. **Index organized tables:** These tables are stored in an index structure. This imposes physical order on the rows themselves. Whereas in a heap the data is stuffed wherever it might fit, in index-organized tables (IOTs) the data is stored in sorted order, according to the primary key.
3. **External tables:** The data in these tables are not stored in the database itself; rather, they reside outside of the database in ordinary operating system files. External tables in Oracle9i and above give you the ability to query a file residing outside the database as if it were a normal table inside the database. They are most useful as a means of getting data into the database (they are a very powerful data-loading tool). Furthermore, in Oracle 10g, which introduces an external table unload capability; they provide an easy way to move data between Oracle databases without using database links.
4. **Object tables:** These tables are created based on an object type. They have special attributes not associated with non-object tables, such as a system-generated REF (object identifier) for each row. Object tables are really special cases of heap, index organized, and temporary tables, and they may include nested tables as part of their structure as well.

## Table Storage

The database stores rows in data blocks. Each row of a table containing data for less than 256 columns is contained in one or more row pieces.

If possible, Oracle Database stores each row as one row piece. However, if all of the row data cannot be inserted into a single data block, or if an update to an existing row causes the row to outgrow its data block, then the database stores the row using multiple row pieces.

### Storage of Null Values

A null is the absence of a value in a column. Nulls indicate missing, unknown, or inapplicable data. Nulls are stored in the database if they fall between columns with data values. In these cases, they require 1 byte to store the length of the column (zero). Trailing nulls in a row require no storage because a new row header signals that the remaining columns in the previous row are null. For example, if the last three columns of a table are null, then no data is stored for these columns.

### Rowids of Row Pieces

A rowid is effectively a 10-byte physical address of a row. Every row stored in the database has an address. Oracle Database uses a ROWID data type to store the address (rowid) of every row in the database. Rowids fall into the following categories:

* Physical rowids store the addresses of rows in heap-organized tables, table clusters, and table and index partitions.
* Logical rowids store the addresses of rows in index-organized tables.

Every row in a heap-organized table has a rowid unique to a table that corresponds to the physical address of a row piece. For table clusters, rows in different tables that are in the same data block can have the same rowid.

Oracle Database uses rowids internally for the construction of indexes. For example, each key in a B-tree index is associated with a rowid that points to the address of the associated row for fast access. Physical rowids provide the fastest possible access to a table row, enabling the database to retrieve a row in as little as a single I/O.

### Table Compression

The database can use table compression to reduce the amount of storage required for the table. Compression saves disk space, reduces memory use in the database buffer cache, and in some cases, speeds query execution. Table compression is transparent to database applications.

Dictionary-based table compression provides good compression ratios for heap-organized tables. Oracle Database supports the following types of dictionary-based table compression:

#### Basic Table Compression and Advanced Row Compression

* **Basic Table Compression.** This type of compression is intended for bulk load operations. The database does not compress data modified using conventional DML. You must use direct-path INSERT operations, ALTER TABLE . . . MOVE operations, or online table redefinition to achieve basic table compression.
* **Advanced row compression.** This type of compression is intended for OLTP applications and compresses data manipulated by any SQL operation.

For the preceding types of compression, the database stores compressed rows in row-major format. All columns of one row are stored together, followed by all columns of the next row, and so on. The database replaces duplicate values with a short reference to a symbol table stored at the beginning of the block. Thus, information needed to recreate the uncompressed data is stored in the data block itself.

#### Hybrid Columnar Compression

With Hybrid Columnar Compression, the database stores the same column for a group of rows together. The data block does not store data in row-major format, but uses a combination of both row and columnar methods.

Storing column data together, with the same data type and similar characteristics, dramatically increases the storage savings achieved from compression. The database compresses data manipulated by any SQL operation, although compression levels are higher for direct path loads. Database operations work transparently against compressed objects, so no application changes are required.

If your underlying storage supports Hybrid Columnar Compression, then you can specify the following compression types, depending on your requirements:

* **Warehouse compression.** This type of compression is optimized to save storage space, and is intended for data warehouse applications.
* **Online archival compression.** This type of compression is optimized for maximum compression levels, and is intended for historical data and data that does not change.

# Overview of Table Clusters

A table cluster is a group of tables that share common columns and store related data in the same blocks. When tables are clustered, a single data block can contain rows from multiple tables. For example, a block can store rows from both the employees and departments tables rather than from only a single table.

