12

Using Indexes in Database Applications

Indexes are optional structures, associated with tables and clusters, which allow SQL queries to execute more quickly. Just as the index in this guide helps you locate information faster than if there were no index, an Oracle Database index provides a faster access path to table data. You can use indexes without rewriting any queries. Your results are the same, but you see them more quickly.

> **See Also:**
>
> - *Oracle Database Concepts* for more information about indexes and index-organized tables
> - *Oracle Database Administrator's Guide* for more information about managing indexes
> - *Oracle Database SQL Tuning Guide* for more information about how indexes and clusters can enhance or degrade performance

Topics:

12.1 Guidelines for Managing Indexes

The summary of guidelines for managing the indexes are as follows:

- Create indexes after inserting table data
- Index the correct tables and columns
- Order index columns for performance
- Limit the number of indexes for each table
- Drop indexes that are no longer needed
- Understand deferred segment creation
- Estimate index size and set storage parameters
- Specify the tablespace for each index
- Consider parallelizing index creation
- Consider creating indexes with `NOLOGGING`
- Understand when to use unusable or invisible indexes

- Consider costs and benefits of coalescing or rebuilding indexes
- Consider cost before disabling or dropping constraints

> **See Also:**
>
> *Oracle Database Administrator's Guide*

12.2 Managing Indexes

Oracle Database Administrator's Guide has this information about managing indexes:

- Creating indexes
- Altering indexes
- Monitoring space use of indexes
- Dropping indexes
- Data dictionary views that display information about indexes

> **See Also:**
>
> Creating Indexes for Use with Constraints

12.3 When to Use Domain Indexes

A **domain index** (also called an **application domain index**) is a customized index specific to an application that was implemented using a data cartridge (for example, a search engine or geographic information system).

> **See Also:**
>
> - *Oracle Database Data Cartridge Developer's Guide* for conceptual background to help you decide when to build domain indexes
> - *Oracle Database Concepts* for information about domain indexes

12.4 When to Use Function-Based Indexes

A **function-based index** computes the value of an expression that involves one or more columns and stores it in the index. The index expression can be an arithmetic expression or an expression that contains a SQL function, PL/SQL function, package function, or C callout. Function-based indexes also support linguistic sorts based on collation keys, efficient linguistic collation of SQL statements, and case-insensitive sorts.

A function-based index improves the performance of queries that use the index expression (especially if the expression computation is intensive). However:

- The database must also evaluate the index expression to process statements that do not use it.
- Function-based indexes on columns that are frequently modified are expensive for the database to maintain.

The optimizer can use function-based indexes only for cost-based optimization, while it can use indexes on columns for both cost-based and rule-based optimization.

> **Note:**
>
> - A function-based index cannot contain the value `NULL`. Therefore, either ensure that no column involved in the index expression can contain `NULL` or use the `NVL` function in the index expression to substitute another value for `NULL`.
> - Oracle Database treats descending indexes as if they were function-based indexes.

Topics:

- Advantages of Function-Based Indexes
- Disadvantages of Function-Based Indexes
- Example: Function-Based Index for Precomputing Arithmetic Expression
- Example: Function-Based Indexes on Object Column
- Example: Function-Based Index for Faster Case-Insensitive Searches
- Example: Function-Based Index for Language-Dependent Sorting

> **See Also:**
>
> - *Oracle Database Concepts* for additional conceptual information about function-based indexes
> - *Oracle Database Administrator's Guide* for information about creating function-based indexes
> - *Oracle Database Globalization Support Guide* for information about function-based linguistic indexes
> - *Oracle Database Concepts* for more information about how the optimizer uses function-based indexes
> - *Oracle Database SQL Tuning Guide* for information about using function-based indexes for performance
> - *Oracle Database SQL Language Reference* for information about `NVL`
> - *Oracle Database SQL Language Reference* for more information about creating index

12.4.1 Advantages of Function-Based Indexes

A function-based index has these advantages:

- A function-based index increases the number of situations where the database can perform an index range scan instead of a full index scan.

  An index range scan typically has a fast response time when the `WHERE` clause selects fewer than 15% of the rows of a large table. The optimizer can more accurately estimate how many rows an expression selects if the expression is materialized in a function-based index.

  Oracle Database represents the index expression as a virtual column, on which the `ANALYZE` statement can build a histogram.

