Expert Oracle Indexing and Access Paths

CHAPTER 1 Introduction to Oracle Indexes

An index is a combination of a value and an associated location that allows you to quickly locate an object. We probably often use index structures to navigate throughout the day, such as:

 • Browsing a catalog menu to efficiently find a desired item

 • Accessing a bookmark to retrieve a web page

 • Glancing at the calendar to determine a date

Similar to the way you use indexes in everyday life, a database index primarily exists to enable fast access to one or more rows of information stored in a database

In Oracle, an index stores a table column value along with the table row’s physical address ( ROWID ).. Each ROWID contains details on the location of a table row (e.g., the data file number, block number, and row location).

if indexes are so important to database performance, why not place them on all tables and column combinations? The answer is short: indexes are not free. They consume disk space and system resources.

indexes use storage, I/O, CPU, and memory resources. A poor choice of indexes leads to wasted disk usage and excessive consumption of system resources. This results in a decrease in database performance.

Introduce basics of index concept:

Index Basics

 Database indexes are typically created to facilitate better query performance.

• Performance without an index • Implementing an index to improve performance • Accessing only the index to return a query result set • Demonstrating when an index is ignored

Performance Without an Index To set up this example , suppose you

create a table to hold customer information, like so: create table cust (cust\_id number ,last\_name varchar2(30) ,first\_name varchar2(30));

insert into cust (cust\_id, last\_name, first\_name) values(1, 'STARK','JIM');

The performance of this query is initially very fast:

select cust\_id, last\_name, first\_name from cust where last\_name = 'STARK';

Run Again the query after insert 10000 rows performance decreased

If business grow rapidly customer increased

insert into cust

select level + 1

 ,dbms\_random.string('U',dbms\_random.value(3,15)) rand\_last\_name

 ,dbms\_random.string('U',dbms\_random.value(3,15)) rand\_first\_name

from dual connect by level < 100000;

 When generating an execution plan, the query optimizer relies on table and index statistics to generate an optimal execution plan. Statistics in this sense means information about the table and associated indexes, such as number of rows, number of blocks, number of distinct values, and so on.

1. The user submits a query. 2. The query is passed to an Oracle server process. 3. The optimizer is invoked. It creates an execution plan, which determines that a full table scan of the CUST table is required. 4. Since there is no index in place, every row of the table must be inspected to determine if it should be returned to the user, even though there is only one row in the table that will eventually be returned. 5. The table row is returned to the Oracle server process. 6. The Oracle server process returns the result set to the user.

Diagram

Description automatically generated

 Implementing an Index to Improve Performance

 index on columns that appear in the WHERE clause of a SQL query might improve performance, and then decide to create an index on the CUST table’s LAST\_NAME column

notice that the cost of the query has dramatically dropped and that the newly created index is being used to assist in the retrieval of data.

To conceptualize how the index improves performance, recall that an index stores two types of information: the value of the table column(s) and the corresponding ROWID . The ROWID uniquely identifies a row (for heap-organized tables) within a database and contains its physical location (data file, block, and row position within the block).

Index is created and subsequent queries execute, the query optimizer considers whether the index will reduce the amount of resources (cost) required to return the results of the query.

1. The user submits a query. 2. The query is passed to an Oracle server process. 3. The optimizer is invoked. It creates an execution plan that includes accessing the index. 4. The index is accessed to retrieve the ROWID of the table row of interest. 5. Using the ROWID , the row is located within a data file and block. 6. The table row is returned to the Oracle server process. 7. The Oracle server process returns the result set to the user.

Diagram

Description automatically generated

 In this case, it wouldn’t matter if there were millions and millions more records in the table; as long as the value contained in the index is fairly unique, Oracle is able to return the required rows with a minimal amount of disk reads.

In the prior example, suppose there are thousands of records in the CUST table but only one record in the table with the last name of STARK. The query optimizer can inspect the index, and within a few disk reads, locate the exact location (via the ROWID ) of the one block and row that contains the record of interest. This results in very fast performance

■ Tip The higher the degree of uniqueness, the more efficient a B-tree index becomes. In database jargon, a very selective (unique) column value compared to the total number of rows in a table is said to have high cardinality . Conversely, low cardinality refers to few unique values compared to the total rows for the table.

since the index contains all of the column values in the SELECT clause

( LAST\_NAME ), Oracle is able to satisfy the results of the query by accessing only the index. Oracle doesn’t have to read the table structure at all. When the SELECT clause columns are all contained with an index, this is known as a **covering index** .

Covering indexes are particularly efficient because only the index blocks need to be read.