The cluster key is the column or columns that the clustered tables have in common. For example, the employees and departments tables share the department\_id column. You specify the cluster key when creating the table cluster and when creating every table added to the table cluster.

The cluster key value is the value of the cluster key columns for a particular set of rows. All data that contains the same cluster key value, such as department\_id=20, is physically stored together. Each cluster key value is stored only once in the cluster and the cluster index, no matter how many rows of different tables contain the value.

For an analogy, suppose an HR manager has two book cases: one with boxes of employees folders and the other with boxes of departments folders. Users often ask for the folders for all employees in a particular department. To make retrieval easier, the manager rearranges all the boxes in a single book case. She divides the boxes by department ID. Thus, all folders for employees in department 20 and the folder for department 20 itself are in one box; the folders for employees in department 100 and the folder for department 100 are in another box, and so on.

You can consider clustering tables when they are primarily queried (but not modified) and records from the tables are frequently queried together or joined. Because table clusters store related rows of different tables in the same data blocks, properly used table clusters offer the following benefits over nonclustered tables:

* Disk I/O is reduced for joins of clustered tables.
* Access time improves for joins of clustered tables.
* Less storage is required to store related table and index data because the cluster key value is not stored repeatedly for each row.

Typically, clustering tables is not appropriate in the following situations:

* The tables are frequently updated.
* The tables frequently require a full table scan.
* The tables require truncating.

There are three types of table clusters.

## Index clustered tables.

Clusters are groups of one or more tables, physically stored on the same database blocks, with all rows that share a common cluster key value being stored physically near each other. Two goals are achieved in this structure. First, many tables may be stored physically joined together. Normally, you would expect data from only one table to be found on a database block, but with clustered tables, data from many tables may be stored on the same block. Second, all data that contains the same cluster key value, such as DEPTNO = 10, will be physically stored together. The data is clustered around the cluster key value. A cluster key is built using a B\*Tree index. The cluster index is separately managed, just like an index on a nonclustered table, and can exist in a separate tablespace from the table cluster.

## Hash clustered tables

These tables are similar to index clustered tables, but instead of using a B\*Tree index to locate the data by cluster key, the hash cluster hashes the key to the cluster to arrive at the database block the data should be on. In a hash cluster, the data is the index (metaphorically speaking). These tables are appropriate for data that is read frequently via an equality comparison on the key. With an indexed table or indexed cluster, Oracle Database locates table rows using key values stored in a separate index. To find or store a row in an indexed table or table cluster, the database must perform at least two I/Os: one or more I/Os to find or store the key value in the index; another I/O to read or write the row in the table or table 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 is an optional way of storing table data to improve the performance of data retrieval.

## Sorted hash clustered tables

This table type is new in Oracle 10g and combines some aspects of a hash-clustered table with those of an IOT. The concept is as follows: you have some key value that rows will be hashed by (say, CUSTOMER\_ID),and then a series of records related to that key that arrive in sorted order(timestamp-based records) and are processed in that sorted order. For example, a customer places orders in your order entry system, and these orders are retrieved and processed in a first in, first out (FIFO) manner. In such a system, a sorted hash cluster may be the right data structure for you.

# Overview: Types of Indexes

The Oracle Database server automatically modifies the values in the index when the values in the corresponding columns are modified. Because the index contains less data than the complete row in the table and because indexes are stored in a special structure that makes them faster to read, it takes fewer I/O operations to retrieve the data in them. Selecting rows based on an index value can be faster than selecting rows based on values in the table rows. In addition, most indexes are stored in sorted order (either ascending or descending, depending on the declaration made when you created the index). Because of this storage scheme, selecting rows based on a range of values or returning rows in sorted order is much faster when the range or sort order is contained in the presorted indexes.

In addition to the data for an index, an index entry stores the ROWID for its associated row. The ROWID is the fastest way to retrieve any row in a database, so the subsequent retrieval of a database row is performed in the most optimal way.

An index can be either unique (which means that no two rows in the table or view can have the same index value) or nonunique. If the column or columns on which an index is based contain NULL values, the row isn’t included in the index.

An index in Oracle refers to the physical structure used within the database. A key is a term for a logical entity, typically the value stored within the index. In most places in the Oracle documentation, the two terms are used interchangeably, with the notable exception of the foreign key constraint.