- A function-based index precomputes and stores the value of an expression.

  Queries can get the value of the expression from the index instead of computing it. The more queries that need the value and the more intensive computation the index expression gets, the index improves application performance.

- You can create a function-based index on an object column or `REF` column.

  The index expression can be the invocation of a method that returns an object type.

- A function-based index lets you perform more powerful sorts.

  The index expression can invoke the SQL functions `UPPER` and `LOWER` for case-insensitive sorts (as in Example 12-3) and the SQL function `NLSSORT` for linguistic-based sorts.

> **See Also:**
>
> - *Oracle Database Concepts* for more information about index-range scan and index scan
> - *Oracle Database SQL Language Reference* for more information about `ANALYZE` statement
> - *Oracle Database Object-Relational Developer's Guide* for more information about function-based index
> - Example: Function-Based Indexes on Object Column and *Oracle Database SQL Language Reference* for examples related to function-based indexes
> - Example: Function-Based Index for Precomputing Arithmetic Expression
> - Example: Function-Based Index for Language-Dependent Sorting for example related to `NLSSORT` SQL function

12.4.2 Disadvantages of Function-Based Indexes

A function-based index has these disadvantages:

- The optimizer can use a function-based index only for cost-based optimization, not for rule-based optimization.

  The cost-based optimizer uses statistics stored in the dictionary. To gather statistics for a function-based index, invoke either `DBMS_STATS.GATHER_TABLE_STATS` or `DBMS_STATS.GATHER_SCHEMA_STATS` .

- The database does not use a function-based index until you analyze the index itself and the table on which it is defined.

To analyze the index and the table on which the index is defined, invoke either `DBMS_STATS.GATHER_TABLE_STATS` or `DBMS_STATS.GATHER_SCHEMA_STATS`.

- The database does not use function-based indexes when doing `OR` expansion.
- You must ensure that any schema-level or package-level PL/SQL function that the index expression invokes is deterministic (that is, that the function always return the same result for the same input).

  You must declare the function as `DETERMINISTIC`, but because Oracle Database does not check this assertion, you must ensure that the function really is deterministic.

  If you change the semantics of a `DETERMINISTIC` function and recompile it, then you must manually rebuild any dependent function-based indexes and materialized views. Otherwise, they report results for the prior version of the function.
- If the index expression is a function invocation, then the function return type cannot be constrained.

  Because you cannot constrain the function return type with `NOT NULL`, you must ensure that the query that uses the index cannot fetch `NULL` values. Otherwise, the database performs a full table scan.
- The index expression cannot invoke an aggregate function.
- A bitmapped function-based index cannot be a descending index.
- Index skip scanning is not supported in function-based indexes.
- The data type of the index expression cannot be `VARCHAR2`, `RAW`, `LONGRAW`, or a PL/SQL data type of unknown length.

  That is, you cannot index an expression of unknown length. However, you can index a known-length substring of that expression. For example:

```
CREATE OR REPLACE FUNCTION initials (
  name  IN VARCHAR2
) RETURN VARCHAR2
  DETERMINISTIC
IS
BEGIN
  RETURN('A. J.');
END;
/

/* Invoke SUBSTR both when creating index and when referencing
   function in queries. */

CREATE INDEX func_substr_index ON
EMPLOYEES(SUBSTR(initials(FIRST_NAME),1,10));

SELECT SUBSTR(initials(FIRST_NAME),1,10) FROM EMPLOYEES;
```

See Also:

- *Oracle Database SQL Language Reference* for notes on function-based indexes
- *Oracle Database SQL Language Reference* for restrictions on function-based indexes
- *Oracle Database PL/SQL Language Reference* for information about the `CREATE FUNCTION` statement, including restrictions
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_STATS`, `GATHER_TABLE_STATS`, `DBMS_STATS`, and `GATHER_SCHEMA_STATS`
- *Oracle Database SQL Language Reference* for more information about aggregate functions
- *Oracle Database SQL Language Reference* for more information about function-based index

12.4.3 Example: Function-Based Index for Precomputing Arithmetic Expression

You can create composite indexes to computer arithmetic expressions.

Example 12-1 creates a table with columns `a`, `b`, and `c`; creates an index on the table, and then queries the table. The index is a composite index on three columns: a virtual column that represents the expression `a+b*(c-1)`, column `a`, and column `b`. The query uses the indexed expression in its `WHERE` clause; therefore, it can use an index range scan instead of a full index scan.

