**Oracle database concepts**

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.



A **hash cluster** is like an indexed cluster, except the index key is replaced with a hash function. No separate cluster index exists. In a hash cluster, the data is the index.

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.



**Partitions**

In an Oracle database, partitioning enables you to decompose very large tables and indexes into smaller and more manageable pieces called partitions. Each partition is an independent object with its own name and optionally its own storage characteristics. A partitioned object has pieces that can be managed either collectively or individually. DDL statements can manipulate partitions rather than entire tables or indexes.

Each partition of a table or index must have the same logical attributes, such as column names, data types, and constraints. For example, all partitions in a table share the same column and constraint definitions. However, each partition can have separate physical attributes, such as the tablespace to which it belongs.

The partition key is a set of one or more columns that determines the partition in which each row in a partitioned table should go. Each row is unambiguously assigned to a single partition.

Oracle Partitioning offers several partitioning strategies that control how the database places data into partitions. The basic strategies are range, list, and hash partitioning.

A single-level partitioning uses only one method of data distribution, for example, only list partitioning or only range partitioning.

In composite partitioning, a table is partitioned by one data distribution method and then each partition is further divided into subpartitions using a second data distribution method.

In **range partitioning**, the database maps rows to partitions based on ranges of values of the partitioning key. The range partition key value determines the non-inclusive high bound for a specified partition.

In **list partitioning**, the database uses a list of discrete values as the partition key for each partition. You can use list partitioning to control how individual rows map to specific partitions. By using lists, you can group and organize related sets of data when the key used to identify them is not conveniently ordered.

In **hash partitioning**, the database maps rows to partitions based on a hashing algorithm that the database applies to the user-specified partitioning key.

A partitioned table is made up of one or more **table partition segments** If you create a partitioned table named hash\_products, then no table [segment](https://docs.oracle.com/database/121/CNCPT/glossary.htm#GUID-EC12AA68-8C89-43B3-B1F9-3AABF7CAEB9F) is allocated for this table. Instead, the database stores data for each table partition in its own partition segment. Each table partition segment contains a portion of the table data.



Range partitions based on TIME\_ID column.

A **partitioned index** is an index that, like a partitioned table, has been divided into smaller and more manageable pieces.

Global indexes are partitioned independently of the table on which they are created, whereas local indexes are automatically linked to the partitioning method for a table. Like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability.

In a local partitioned index, the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as its table.

Local partitioned indexes are either prefixed or nonprefixed.

In the prefixed case, the partition keys are on the leading edge of the index definition.

**Index-Organized Table**

An index-organized table is a table stored in a variation of a B-tree index structure. In a heap-organized table, rows are inserted where they fit. In an index-organized table, rows are stored in an index defined on the primary key for the table. Each index entry in the B-tree also stores the non-key column values.

An index-organized table stores all data in the same structure and does not need to store the rowid.

When creating an index-organized table, you can specify a separate segment as a row overflow area. In index-organized tables, B-tree index entries can be large because they contain an entire row, so a separate segment to contain the entries is useful. In contrast, B-tree entries are usually small because they consist of the key and rowid.

If a row overflow area is specified, then the database can divide a row in an index-organized table into the following parts:

The index entry

This part contains column values for all the primary key columns, a physical rowid that points to the overflow part of the row, and optionally a few of the non-key columns. This part is stored in the index segment.

The overflow part

This part contains column values for the remaining non-key columns. This part is stored in the overflow storage area segment.

A secondary index is an index on an index-organized table. The secondary index is an independent schema object and is stored separately from the index-organized table. **Rows in index leaf blocks can move** within or between blocks **because of insertions**. Because rows in index-organized tables do not have permanent physical addresses, the database uses logical rowids based on primary key.

Secondary indexes use the logical rowids to locate table rows. A logical rowid includes a physical guess, which is the physical rowid of the index entry when it was first made.

With inaccurate physical guesses, access involves a secondary index scan and an I/O to fetch the wrong data block (as indicated by the guess), followed by an index unique scan of the index organized table by primary key value.



**Bitmap Join Index**

A bitmap join index is a bitmap index for the join of two or more tables. For each value in a table column, the index stores the rowid of the corresponding row in the indexed table.

An example when a bitmap join index would be useful:

SELECT COUNT(\*) FROM employees, jobs

WHERE employees.job\_id = jobs.job\_id

AND jobs.job\_title = '**Accountant**';



jobs.job\_title employees.rowid

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

Accountant AAAQNKAAFAAAABSAAM

Accountant AAAQNKAAFAAAABSAAJ

Accountant AAAQNKAAFAAAABSAAK

Accounting Manager AAAQNKAAFAAAABTAAH

Administration Assistant AAAQNKAAFAAAABTAAC

Administration Vice President AAAQNKAAFAAAABSAAC

Administration Vice President AAAQNKAAFAAAABSAAB

**CREATE BITMAP INDEX** employees\_bm\_idx **ON employees (jobs.job\_title)**

**FROM** employees, jobs **WHERE** employees.job\_id=jobs.job\_id;

You can find information about Bitmap Join indexes in the data dictionary:

DBA\_INDEXES.JOIN\_INDEX 🡪 ’YES’