**B\*Tree indexes:** They are, by far, the most common indexes in use in Oracle and most other databases. Similar in construct to a binary tree, B\*Tree indexes provide fast access, by key, to an individual row or range of rows, normally requiring few reads to find the correct row. It is important to note, however, that the “B” in “B\*Tree” does not stand for binary but rather for balanced. The B\*Tree index has several subtypes:

* *Index organized tables*: These are tables stored in a B\*Tree structure. Whereas rows of data in a heap table are stored in an unorganized fashion (data goes wherever there is available space), data in an IOT is stored and sorted by primary key. IOTs behave just like “regular” tables as far as your application is concerned; you use SQL to access them as normal. IOTs are especially useful for information retrieval, spatial, and OLAP applications.
* *B\*Tree cluster indexes:* These are a slight variation of conventional B\*Tree indexes. They are used to index the cluster. Rather than having a key that points to a row, as for a conventional B\*Tree, a B\*Tree cluster has a cluster key that points to the block that contains the rows related to that cluster key.
* *Descending indexes:* Descending indexes allow for data to be sorted from big-to-small (descending) instead of small-to-big (ascending) in the index structure.
* *Reverse key indexes:* These are B\*Tree indexes whereby the bytes in the key are reversed. Reverse key indexes can be used to obtain a more even distribution of index entries throughout an index that is populated with increasing values. For example, if I am using a sequence to generate primary key, the sequence will generate values like 987500, 987501, 987502, and so on. These sequence values are monotonic, so if I were using a conventional B\*Tree index, they would all tend to go the same right-hand side block, thus increasing contention for that block. With a reverse key index, Oracle will logically index 205789, 105789, 005789, and so on instead. Oracle will reverse the bytes of the data to be stored before placing them in the index, so values that would have been next to each other in the index before the byte reversal will instead be far apart. This reversing of the bytes spreads out the inserts into the index over many blocks.

**Function-based indexes:** These are B\*Tree or bitmap indexes that store the computed result of a function on a row’s column(s), not the column data itself. You can consider them an index on a virtual (or derived) column—in other words, a column that is not physically stored in the table. These may be used to speed up queries of the form SELECT \* FROM T WHERE FUNCTION(DATABASE\_COLUMN) =SOME\_VALUE, since the value FUNCTION(DATABASE\_COLUMN) has already been computed and stored in the index.

**Bitmap indexes:** Normally in a B\*Tree, there is a one-to-one relationship between an index entry and a row: an index entry points to a row. With bitmap indexes, a single index entry uses a bitmap to point to many rows simultaneously. They are appropriate for highly repetitive data (data with few distinct values relative to the total number of rows in the table) that is mostly read-only. Consider a column that takes on three possible values—Y, N, and NULL—in a table of 1 million rows. This might be a good candidate for a bitmap index if, for example, you need to frequently count how many rows has a value of Y. That is not to say that a bitmap index on a column with 1,000 distinct values in that same table would not be valid—it certainly can be. Bitmap indexes should never be considered in an OLTP database for concurrency-related issues.

**Bitmap join indexes:** These provide a means of denormalizing data in an index structure, instead of in a table. For example, consider the simple EMP and DEPT tables. Someone might ask the question, “How many people work in departments located in the city of Boston?” EMP has a foreign key to DEPT, and in order to count the employees in departments with a LOC value of Boston, we would normally have to join the tables to get the LOC column joined to the EMP records to answer this question. Using a bitmap join index, we can instead index the LOC column against the EMP table. The same caveat in regard to OLTP systems applies to a bitmap join index as a regular bitmap index.

**Application domain indexes:** These are indexes you build and store yourself, either in Oracle or perhaps even outside of Oracle. You tell the optimizer how selective your index is and how costly it is to execute, and the optimizer will decide whether to use your index based on that information. The Oracle text index is an example of an application domain index; it is built using the same tools you may use to build your own index. It should be noted that the index created here need not use a traditional index structure. The Oracle text index, for example, uses a setoff tables to implement its concept of an index.

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.
3. Greenwald R., & Stackowiak R., & Stern J. Oracle Essentials, Fifth Edition. Sebastopol: O’Reilly Media, Inc., 2013.