Oracle Ignoring an Index

 Consider if the value in the LAST\_NAME column wasn’t very unique. Suppose that thousands of records exist in the CUST table with the same value of STARK for the LAST\_NAME . If the query optimizer did use the index, it would have to read from the index thousands of times, retrieve the ROWID s, and then also read from the table thousands of times. In this situation, it’s faster to bypass the index and instead scan every block in the table. For this reason, sometimes the optimizer calculates that the index isn’t beneficial to performance and ignores it.

truncate table cust;

insert into cust select level ,'STARK' ,'JIM' from dual connect by level <= 100000;

Next, statistics are generated and an execution plan is displayed: exec dbms\_stats.gather\_table\_stats( user, 'CUST' );

 Index Basics Wrap-up Before reading on, let’s review the concepts introduced up to this point in the chapter: • Indexes are optional database objects defined on a table and one or more columns. • Indexes are primarily used to improve query performance, but are also used to enforce primary/unique keys and help prevent certain table-locking scenarios. • Indexes consume resources (disk, CPU, memory). Too many indexes on a table significantly slows down the performance of INSERT , UPDATE , and DELETE statements. • The B-tree index is the default index type in Oracle. • Without an index in place, Oracle must inspect every row in the table (full table scan) to determine if the row is a candidate to be returned to a query. • A fairly unique column value compared to all other rows in a table (high cardinality) results in a very efficient B-tree index. In this situation, excellent performance is achieved even when tables contain millions of rows.

• In some situations, Oracle can retrieve data for a query by only accessing the index (covering index); the table doesn’t have to be accessed. • In some scenarios, the query optimizer chooses not to use an index. In other words, the query optimizer calculates that the cost of a full table scan is less than the cost when using an index

Index Type                     Usage

B- tree                 Default, balanced tree index; good for high-cardinality (high   degree of distinct values) columns. Use a normal B-tree index unless you have a    concrete reason to use a different index type or feature.

Index-organized table (IOT)   Efficient when most of the column values are included in the primary key. You access the index as if it were a table. The data is stored in a B-tree-like structure.

 Unique

          A form of B-tree index used to enforce uniqueness in column values. Often used with primary key and unique key constraints, but can be created independently of constraints.

Reverse key       A form of B-tree index useful to balance I/O in an index that has many sequential inserts. It has the disadvantage of not being able to perform range scans.

Key compressed     Good for concatenated indexes where the leading column is often repeated; compresses leaf block entries. This feature applies to a B-tree or an IOT index.

Bitmap

          Excellent in data warehouse environments with low-cardinality columns and SQL statements using many AND or OR operators in the WHERE clause. Bitmap indexes aren’t appropriate for online transaction processing (OLTP) databases where rows are frequently updated. You can’t create a unique bitmap index.

Bitmap join           Useful in data warehouse environments for queries that utilize star schema structures that join fact and dimension tables.

Indexed virtual column    An index defined on a virtual column (of a table). Useful for columns that have SQL functions applied to them. A viable alternative to using a function-based index.

Virtual                         Allows you to create an index with no physical segment or extents via the NOSEGMENT clause of CREATE INDEX . Useful in tuning SQL without consuming resources required to build the physical index. Any index type can be created as virtual.

Invisible                   The index is not visible to the query optimizer. However, the structure of the index is maintained as table data is modified. Useful for testing an index before making it visible to the application. Any index type can be created as invisible.

 Global partitioned A global index across all partitions in a partitioned table or regular table. This can be a B-tree index type and can’t be a bitmap index type.

 Local partitioned A local index based on individual partitions in a partitioned table. This can be either a B-tree or bitmap index type.

B-tree Indexes

==============

The default index type in Oracle is a B-tree index. This index type is very efficient for high-cardinality column values. For most applications, this index type is appropriate.

create index cust\_idx2 on cust(first\_name);

Unless you have verifiable performance reasons to use a different index type, use a B-tree. Too often, DBAs or developers read about a new indexing feature and assume that the vendor’s exaggeration of a feature matches the actual realized benefits

 There are several subtypes of B-tree indexes. • Index-organized table • Unique • Reverse key • Key compressed • Descending

Index-Organized Table

==================

An index-organized table (IOT) stores the entire contents of the table’s row in a B-tree index structure. An IOT provides fast access for queries that have exact matches and/or range searches on the primary key.

create table prod\_sku (prod\_sku\_id number ,sku varchar2(256), constraint prod\_sku\_pk primary key(prod\_sku\_id, sku) ) **organization index;**

**Unique Indexes**

=============

When creating a B-tree index, you can define it to be a unique [index.it](https://urldefense.com/v3/__http:/index.it__;!!CnbCE_LObTUL5LY!J8GHnCsZUVGGP0Mx7OKN_h6GdVBP7bT06jNW5iHulVfVuYoaXP4OY4Tq8AjjdA$) acts like a unique key constraint. When inserting into the corresponding table, the unique index guarantees that any non-null values inserted into the table are distinct.

create unique index cust\_uidx1 on cust(last\_name, first\_name);

 Reverse Key Indexes

==================

Reverse key indexes are useful to balance I/O in an index that has many sequential inserts in a table with a column managed by a sequence. Thus, when using a reverse key index, you avoid having I/O concentrated in one physical disk location within the index during large inserts of sequential values.

create index cust\_ridx1 on cust(cust\_id) reverse;

Reverse key indexes may be appropriate in Real Application Clusters (RAC) environments where the ID column is populated from a sequence.In RAC environments, performance can suffer if each instance is using sequence numbers close in value while doing inserts.

