All about Indexes in Oracle

What is an Index?

An Oracle index is an optional structure associated with a table that provides direct access to rows. It can be created on one or more columns of a table. By creating an index, you help the database locate and retrieve data faster, significantly speeding up query performance.

How does an index work?

When you create an index on a column, Oracle creates a data structure that stores the values in that column in a sorted order. This allows the database to quickly locate the rows that match a query, much like how you'd use a table of contents to jump to the right section in a book without reading every page.

An index's selectivity is optimal if few rows have the same value.

What are the different states of Oracle Indexes

VALID	Index is usable and can be used by optimizer.
UNUSABLE	Index is not usable until it is rebuilt. This may occur due to structural changes to be underlying table or other maintanance actions.
INACTIVE	Index is temporarily Inactive, typically during maintanance operations like rebuilding or altering.
N/A	Indexes that is not applicable, often for indexes that are being created or modified.

Rebuild indexes in Oracle

Rebuilding an index means dropping the existing structure and creating a new one. This removes fragmentation and reorganizes the data, thus reducing disk I/O and improving query response time. Also, rebuilding helps reclaim unused space in the database, improving storage utilization.

1. If the index has height greater than four, rebuild the index.
2. The deleted leaf rows should be less than 20%.

SELECT name, height,lf_rows,lf_blks,del_lf_rows FROM INDEX_STATS;

command to rebuild index online: Alter index <index name> rebuild online;

Data dictionary views on Indexes

- DBA_INDEXES
- DBA_IND_COLUMNS
- DBA_IND_STATISTICS

How to Write Statements that Avoid Using Indexes

- Use NO_INDEX optimizer hint.

  SELECT /+ NO_INDEX(employees emp_empid) */ employee_id FROM employees WHERE employee_id > 200;*

- Use FULL hint to choose Full Table Scan instead of an Index Scan.

  SELECT /+ FULL(e) */ employee_id, last_name FROM employees e WHERE last_name LIKE :b1;*

- You can use the INDEX, INDEX_COMBINE, or AND_EQUAL hints to force the optimizer to use one index or a set of listed indexes instead of another.

  SELECT /+ INDEX_FFS(e emp_name_ix) */ first_name FROM employees e;*

  SELECT /+ INDEX_COMBINE(e emp_manager_ix emp_department_ix) */ * FROM employees e WHERE manager_id = 108 OR department_id = 110;*

Gather statistics for Indexes

- Index statistics are gathered using the ANALYZE INDEX or dbms_stats statement.
- Available options are COMPUTE/ESTIMATE STATISTICS or VALIDATE STRUCTURE.
- The statistics contain the number of leaf rows & blocks (LF_ROWS, LF_BLKS), number of branch rows & blocks (BR_ROWS, BR_BLKS), number of deleted leaf rows (DEL_LF_ROWS), used space (USED_SPACE), number of distinct keys (DISTINCT_KEYS), etc.
- These statistics can be used to determine if the index should be rebuilt or not

Invisible Indexes in Oracle

Invisible indexes are maintained like any other index, but they are ignored by the optimizer unless the OPTIMIZER_USE_INVISIBLE_INDEXES parameter is set to TRUE at the instance or session level. Indexes can be created as invisible by using the INVISIBLE keyword, and their visibility can be toggled using the ALTER INDEX command.

create index index_name on table_name(column_name) invisible;

alter index index_name invisible;

alter index index_name visible;

Invisible indexes can be useful for processes with specific indexing needs, where the presence of the indexes may adversely affect other functional areas. They are also useful for testing the impact of adding or dropping an index.

Automatic Indexing feature in 19c

Automatic Indexing is a new feature in Oracle 19c which automatically creates, rebuilds, and drops indexes in a database based on the application workload.

The index management task is now dynamically performed by the database itself via a task which executes in the background every 15 minutes.

Automatic indexing task analyzes the current workload and identifies candidates for indexes.

It then creates the indexes as invisible indexes and evaluates the identified candidate SQL statements. If the performance is improved, then the indexes are made visible and can be then used by the application. If there is no improvement in the performance, then the indexes are marked as unusable and dropped after a predefined interval.

The automatic indexing feature is managed via the DBMS_AUTO_INDEX package.

Note that this feature is currently available only on the Oracle Engineered Systems platform.

What is Selectivity

- It represents the fraction of rows filtered by an operation so you can say it is a measure of uniqueness. Its value is between 0 and 1.

```
selectivity = No.of rows returning from the query / total no of rows
```

High selective – If a SQL return small number of unique rows. (Value=0)
Least selective – If a SQL return all or large number of rows. (Value=1)

What is Cardinality

- The number of rows returned by an operation is the cardinality. The relationship between selectivity and cardinality is below:

```
cardinality = selectivity × number of input rows
```

High cardinality – large number of distinct values
Low cardinality – a smaller number of distinct values

Example:

EMP_NO	FIRST_NAME	LAST_NAME
1	JIM	JERRY
2	TOM	MATHEW
3	HILL	JOHN
4	STEVE	BLESS
5	SAM	MARK
6	LOUIS	LUKE
7	BIJI	BENS
8	STEPHEN	SAPH
9	RAPH	JERRY
10	ROSH	LORA

scenario 1: We want to see the maximum of EMP_NO from the EMP table.