Example 12-1 Function-Based Index for Precomputing Arithmetic Expression

Create table on which to create index:

```
DROP TABLE Fbi_tab;
CREATE TABLE Fbi_tab (
  a INTEGER,
  b INTEGER,
  c INTEGER
);
```

Create index:

```
CREATE INDEX Idx ON Fbi_tab (a+b*(c-1), a, b);
```

This query can use an index range scan instead of a full index scan:

```
SELECT a FROM Fbi_tab WHERE a+b*(c-1) < 100;
```

Note:

This example uses composite indexes (indexes on multiple table columns).

> **See Also:**
>
> - *Oracle Database Concepts* for information about fast full index scans
> - *Oracle Database Concepts* for more information about composite indexes

12.4.4 Example: Function-Based Indexes on Object Column

In Example 12-2, assume that the object type `Reg_obj` has been defined, and that it stores information about a city. The example creates a table whose first column has type `Reg_obj`, a deterministic function with a parameter of type `Reg_obj`, and two function-based indexes that invoke the function. The first query uses the first index to quickly find cities further than 1000 miles from the equator. The second query uses the second index (which is composite) to quickly find cities where the temperature delta is less than 20 and the maximum temperature is greater than 75. (The table is not populated for the example, so the queries return no rows.)

Example 12-2 Function-Based Indexes on Object Column

Create table with object column:

```
DROP TABLE Weatherdata_tab;
CREATE TABLE Weatherdata_tab (
  Reg_obj INTEGER,
  Maxtemp INTEGER,
  Mintemp INTEGER
);
```

Create deterministic function with parameter of type `Reg_obj`:

```
CREATE OR REPLACE FUNCTION Distance_from_equator (
  Reg_obj IN INTEGER
) RETURN INTEGER
  DETERMINISTIC
IS
BEGIN
  RETURN(3000);
END;
/
```

Create first function-based index:

```
CREATE INDEX Distance_index
ON Weatherdata_tab (Distance_from_equator (Reg_obj));
```

Use index expression in query:

```
SELECT * FROM Weatherdata_tab
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

Result:

```
no rows selected
```

Create second function-based (and composite) index:

```
CREATE INDEX Compare_index
ON Weatherdata_tab ((Maxtemp - Mintemp) DESC, Maxtemp);
```

Use index expression and indexed column in query:

```
SELECT * FROM Weatherdata_tab
WHERE ((Maxtemp - Mintemp) < '20' AND Maxtemp > '75');
```

Result:

```
no rows selected
```

12.4.5 Example: Function-Based Index for Faster Case-Insensitive Searches

Example 12-3 creates an index that allows faster case-insensitive searches in the `EMPLOYEES` table and then uses the index expression in a query.

Example 12-3 Function-Based Index for Faster Case-Insensitive Searches

Create index:

```
CREATE INDEX emp_lastname ON EMPLOYEES (UPPER(LAST_NAME));
```

Use index expression in query:

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE UPPER(LAST_NAME) LIKE 'J%S_N';
```

Result:

```
FIRST_NAME           LAST_NAME
-------------------- -------------------------
Charles              Johnson

1 row selected.
```

12.4.6 Example: Function-Based Index for Language-Dependent Sorting

You can use the `NLSSORT` API for language-dependent sorting.

Example 12-4 creates a table with one column, `NAME`, and a function-based index to sort that column using the collation sequence `GERMAN`, and then selects all columns of the table, ordering them by `NAME`. Because the query can use the index, the query is faster. (Assume that the query is run in a German session, where `NLS_SORT` is `GERMAN` and `NLS_COMP` is `ANSI`. Otherwise, the query would have to specify the values of these Globalization Support parameters.)

Example 12-4 Function-Based Index for Language-Dependent Sorting

Create table on which to create index:

```
DROP TABLE nls_tab;
CREATE TABLE nls_tab (NAME VARCHAR2(80));
```

Create index:

```
CREATE INDEX nls_index
  ON nls_tab (NLSSORT(NAME, 'NLS_SORT = GERMAN'));
```

Select all table columns, ordered by `NAME`:

```
SELECT * FROM nls_tab
WHERE NAME IS NOT NULL
ORDER BY NAME;
```