 Specifying a reverse key index spreads out the IO and reduces contention for the same blocks. The disadvantage of reverse key indexes is that it can’t be used for range scans.

 Key Compressed Indexes

================

A key compressed index is useful in reducing the storage and I/O requirements of concatenated indexes where the leading column is often repeated. Use the COMPRESS N clause to create a compressed index, as shown here:

 create index cust\_cidx\_1 on cust(last\_name, first\_name) compress 2;

Descending Indexes

 By default, Oracle stores B-tree indexes in ascending order. For example, if you have an index on a column with a number data type, the smallest numbers appear first in the index (the leftmost leaf node) and the highest numbers are stored in the rightmost leaf nodes.

       create index cust\_didx1 on cust(cust\_id desc);

 Descending indexes are useful for queries that sort some columns in ascending order and other columns in descending order.

 Bitmap Index

===================

 Bitmap indexes are commonly used in data warehouse environments. These indexes are recommended for columns with a relatively low number of distinct values (low cardinality). Bitmap indexes are also efficient for SQL statements that use multiple AND or OR join operators in the WHERE clause (which is typical in a data warehouse environment).

 we should not use bitmap indexes in OLTP databases with high INSERT / UPDATE / DELETE activities. because the structure of the bitmap index results in many locked rows during singular DML operations (which results in locking problems for high-transaction OLTP systems). A bitmap index is created using the BITMAP keyword.

**create table f\_sales( sales\_amt number ,d\_date\_id number ,d\_product\_id number ,d\_customer\_id number);**

**create bitmap index f\_sales\_fk1 on f\_sales(d\_date\_id);**

Bitmap indexes and bitmap join indexes are available only with the Oracle Enterprise Edition of the database.

 Bitmap Join

=====================

A bitmap join index stores the results of a join between two tables in an index. Bitmap join indexes are appropriate in situations where you’re joining two tables using the foreign key column(s) in one table that relate to primary key column(s) in another table.

Bitmap join indexes are usually suitable for data warehouse environments that have tables periodically batch loaded but then are not updated.

When updating tables that have bitmap join indexes, this potentially results in several rows being locked. Therefore, this type of index is not suitable for an OLTP database.

create table d\_customers (d\_customer\_id number primary key ,cust\_name varchar2(30));

create bitmap index f\_sales\_bmj\_idx1 on f\_sales(d\_customers.cust\_name) from f\_sales, d\_customers where f\_sales.d\_customer\_id = d\_customers.d\_customer\_id;

Function-Based Indexes

====================

 Function-based indexes are created with SQL functions or expressions in their definitions. Function-based indexes allow index lookups on columns referenced by SQL functions in the WHERE clause of a query.

create index cust\_fidx1 on cust(upper(last\_name));

 Indexed Virtual Column An alternative to a function-based index is to add a virtual column to a table and then create an index on that virtual column.

**Indexed Virtual Column**

An alternative to a function-based index is to add a virtual column to a table and then create an index on that virtual column.

create table inv( inv\_id number ,inv\_count number ,inv\_status generated always as ( case when inv\_count <= 100 then 'GETTING LOW' when inv\_count > 100 then 'OKAY' end) );

Now you can create a regular index on the virtual column, as follows:

**create index inv\_idx1 on inv(inv\_status);**

■ Note Virtual columns are only available in Oracle Database 11 g and higher.

Virtual Index You can instruct Oracle to create an index that will never be used and won’t have any extents allocated to it via the NOSEGMENT clause, as follows:

**create index cust\_idx1 on cust(first\_name) nosegment;**

Even though this index is not physically instantiated, you can instruct Oracle to determine if the index might be used by the optimizer via the \_USE\_NOSEGMENT\_INDEXES initialization parameter; for example: SQL> alter session set "\_use\_nosegment\_indexes"=true;

When would this be useful? Let’s suppose you have a very large index that you want to create without allocating space. To determine if the index would be used by the optimizer, you can create an index with NOSEGMENT , which allows you to test that scenario.

Global and Local Partitioned Indexes

===========================

A partitioned index is one where you have one logical index, but physically the index is implemented in several different segments. This allows for good performance even with very large databases. A partitioned index can be either global or local.