select max(emp_no) from EMP;

selectivity: 10/10 =1 Cardinality = 1*10 = 10

scenario 2: We want to see the maximum of EMP_NO from the EMP table whose last name is Jerry.

select max(emp_no) from EMP where last_name='JERRY';

selectivity = 2/10 =0.2 Cardinality = 0.2*10 = 2

Low cardinality

EMP_NO	FIRST_NAME	LAST_NAME	GENDER
1	JIM	JERRY	Male
2	TOM	MATHEW	Male
3	HILL	JOHN	Male
4	STEVE	BLESS	Female
5	SAM	MARK	Male
6	LOUIS	LUKE	Male
7	BIJI	BENS	Male
8	STEPHEN	SAPH	Female
9	RAPH	JERRY	Male
10	ROSH	LORA	Female

The GENDER column has a low cardinality, because it only has two distinct values, but they are all appearing lots of times.

High Cardinality

EMP_NO	FIRST_NAME	LAST_NAME	EMAIL
1	JIM	JERRY	jimjerry@gmail.com
2	TOM	MATHEW	tommathew@gmail.com
3	HILL	JOHN	hilljohn@gmail.com
4	STEVE	BLESS	stevebless@gmail.com
5	SAM	MARK	sammark@gmail.com
6	LOUIS	LUKE	louisluke@gmail.com
7	BIJI	BENS	bijibens@gmail.com
8	STEPHEN	SAPH	stephensaph@gmail.com
9	RAPH	JERRY	raphjerry@gmail.com
10	ROSH	LORA	roshlora@gmail.com

The EMAIL column has a high cardinality, because it has large number of distinct values.

In general, columns with high cardinality tend to be more useful for identifying specific rows in a table, because the values in these columns are more unique. Low cardinality columns, on the

other hand, are less useful for identifying specific rows, because the values in these columns are not as unique.

Index clustering factor

Oracle does I/O by blocks. Therefore, the optimizer's decision to use full table scans is influenced by the percentage of blocks accessed, not rows. This is called the index clustering factor. If blocks contain single rows, then rows accessed, and blocks accessed are the same.

A lower clustering factor indicates that the individual rows are concentrated within fewer blocks in the table. Conversely, a high clustering factor indicates that the individual rows are scattered more randomly across blocks in the table. Therefore, a high clustering factor means that it costs more to use a range scan to fetch rows by ROWID, because more blocks in the table need to be visited to return the data.

Types of Indexes

- **B-Tree Index:** B-Tree indexes are the most used type of index in Oracle databases. They are organized in a balanced tree structure and are used to quickly locate data based on the indexed column.

  Use Case: Suitable for columns with high cardinality (many unique values), such as primary keys or unique keys.

  The command to create a B-Tree index is:

```
CREATE INDEX index_name ON table_name (column_name);
```

- **Bitmap Index:** Bitmap indexes are used to index low-cardinality columns, such as Boolean columns. They are stored as a bitmap, with each bit representing a row in the table.

  The command to create a Bitmap index is:

```
CREATE BITMAP INDEX index_name ON table_name (column_name);
```

- **Unique Index:** Ensures that all values in the indexed column(s) are unique. Automatically created with primary key and unique constraints. Enforcing uniqueness for columns like email addresses or user IDs.

```
CREATE UNIQUE INDEX idx_employee_email ON employees (email);
```

- **Composite Index:** An index on multiple columns. It can be either a B-Tree or Bitmap index. Optimizes queries that filter on multiple columns together, such as a combination of first name and last name.

```
CREATE INDEX idx_employee_fullname ON employees (first_name, last_name);
```

- **Function-based Index:** Function-based indexes are used to index the result of a function applied to one or more columns. This can be useful for improving performance when querying on a specific function of a column. The command to create a function-based index is:

```
CREATE INDEX index_name ON table_name (function_name(column_name));
```

- **Reverse Key Index:** Reverse Key indexes are used to improve the performance of queries that use the LIKE operator on columns that have a high number of trailing spaces. The command to create a Reverse Key index is:

```
CREATE INDEX index_name ON table_name (column_name) REVERSE;
```

- **Domain Index:** Custom index designed for complex data types like spatial, text, or XML data. Requires using Oracle Data Cartridge.

```
CREATE INDEX idx_location ON locations (location) INDEXTYPE IS
MDSYS.SPATIAL_INDEX;
```

- **Clustered Index:** Physically reorders the data in the table to match the index. Oracle does not support clustered indexes natively but achieves similar functionality through Index Organized Tables (IOT).

```
CREATE TABLE employees ( employee_id NUMBER PRIMARY KEY, name VARCHAR2(50),
... ) ORGANIZATION INDEX;
```

Best practices for Index

- Consider indexing keys that are used frequently in WHERE clauses.
- Consider indexing keys that are used frequently to join tables in SQL statements. For more information on optimizing joins, see the section "Using Hash Clusters for Performance".
- Choose index keys that have high selectivity. The selectivity of an index is the percentage of rows in a table having the same value for the indexed key. An index's selectivity is optimal if few rows have the same value. Note: Oracle automatically creates indexes, or uses existing indexes, on the keys and expressions of unique and primary keys that you define with integrity constraints. Indexing low selectivity columns can be helpful if the data distribution is skewed so that one or two values occur much less often than other values.
- Do not use standard B-tree indexes on keys or expressions with few distinct values. Such keys or expressions usually have poor selectivity and therefore do not optimize performance unless the frequently selected key values appear less frequently than the other key values. You can use

bitmap indexes effectively in such cases, unless the index is modified frequently, as in a high concurrency OLTP application.