A global partitioned index is an index that uses a partitioning strategy that is not mapped to the underlying table’s segments. Global partitioned indexes are implemented as type B-tree and can be defined as unique. Use the GLOBAL PARTITION clause to create an index that is globally partitioned

create index f\_sales\_gidx1 on f\_sales(sales\_amt) global partition by range(sales\_amt) (partition pg1 values less than (25) ,partition pg2 values less than (50) ,partition pg3 values less than (maxvalue));

 A local partitioned index must be built on a partitioned table. This index type follows the same partitioning strategy as its underlying table.A given partition of a local partitioned index only contains values from its corresponding partition of the table. A local partitioned index can be either B-tree or bitmap.

create table f\_sales( sales\_amt number ,d\_date\_id number ,d\_product\_id number ,d\_customer\_id number) partition by range(sales\_amt)( partition p1 values less than (100) ,partition p2 values less than (1000) ,partition p3 values less than (maxvalue));

 create index f\_sales\_idx2 on f\_sales(d\_date\_id, sales\_amt) local;

An application domain index is custom to a specific application. This accommodates indexes on custom data types, documents, images, video, and spatial data.

 A B-tree cluster index is an index defined on a cluster table key. The B-tree cluster index associates a cluster key with a database block address. This index type is used with table clusters

A hash cluster is similarly used with cluster tables; the difference is that a hash cluster uses a hash function instead of the index key.

**Define a primary key constraint for each table** . This results in an index automatically being created on the columns specified in the primary key.

**• Create unique key constraints on columns** that are required to be unique and are different from the primary key columns . Each unique key constraint results in an index automatically being created on the columns specified in the constraint.

**• Manually create indexes on foreign key columns** . This is done for better performance and to avoid certain locking issues

**Indexes on Foreign Key Columns**

One reason is that foreign key columns are often referenced in WHERE clauses and therefore performance can be improved with these queries.

Another reason to index foreign keys is to reduce table-locking issues. Namely, if no index exists on the foreign key column, when deleting from a child table, the parent table becomes locked. This unnecessarily locks rows and reduces concurrency.

• Create indexes on columns used often as predicates in the WHERE clause; when multiple columns from a table are used in the WHERE clause, consider using a concatenated (multicolumn) index.

• Create a covering index on columns used in the SELECT clause.

• Consider creating indexes on columns used in the ORDER BY , GROUP BY , UNION , or DISTINCT clauses.

Guideline Reasoning

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

Create as many indexes as you need, but try to Indexes increase performance, but also

keep the number to a minimum. Add indexes space and processing resources. Don't add

judiciously. Test first to determine quantifiable indexes unnecessarily.

performance gains.

The required performance of queries you Indexing columns used in SQL queries helps

execute against a table should form the basis of performance the most.

your indexing strategy

Consider using the SQL Tuning Advisor for These tools provide recommendations and a

indexing recommendations second set

Create primary key constraints for all tables. This automatically creates a B-tree index (if the

Columns in the primary key aren't already indexed).

Create unique key constraints where This automatically creates a B-tree index (if the columns

appropriate. in the unique key aren't already indexed).

Create indexes on foreign key columns. Foreign key columns are usually included in the

WHERE clause when joining tables, and thus improve

performance of SQL SELECT statements. Creating a

B-tree index on foreign key columns also reduces locking

issues when updating/deleting from parent tables.

Carefully select and test indexes Even on small tables, indexes can sometimes perform

on small tables (small being less better than full table scans.

than a few thousand rows).

Use the correct type of index. Correct index usage maximizes performance.

B-tree indexes are suitable for most Use the basic B-tree index type if you don't

applications where have a verifiable a you have high-cardinality column values.

performance gain from using

Consider using bitmap indexes These indexes are ideal for low-cardinality columns where the

in data warehouse environments. values aren't updated often. Bitmap indexes work well on foreign key columns on star schema fact tables where you often run queries that use AND and OR join conditions.

Consider using a separate tablespace Table and index data may have different storage and/ or backup

for indexes (separate from tables). and recovery requirements. Using separate tablespaces lets you

manage indexes separately from tables. If you don't have these types of requirements, it's fine to place tables and indexes in the same tablespace.

Let the index inherit its storage

properties from the tablespace. This makes it easier to manage and maintain index storage.

Use consistent naming standards. This makes maintenance and troubleshooting easier.

Don't rebuild indexes unless you Rebuilding indexes is generally unnecessary unless an index is

have a solid reason to do so. corrupt or you want to move an index to different tablespace.

Monitor your indexes. Drop indexes Doing this frees up physical space and improves the performance

that aren't used. of data manipulation language (DML) statements.

Before dropping an index, consider This allows you to better determine if there are any

Marking It as unusable or invisible. These options let you rebuild or re-enable the index without

performance issues before requiring the data definition language (DDL) creation statement.

you drop the index.