- Do not index columns that are modified frequently. UPDATE statements that modify indexed columns and INSERT and DELETE statements that modify indexed tables take longer than if there were no index. Such SQL statements must modify data in indexes as well as data in tables. They also generate additional undo and redo.
- Do not index keys that appear only in WHERE clauses with functions or operators. A WHERE clause that uses a function, other than MIN or MAX, or an operator with an indexed key does not make available the access path that uses the index except with function-based indexes.
- Consider indexing foreign keys of referential integrity constraints in cases in which many concurrent INSERT, UPDATE, and DELETE statements access the parent and child tables. Such an index allows UPDATEs and DELETEs on the parent table without share locking the child table.
- When choosing to index a key, consider whether the performance gain for queries is worth the performance loss for INSERTs, UPDATEs, and DELETEs and the use of the space required to store the index. You might want to experiment by comparing the processing times of the SQL statements with and without indexes. You can measure processing time with the SQL trace facility.

Drawbacks of using indexes

- Indexes consume disk space and memory, so it's important to monitor the space usage and make sure that it is not becoming a bottleneck.
- Indexes also require additional maintenance, as they need to be rebuilt or reorganized periodically to maintain their performance.
- Indexes also add some overhead to DML operations(Insert, update, delete) as the indexes need to be updated as well.

TYPES OF INDEX SCAN

1. INDEX UNIQUE SCAN

   The database performs a unique scan when a predicate references all the columns in a UNIQUE index key using an equality operator.

   Example: EMP_ID column is PRIMARY KEY and has an INDEX.

EMP_ID	FIRST_NAME	LAST_NAME	EMAIL	GENDER
1	JIM	JERRY	jimjerry@gmail.com	male
2	TOM	MATHEW	jimjerry@gmail.com	male
3	HILL	JOHN	hilljohn@gmail.com	male
4	STEVE	BLESS	stevebless@gmail.com	male
5	SAM	MARK	sammark@gmail.com	male
6	LOUIS	LUKE	louisluke@gmail.com	female
7	BIJI	BENS	bijibens@gmail.com	female
9	RAPH	JERRY	raphjerry@gmail.com	male
10	ROSH	LORA	roshlora@gmail.com	female
8	STEPHEN	SAPH	stephensa@gmail.com	male

select * from test.emp where emp_id=10;

```
06:38:07 SQL> select * from test.emp where emp_id=10;

    EMP_ID|FIRST_NAME          |LAST_NAME           |EMAIL               |GENDER
----------|--------------------|--------------------|--------------------|--------------------
        10|ROSH                |LORA                |roshlora@gmail.com  |female

1 row selected.

Elapsed: 00:00:00.00

Execution Plan
----------------------------------------------------------
Plan hash value: 2384765270

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    61 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP           |     1 |    61 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMP_ID"=10)
```

2. INDEX RANGE SCAN

   An index range scan is a common operation for accessing selective data.

   FIRST_NAME column has an INDEX created.

   select * from TEST.EMP where first_name='HILL';

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1368747079

-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |     1 |    61 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP               |     1 |    61 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | EMP_FIRST_NAME_IND |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("FIRST_NAME"='HILL')
```

3. INDEX FULL SCAN

   In INDEX FULL SCAN, the table will NOT be referenced at all and instead all information will be fetched from the index itself using single block read.

   Prerequisites for Index Full Scan

   - All the columns that are used in the SELECT must be part of index, if any column is missing, it will NOT use INDEX FULL SCAN.

- An ORDER BY clause that meets the following requirements is present in the query.
- The query requires a sort merge join.

4. INDEX FAST FULL SCAN

   In INDEX FAST FULL SCAN the table will NOT be referenced at all and instead all information will be fetched from the index itself using Multi-block read (128 times faster than sequential read). This result in performance upgradation.

   Prerequisites for Index Full Scan

   - All the columns that are used in the SELECT must be part of index, if any column is missing, it will NOT use INDEX FAST FULL SCAN.
   - There should not be any ORDER BY clause in the query. The optimizer should randomly perform multi-block read.

5. INDEX SKIP SCAN

   An index skip scan occurs when the initial column of a composite index is “skipped” or not specified in the query.

   Prerequisites for Index skip scan

   - The leading column of a composite index is not specified in the query predicate.
   - Few distinct values exist in the leading column of the composite index, but many distinct values exist in the nonleading key of the index.